



Puppet Enterprise 2019.2

Contents

Welcome to Puppet Enterprise® 2019.2.2.....	13
PE architecture.....	15
The master and compilers.....	16
The Puppet agent.....	16
Console services.....	17
Code Manager and r10k.....	19
Orchestration services.....	19
PE databases.....	20
Security and communications.....	20
Component versions in recent PE releases.....	21
Master and agent compatibility.....	23
Task compatibility.....	23
PE and open source version numbers.....	23
PE and FIPS compliance.....	24
Getting support.....	25
Puppet Enterprise support life cycle.....	25
The customer support portal.....	25
Getting support from the Puppet community.....	33
API index.....	33
Puppet platform documentation for PE.....	35
Help us localize docs.....	37
Create a localization account.....	37
Join an existing language team.....	37
Request a new language team.....	37
Questions?.....	37
 Release notes.....	 38
PE release notes.....	38
PE 2019.2.2.....	38
PE 2019.2.1.....	38
PE 2019.2.0.....	39
PE known issues.....	44
Installation and upgrade known issues.....	44
Supported platforms known issues.....	45
Configuration and maintenance known issues.....	45
Console and console services known issues.....	46
Orchestration services known issues.....	46
SSL and certificate known issues.....	47
Code management known issues.....	47
Internationalization known issues.....	49
 Getting started with Puppet Enterprise on *nix.....	 49
What is Puppet Enterprise (PE)?.....	49
Key components of PE.....	50
Install and administer Puppet Enterprise.....	51
Start installing PE on *nix.....	51
Start installing *nix agents.....	53

Manage your infrastructure.....	55
Restart a service using the console.....	55
Manage Apache services.....	57
Synchronize your clocks with NTP.....	65
Resolve nodes to your internal nameserver.....	68
Manage SSH keys and permissions.....	71
Control sudo privileges.....	74
Next steps.....	77

Getting started with Puppet Enterprise on Windows.....77

What is Puppet Enterprise (PE)?.....	77
Key components of PE.....	78
Install and administer Puppet Enterprise on Windows.....	79
Start installing PE in a Windows environment.....	79
Start installing Windows agents.....	81
Start installing modules.....	83
Start adding classes.....	84
Start assigning user access.....	84
Start writing modules for Windows.....	86
Manage your Windows infrastructure.....	88
Using the Windows module pack.....	89
Managing Windows configurations.....	103
Next steps.....	112

Installing.....112

Choosing an architecture.....	112
System requirements.....	114
Hardware requirements.....	114
Supported operating systems.....	118
Supported browsers.....	123
System configuration.....	124
What gets installed and where?.....	129
Software components installed.....	129
Executable binaries and symlinks installed.....	131
Modules and plugins installed.....	131
Configuration files installed.....	131
Tools installed.....	132
Databases installed.....	132
Services installed.....	132
User and group accounts installed.....	133
Log files installed.....	133
Certificates installed.....	135
Secret key file installed.....	135
Installing Puppet Enterprise.....	135
Download and verify the installation package.....	135
Install using express install.....	136
Install using text install.....	136
Configuration parameters and the pe.conf file.....	138
Purchasing and installing a license key.....	145
Getting a license.....	146
Install a license key.....	146
View license details for your environment.....	146
Installing agents.....	147
Using the install script	147

Install *nix agents.....	150
Install Windows agents.....	154
Install macOS agents.....	160
Install non-root agents.....	161
Managing certificate signing requests.....	164
Configuring agents.....	165
Installing compilers.....	165
How compilers work.....	165
Using load balancers with compilers.....	167
Install compilers.....	167
Configure compilers.....	168
Installing PE client tools.....	169
Supported PE client tools operating systems.....	170
Install PE client tools on a managed workstation.....	170
Install PE client tools on an unmanaged workstation.....	171
Configuring and using PE client tools.....	172
Installing external PostgreSQL.....	173
Install standalone PE-PostgreSQL.....	174
Install unmanaged PostgreSQL.....	174
Installing FIPS-compliant Puppet Enterprise.....	176
Uninstalling.....	177
Uninstall infrastructure nodes.....	177
Uninstall agents.....	178
Uninstaller options.....	178
Upgrading.....	179
Upgrading Puppet Enterprise.....	179
Upgrade paths.....	179
Upgrade cautions.....	180
Test modules before upgrade.....	182
Upgrade a standard installation.....	183
Migrate from a split to a standard installation.....	184
Upgrading PostgreSQL.....	185
Checking for updates.....	185
Upgrading agents.....	186
Upgrade *nix or Windows agents using the puppet_agent module.....	186
Upgrade a *nix or Windows agent using a script.....	187
Upgrading the agent independent of PE.....	188
Configuring Puppet Enterprise.....	189
Methods for configuring Puppet Enterprise.....	189
Configure settings using the console.....	189
Configure settings with Hiera.....	190
Configure settings in pe.conf.....	191
Configuring and tuning Puppet Server.....	191
Tune the maximum number of JRuby instances.....	191
Tune the maximum requests per JRuby instance.....	192
Tune the Ruby load path.....	192
Enable or disable cached data when updating classes.....	192
Change the environment_timeout setting.....	193
Add certificates to the puppet-admin certificate whitelist.....	193
Disable update checking.....	193
Puppet Server configuration files.....	194
pe-puppet-server.conf settings.....	194

Configuring and tuning the console.....	195
Configure the PE console and console-services.....	195
Manage the HTTPS redirect.....	197
Tuning the PostgreSQL buffer pool size.....	197
Enable data editing in the console.....	198
Configuring and tuning security settings.....	198
Configure cipher suites.....	198
Configure SSL protocols.....	199
Configure RBAC and token-based authentication settings.....	199
Configuring and tuning PuppetDB.....	200
Configure agent run reports.....	200
Configure command processing threads.....	201
Configure how long before PE stops managing deactivated nodes.....	201
Change the PuppetDB user password.....	201
Configure excluded facts.....	202
Configuring and tuning orchestration.....	202
Configure the orchestrator and pe-orchestration-services.....	202
Configure the PXP agent.....	203
Correct ARP table overflow.....	203
Configuring proxies.....	204
Downloading agent installation packages through a proxy.....	204
Setting a proxy for agent traffic.....	205
Setting a proxy for Code Manager traffic.....	205
Configuring Java arguments.....	205
Increase the Java heap size for PE Java services.....	206
Disable Java garbage collection logging.....	207
Configuring ulimit for PE services.....	207
Configure ulimit for PE services.....	207
Tuning standard installations.....	208
Master tuning.....	209
Compiler tuning.....	210
Using the puppet infrastructure tune command.....	210
Writing configuration files.....	211
Configuration file syntax.....	211
Analytics data collection.....	212
What data does Puppet Enterprise collect?.....	212
Opt out during the installation process.....	215
Opt out after installing.....	215
Static catalogs in Puppet Enterprise.....	216
Enabling static catalogs.....	216
Disabling static catalogs globally with Hiera.....	217
Using static catalogs without file sync.....	218
Configuring high availability.....	218
High availability.....	218
High availability architecture.....	219
What happens during failovers.....	219
System and software requirements.....	220
Classification changes in high availability installations.....	221
Load balancer timeout in high availability installations.....	222
Configure high availability.....	222
Provision a replica.....	223
Enable a replica.....	224
Promote a replica.....	225
Enable a new replica using a failed master.....	226

Forget a replica.....	227
Reinitialize a replica.....	227

Accessing the console.....227

Reaching the console.....	228
Accepting the console's certificate.....	228
Logging in.....	228
Generate a user password reset token.....	228
Reset the console administrator password.....	228
Troubleshooting login to the PE admin account.....	229

Managing access.....229

User permissions and user roles.....	229
Structure of user permissions.....	229
User permissions.....	230
Working with node group permissions.....	234
Best practices for assigning permissions.....	234
Creating and managing local users and user roles.....	235
Create a new user.....	235
Give a new user access to the console.....	235
Create a new user role.....	235
Assign permissions to a user role.....	235
Add a user to a user role.....	236
Remove a user from a user role.....	236
Revoke a user's access.....	236
Delete a user.....	236
Delete a user role.....	237
Connecting external directory services to PE.....	237
Connect to an external directory service.....	237
External directory settings.....	237
Verify directory server certificates.....	241
Working with user groups from an external directory service.....	241
Import a user group from an external directory service.....	241
Assign a user group to a user role.....	242
Delete a user group.....	242
Removing a remote user's access to PE.....	242
Token-based authentication.....	242
Generate a token using puppet-access.....	243
Generate a token using the API endpoint.....	245
Use a token with the PE API endpoints.....	245
Change the token's default lifetime.....	246
Setting a token-specific lifetime.....	246
Set a token-specific label.....	246
Revoking a token.....	247
Delete a token file.....	247
View token activity.....	247
RBAC API v1.....	247
Endpoints.....	247
Forming RBAC API requests.....	248
Users endpoints.....	249
User group endpoints.....	253
User roles endpoints.....	256
Permissions endpoints.....	259
Token endpoints.....	260

Directory service endpoints.....	262
Password endpoints.....	265
RBAC service errors.....	266
Configuration options.....	269
RBAC API v2.....	270
Tokens endpoints.....	271
Activity service API.....	274
Forming activity service API requests.....	274
Event types reported by the activity service.....	275
Events endpoints.....	278

Inspecting your infrastructure..... 280

Monitoring current infrastructure state.....	281
Node run statuses.....	281
Filtering nodes on the Overview page.....	283
Filtering nodes in your node list.....	284
Monitor PE services.....	285
Exploring your catalog with the node graph.....	285
How the node graph can help you.....	285
Investigate a change with the node graph.....	286
Viewing and managing all packages in use.....	286
Enable package data collection.....	287
View and manage package inventory.....	287
View package data collection metadata.....	287
Disable package data collection.....	288
Infrastructure reports.....	288
Working with the reports table.....	288
Filtering reports.....	289
Working with individual reports.....	290
Analyzing changes across Puppet runs.....	291
What is an event?.....	291
Event types.....	292
Working with the Events page.....	292
Viewing and managing Puppet Server metrics.....	294
Getting started with Graphite.....	294
Available Graphite metrics.....	298
Status API.....	302
Authenticating to the status API.....	303
Forming requests to the status API.....	303
JSON endpoints.....	303
Activity service plaintext endpoints.....	306
Metrics endpoints.....	307
The metrics API.....	313

Managing nodes..... 314

Adding and removing agent nodes.....	314
Managing certificate signing requests.....	314
Remove agent nodes.....	315
Adding and removing agentless nodes.....	316
Add agentless nodes to the inventory.....	316
Transport configuration options.....	317
Add devices to the inventory.....	318
Remove agentless nodes and devices from the inventory.....	318
How nodes are counted.....	319

Running Puppet on nodes.....	320
Running Puppet with the orchestrator.....	320
Running Puppet with SSH.....	320
Running Puppet from the console.....	321
Activity logging on console Puppet runs.....	321
Troubleshooting Puppet run failures.....	321
Grouping and classifying nodes.....	322
How node group inheritance works.....	322
Best practices for classifying node groups.....	322
Create node groups.....	322
Add nodes to a node group.....	323
Declare classes.....	325
Enable data editing in the console.....	326
Define data used by node groups.....	326
View nodes in a node group.....	328
Making changes to node groups.....	329
Edit or remove node groups.....	329
Remove nodes from a node group.....	329
Remove classes from a node group.....	329
Edit or remove parameters.....	330
Edit or remove variables.....	330
Environment-based testing.....	330
Test and promote a parameter.....	330
Test and promote a class.....	330
Testing code with canary nodes using alternate environments.....	330
Preconfigured node groups.....	331
All Nodes node group.....	331
Infrastructure node groups.....	331
Environment node groups.....	334
Designing system configs: roles and profiles.....	334
The roles and profiles method.....	334
Roles and profiles example.....	337
Designing advanced profiles.....	339
Designing convenient roles.....	356
Node classifier API v1.....	359
Forming node classifier requests.....	359
Groups endpoint.....	361
Groups endpoint examples.....	378
Classes endpoint.....	380
Classification endpoint.....	382
Commands endpoint.....	392
Environments endpoint.....	393
Nodes check-in history endpoint.....	394
Group children endpoint.....	397
Rules endpoint.....	400
Import hierarchy endpoint.....	401
Last class update endpoint.....	402
Update classes endpoint.....	402
Validation endpoints.....	403
Node classifier errors.....	406
Node classifier API v2.....	407
Classification endpoint.....	407
Node inventory API.....	410
Node inventory API: forming requests.....	410
POST /command/create-connection.....	410
POST /command/delete-connection.....	412

Command endpoint error responses.....	412
GET /query/connections.....	413
POST /query/connections.....	415
Query endpoint error responses.....	417

Orchestrating Puppet, tasks, and plans..... 418

Running jobs with Puppet orchestrator.....	419
Puppet orchestrator technical overview.....	419
Configuring Puppet orchestrator.....	421
Orchestration services settings.....	421
Setting PE RBAC permissions and token authentication for orchestrator.....	424
Enable cached catalogs for use with the orchestrator (optional).....	425
Orchestrator configuration files.....	426
PE Bolt services configuration.....	427
PE ACE services configuration.....	428
Disabling application management or orchestration services.....	428
Using Bolt with orchestrator.....	429
Direct Puppet: a workflow for controlling change.....	430
Direct Puppet workflow.....	430
Running Puppet on demand.....	434
Running Puppet on demand from the console.....	434
Running Puppet on demand from the CLI.....	439
Running Puppet on demand with the API.....	447
Tasks and plans in PE.....	450
Installing tasks and plans.....	451
Running tasks in PE.....	451
Running plans in PE.....	463
Writing tasks.....	467
Secure coding practices for tasks.....	468
Naming tasks.....	470
Sharing executables.....	471
Sharing task code.....	472
Writing remote tasks.....	474
Defining parameters in tasks.....	475
Returning errors in tasks.....	476
Structured input and output.....	477
Converting scripts to tasks.....	478
Supporting no-op in tasks.....	479
Task metadata.....	480
Specifying parameters.....	483
Writing plans.....	484
Writing plans in YAML.....	484
Writing plans in Puppet Language.....	495
Reviewing jobs.....	505
Review jobs from the console.....	505
Review jobs from the command line.....	508
Puppet orchestrator API v1 endpoints.....	509
Puppet orchestrator API: forming requests.....	509
Puppet orchestrator API: command endpoint.....	510
Puppet orchestrator API: events endpoint.....	518
Puppet orchestrator API: inventory endpoint.....	523
Puppet orchestrator API: jobs endpoint.....	525
Puppet orchestrator API: scheduled jobs endpoint.....	534
Puppet orchestrator API: plans endpoint.....	536
Puppet orchestrator API: plan jobs endpoint.....	538

Puppet orchestrator API: tasks endpoint.....	544
Puppet orchestrator API: root endpoint.....	547
Puppet orchestrator API: usage endpoint.....	548
Puppet orchestrator API: error responses.....	549

Managing and deploying Puppet code..... 550

Managing environments with a control repository.....	550
How the control repository works.....	550
Create a control repo from the Puppet template.....	551
Create an empty control repo.....	553
Add an environment.....	554
Delete an environment with code management.....	554
Managing environment content with a Puppetfile.....	555
The Puppetfile.....	555
Managing modules with a Puppetfile.....	555
Creating a Puppetfile.....	556
Create a Puppetfile.....	556
Change the Puppetfile module installation directory.....	556
Declare Forge modules in the Puppetfile.....	557
Declare Git repositories in the Puppetfile.....	557
Managing code with Code Manager.....	559
How Code Manager works.....	559
Configuring Code Manager.....	561
Customize Code Manager configuration in Hiera.....	565
Triggering Code Manager on the command line.....	572
Triggering Code Manager with a webhook.....	577
Triggering Code Manager with custom scripts.....	579
Troubleshooting Code Manager.....	581
Code Manager API.....	584
Managing code with r10k.....	594
Configuring r10k.....	594
Customizing r10k configuration.....	596
Deploying environments with r10k.....	603
r10k command reference.....	605
About file sync.....	606
File sync terms.....	606
How file sync works.....	607
Checking your deployments.....	608
Cautions.....	608

Provisioning with Razor.....609

How Razor works.....	609
Razor system requirements.....	611
Pre-requisites for machines provisioned with Razor.....	611
Setting up a Razor environment.....	611
Set up a Razor environment.....	611
Installing Razor.....	612
Install Razor.....	613
Using the Razor client.....	618
Using positional arguments with Razor client commands.....	618
Razor client commands.....	620
Protecting existing nodes.....	643
Protecting new nodes.....	643
Registering nodes.....	644

Limiting the number of nodes a policy can bind to.....	644
Provisioning a *nix node.....	644
What triggers provisioning.....	644
Provision for new users.....	644
Provision for advanced users.....	648
Viewing information about nodes.....	652
Provisioning a Windows node.....	653
What triggers provisioning.....	653
Provision a Windows node.....	653
Viewing information about nodes.....	658
Provisioning with custom facts.....	658
How the microkernel extension works.....	659
Microkernel extension configuration.....	659
Create the microkernel extension.....	659
Tips and limitations of the microkernel extension.....	659
Working with Razor objects.....	660
Repositories.....	660
Razor tasks.....	661
Tags.....	663
Policies.....	665
Brokers.....	666
Hooks.....	668
Keeping Razor scalable.....	673
Using the Razor API.....	673
Commands.....	674
Collections.....	675
Razor API reference.....	676
Upgrading Razor.....	688
Upgrade Razor from Puppet Enterprise 2015.2.x or later.....	688
Uninstalling Razor.....	689
Uninstall the Razor server.....	689
Uninstall the Razor client.....	689

SSL and certificates.....689

Regenerate certificates.....	689
Delete and recreate the certificate authority.....	690
Regenerate compiler certificates.....	690
Regenerate *nix agent certificates.....	691
Regenerate Windows agent certificates.....	691
Regenerate master certificates.....	692
Regenerate replica certificates.....	693
Use an independent intermediate certificate authority.....	693
Use a custom SSL certificate for the console.....	694
Change the hostname of a master.....	695
Generate a custom Diffie-Hellman parameter file.....	696
Enable TLSv1.....	697

Maintenance.....697

Backing up and restoring Puppet Enterprise.....	697
Back up your infrastructure.....	698
Restore your infrastructure.....	699
Backup and restore reference.....	700
Database maintenance.....	703
Databases in Puppet Enterprise.....	703

Optimize a database.....	704
List all database names.....	704

Troubleshooting.....704

Troubleshooting installation.....	704
Troubleshooting high availability.....	705
Troubleshooting puppet infrastructure run commands.....	706
Troubleshooting connections between components.....	706
Agents can't reach the Puppet master.....	706
Agents don't have signed certificates.....	707
Agents aren't using the master's valid DNS name.....	707
Time is out of sync.....	707
Node certificates have invalid dates.....	707
A node is re-using a certname.....	707
Agents can't reach the filebucket server.....	708
Orchestrator can't connect to PE Bolt server.....	708
Troubleshooting the databases.....	708
PostgreSQL is taking up too much space.....	708
PostgreSQL buffer memory causes installation to fail.....	708
The default port for PuppetDB conflicts with another service.....	709
puppet resource generates Ruby errors after connecting puppet apply to PuppetDB.....	709
Troubleshooting Windows.....	709
Installation fails.....	709
Upgrade fails.....	709
Errors when applying a manifest or doing a Puppet agent run.....	710
Error messages.....	711
Logging and debugging.....	712

Welcome to Puppet Enterprise® 2019.2.2

Puppet Enterprise (PE) helps you be productive, agile, and collaborative while managing your IT infrastructure. PE combines a model-driven approach with imperative task execution so you can effectively manage hybrid infrastructure across its entire lifecycle. PE provides the common language that all teams in an IT organization can use to successfully adopt practices such as version control, code review, automated testing, continuous integration, and automated deployment.

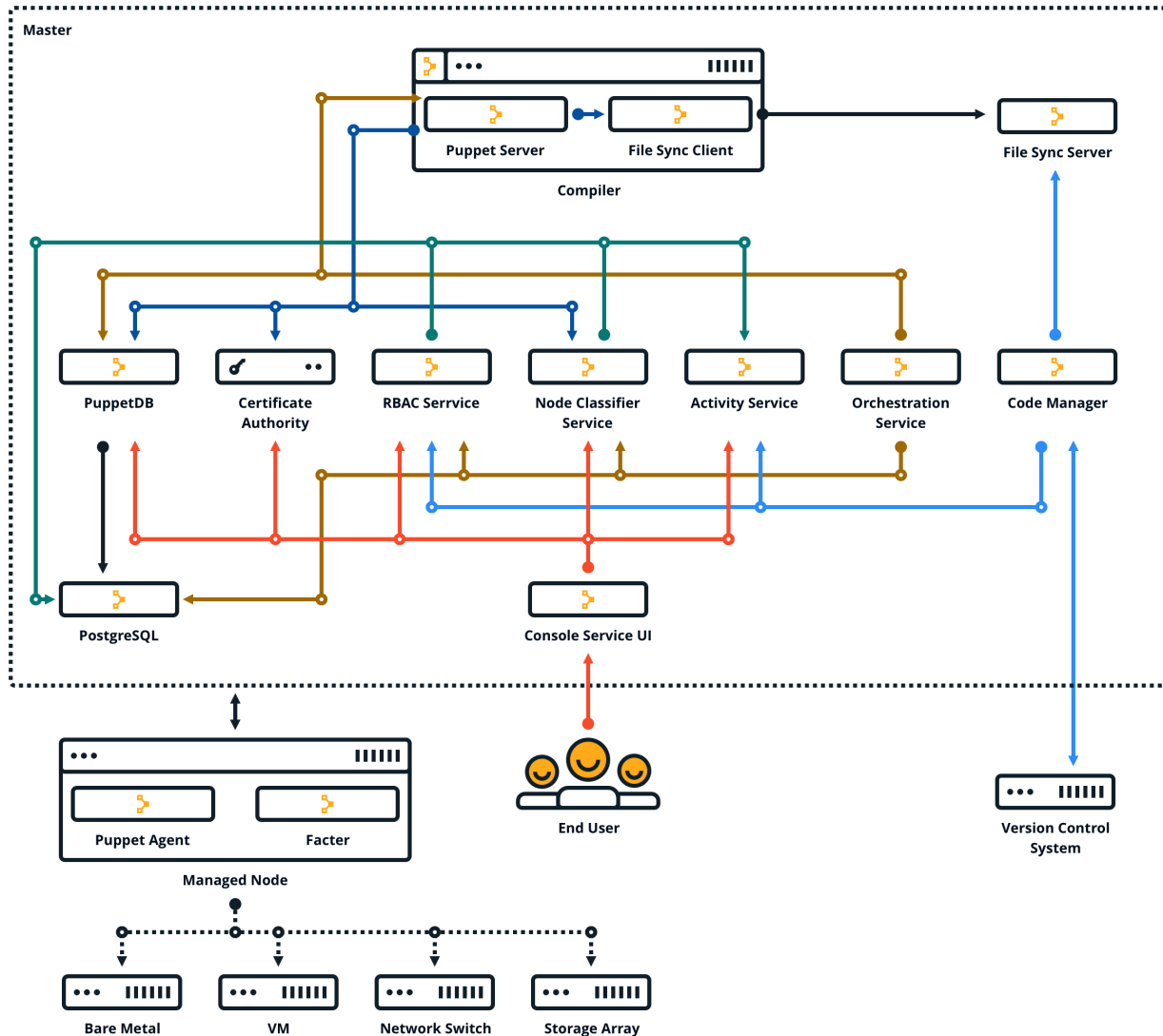
Helpful Puppet Enterprise docs links	Other useful places
<p>Before you upgrade or install</p> <p>Release notes - What's new, what's removed, what's resolved.</p> <p>System requirements - For hardware, software, browsers.</p> <p>What gets installed and where?</p> <p> System configuration - Set up the network, firewall, and ports.</p> <p>Administer and maintain PE</p> <p>Configuring and tuning PE - Improve performance and scale up.</p> <p>User permissions and roles - Manage access with role-based access control.</p> <p>SSL and certificates - Manage and regenerate security certificates.</p> <p>Backing up and restoring PE - Prepare for and recover from the worst case scenario.</p> <p>Learn the basics</p> <p>Getting started for Linux, for Windows - Walk through installing and doing basic operations with PE.</p> <p>Accessing the console - Find PE's management tools in the console.</p> <p>Inspecting your infrastructure - See what PE knows about your infrastructure.</p> <p>Manage your IT infrastructure</p> <p>Managing nodes - Use classification, groups, and environments to make changes to your infrastructure.</p> <p>Roles and profiles - Build a reliable, configurable, refactorable system for your infrastructure.</p> <p>Running jobs and tasks - Make controlled, on-demand changes.</p> <p>Managing and deploying Puppet code - Use Code Manager to stage, commit, and sync code changes to environments and modules</p> <p> API index on page 33 - A list of links to documentation for PE and Puppet APIs</p>	<p>Docs for related Puppet products</p> <p>Open source Puppet</p> <p>Continuous Delivery for Puppet Enterprise</p> <p>Puppet Remediate</p> <p>Bolt</p> <p>Puppet Development Kit</p> <p>Why and how people are using PE</p> <p>Read recent blog posts about Puppet Enterprise</p> <p>Find PE product information</p> <p>Download and try Puppet Enterprise on 10 nodes for free</p> <p>Learn PE and Puppet</p> <p>Learn at your own pace on a VM</p> <p>Plan your skill-building path with our learning roadmap</p> <p>Find an online, in-person or self-paced class</p> <p>Get certified</p> <p>Get support</p> <p>Search the Support portal and knowledge base</p> <p>Find out which PE versions are supported, and for how long</p> <p>Upgrade your support plan</p> <p>Share and contribute</p> <p>Engage with the Puppet community</p> <p>Puppet Forge - Find modules you can use, and contribute modules you've made to the community</p> <p>Open source projects from Puppet on Github</p>

To send us feedback or let us know about a docs error, [open a ticket](#) (you need a Jira account) or give the page a rating out of five stars and leave a comment.

PE architecture

Puppet Enterprise (PE) is made up of various components and services including the master and compilers, the Puppet agent, console services, Code Manager and r10k, orchestration services, and databases.

The following diagram shows the architecture of a typical PE installation.



Related information

[Component versions in recent PE releases](#) on page 21

These tables show which components are in Puppet Enterprise (PE) releases, covering recent long-term supported (LTS) releases. To see component version tables for a release that has passed its support phase, switch to the docs site for that release.

[PE and open source version numbers](#) on page 23

In October 2018, the "x.y.z" version numbering system for Puppet Enterprise (PE) changed so that the first number ("x") changes only when PE adopts a new major version of the Puppet Platform.

The master and compilers

The Puppet master is the central hub of activity and process in Puppet Enterprise. This is where code is compiled to create agent catalogs, and where SSL certificates are verified and signed.

PE infrastructure components are installed on a single node: the *master*. The master always contains a compiler and a Puppet Server. As your installation grows, you can add additional compilers to distribute the catalog compilation workload.

Each compiler contains the Puppet Server, the catalog compiler, and an instance of file sync.

Puppet Server

Puppet Server is an application that runs on the Java Virtual Machine (JVM) on the master. In addition to hosting endpoints for the certificate authority service, it also powers the catalog compiler, which compiles configuration catalogs for agent nodes, using Puppet code and various other data sources.

Catalog compiler

To configure a managed node, the agent uses a document called a catalog, which it downloads from the master or a compiler. The catalog describes the desired state for each resource that should be managed on the node, and it can specify dependency information for resources that should be managed in a certain order.

File sync

File sync keeps your code synchronized across multiple compilers. When triggered by a web endpoint, file sync takes changes from the working directory on the master and deploys the code to a live code directory. File sync then deploys that code to any compilers, ensuring that all masters in a multi-master configuration are kept in sync, and that your code is deployed only when it's ready.

Certificate Authority

The internal [certificate authority \(CA\) service](#) accepts certificate signing requests (CSRs) from nodes, serves certificates and a certificate revocation list (CRL) to nodes, and optionally accepts commands to sign or revoke certificates.

The CA service uses `.pem` files in the standard [ssldir](#) to store credentials. You can use the `puppetserver ca` command to interact with these credentials, including listing, signing, and revoking certificates.

Note: Depending on your architecture and security needs, the CA can be hosted either on the master or on its own node. The CA service on compilers is configured, by default, to proxy CA requests to the CA.

Related information

[Hardware requirements](#) on page 114

These hardware requirements are based on internal testing at Puppet and are meant only as guidelines to help you determine your hardware needs.

[Installing compilers](#)

[About file sync](#) on page 606

File sync helps Code Manager keep your Puppet code synchronized across multiple masters.

The Puppet agent

Managed nodes run the Puppet agent application, usually as a background service. The master and any compilers also run a Puppet agent.

Periodically, the agent sends facts to a master and requests a catalog. The master compiles the catalog using several sources of information, and returns the catalog to the agent.

After it receives a catalog, the agent applies it by checking each resource the catalog describes. If it finds any resources that are not in their desired state, it makes the changes necessary to correct them. (Or, in no-op mode, it reports on what changes would have been made.)

After applying the catalog, the agent submits a report to its master. Reports from all the agents are stored in PuppetDB and can be accessed in the console.

Puppet agent runs on *nix and Windows systems.

- [Puppet Agent on *nix Systems](#)
- [Puppet Agent on Windows Systems](#)

Factor

[Factor](#) is the cross-platform system profiling library in Puppet. It discovers and reports per-node facts, which are available in your Puppet manifests as variables.

Before requesting a catalog, the agent uses Factor to collect system information about the machine it's running on.

For example, the fact `os` returns information about the host operating system, and `networking` returns the networking information for the system. Each fact has various elements to further refine the information being gathered. In the `networking` fact, `networking.hostname` provides the hostname of the system.

Factor ships with a built-in list of [core facts](#), but you can build your own custom facts if necessary.

You can also use facts to determine the operational state of your nodes and even to group and classify them in the NC.

Console services

The console services includes the console, role-based access control (RBAC) and activity services, and the node classifier.

The console

The console is the web-based user interface for managing your systems.

The console can:

- browse and compare resources on your nodes in real time.
- analyze events and reports to help you visualize your infrastructure over time.
- browse inventory data and backed-up file contents from your nodes.
- group and classify nodes, and control the Puppet classes they receive in their catalogs.
- manage user access, including integration with external user directories.

The console leverages data created and collected by PE to provide insight into your infrastructure.

RBAC

In PE, you can use RBAC to manage user permissions. Permissions define what actions users can perform on designated objects.

For example:

- Can the user grant password reset tokens to other users who have forgotten their passwords?
- Can the user edit a local user's role or permissions?
- Can the user edit class parameters in a node group?

The RBAC service can connect to external LDAP directories. This means that you can create and manage users locally in PE, import users and groups from an existing directory, or do a combination of both. PE supports OpenLDAP and Active Directory.

You can interact with the RBAC and activity services through the console. Alternatively, you can use the RBAC service API and the activity service API. The activity service logs events for user roles, users, and user groups.

PE users generate tokens to authenticate their access to certain command line tools and API endpoints. Authentication tokens are used to manage access to the following PE services and tools: Puppet orchestrator, Code Manager, Node Classifier, role-based access control (RBAC), and the activity service.

Authentication tokens are tied to the permissions granted to the user through RBAC, and provide users with the appropriate access to HTTP requests.

Node classifier

PE comes with its own node classifier (NC), which is built into the console.

Classification is when you configure your managed nodes by assigning classes to them. **Classes** provide the Puppet code—distributed in modules—that enable you to define the function of a managed node, or apply specific settings and values to it. For example, you might want all of your managed nodes to have time synchronized across them. In this case, you would group the nodes in the NC, apply an NTP class to the group, and set a parameter on that class to point at a specific NTP server.

You can create your own classes, or you can take advantage of the many classes that have already been created by the Puppet community. Reduce the potential for new bugs and to save yourself some time by using existing classes from modules on the [Forge](#), many of which are approved or supported by Puppet, Inc.

You can also classify nodes using the NC API.

Related information

[Monitoring current infrastructure state](#) on page 281

When nodes fetch their configurations from the Puppet master, they send back inventory data and a report of their run. This information is summarized on the **Overview** page in the console.

[Managing access](#) on page 229

Role-based access control, more succinctly called RBAC, is used to grant individual users the permission to perform specific actions. Permissions are grouped into user roles, and each user is assigned at least one user role.

[Endpoints](#) on page 247

The role-based access control (RBAC) service enables you to manage users, directory groups, and roles.

[Activity service API](#) on page 274

The activity service logs changes to role-based access control (RBAC) entities, such as users, directory groups, and user roles.

[Token endpoints](#) on page 260

A user's access to PE services can be controlled using authentication tokens. Users can generate their own authentication tokens using the `token` endpoint.

[Setting PE RBAC permissions and token authentication for orchestrator](#) on page 424

Before you run any orchestrator jobs, you need to set the appropriate permissions in PE role-based access control (RBAC) and establish token-based authentication.

[Request an authentication token for deployments](#) on page 563

Request an authentication token for the deployment user to enable secure deployment of your code.

[Authenticating to the node classifier API](#) on page 360

You need to authenticate requests to the node classifier API. You can do this using RBAC authentication tokens or with the RBAC certificate whitelist.

[Forming RBAC API requests](#) on page 248

Token-based authentication is required to access the RBAC API. You can authenticate requests by using either user authentication tokens or whitelisted certificates.

[Forming activity service API requests](#) on page 274

Token-based authentication is required to access the activity service API. You can authenticate requests with user authentication tokens or whitelisted certificates.

[User permissions and user roles](#) on page 229

The "role" in role-based access control refers to a system of user roles, which are assigned to user groups and their users. Those roles contain permissions, which define what a user can or can't do within PE.

[Grouping and classifying nodes](#) on page 322

Configure nodes by assigning classes, parameters, and variables to them. This is called *classification*.

[Node classifier API v1](#) on page 359

These are the endpoints for the node classifier v1 API.

Code Manager and r10k

PE includes tools for managing and deploying your Puppet code: Code Manager and r10k.

These tools install modules, create and maintain [environments](#), and deploy code to your masters, all based on code you keep in Git. They sync the code to your masters, so that all your servers start running the new code at the same time, without interrupting agent runs.

Both Code Manager and r10k are built into PE, so you don't have to install anything, but you need to have a basic familiarity with Git.

Code Manager comes with a command line tool which you can use to trigger code deployments from the command line.

Related information

[Managing and deploying Puppet code](#) on page 550

Puppet Enterprise (PE) includes built-in tools for managing and deploying your Puppet infrastructure. Code Manager and r10k are code management tools that automatically install modules, create and maintain environments, and deploy new code to your masters, all based on version control of your Puppet code and data.

[Triggering Code Manager on the command line](#) on page 572

The `puppet-code` command allows you to trigger Code Manager from the command line to deploy your environments.

[Running jobs with Puppet orchestrator](#) on page 419

With the Puppet orchestrator, you can run two types of "jobs": on-demand Puppet runs or Puppet tasks.

Orchestration services

Orchestration services is the underlying toolset that drives Puppet Application Orchestration and the Puppet orchestrator.

Puppet Application Orchestration provides Puppet language extensions and command-line tools to help you configure and manage multi-service and multi-node applications. Specifically, application orchestration is:

- Puppet language elements for describing configuration relationships between components of a distributed application.

For example, in a three-tier stack application infrastructure—a load-balancer, an application/web server, and a database server—these servers have dependencies on one another. You want the application server to know where the database service is and how they connect, so that you can cleanly bring up the application. You then want the load balancer to automatically configure itself to balance demand on a number of application servers. And if you update the configuration of these machines, or roll out a new application release, you want the three tiers to reconfigure in the correct order

- A service that orchestrates ordered configuration enforcement from the node level to the environment level.

The orchestrator is a command-line tool for planning, executing, and inspecting orchestration jobs. For example, you can use it to review application instances declared in an environment, or to enforce change on the environment level without waiting for nodes to check in in regular 30-min intervals.

The orchestration service interacts with PuppetDB to retrieve facts about nodes. To run orchestrator jobs, users must first authenticate to Puppet Access, which verifies their user and permission profile as managed in RBAC.

PE databases

PE uses PostgreSQL as a database backend. You can use an existing instance, or PE can install and manage a new one.

The PE PostgreSQL instance includes the following databases:

Database	Description
pe-activity	Activity data from the Classifier, including who, what and when
pe-classifier	Classification data, all node group information
pe-puppetdb	Exported resources, catalogs, facts, and reports (see more, below)
pe-rbac	Users, permissions, and AD/LDAP info
pe-orchestrator	Details about job runs, users, nodes, and run results

PuppetDB

PuppetDB collects data generated throughout your Puppet infrastructure. It enables advanced features like exported resources, and is the database from which the various components and services in PE access data. Agent run reports are stored in PuppetDB.

See the PuppetDB overview for more information.

Related information

[Database maintenance](#) on page 703

You can optimize the Puppet Enterprise (PE) databases to improve performance.

Security and communications

The services and components in PE use a variety of communication and security protocols.

Service/Component	Communication Protocol	Authentication	Authorization
Puppet Server	HTTPS, SSL/TLS	SSL certificate verification with Puppet CA	trapperkeeper-auth
Certificate Authority	HTTPS, SSL/TLS	SSL certificate verification with Puppet CA	trapperkeeper-auth
Puppet agent	SSL/TLS	SSL certificate verification with Puppet CA	n/a
PuppetDB	HTTPS externally, or HTTP on the loopback interface	SSL certificate verification with Puppet CA	SSL certificate whitelist
PostgreSQL	PostgreSQL TCP, SSL for PE	SSL certificate verification with Puppet CA	SSL certificate whitelist
Activity service	SSL	SSL certificate verification with Puppet CA, token authentication	RBAC user-based authorization
RBAC	SSL	SSL certificate verification with Puppet CA, token authentication	RBAC user-based authorization

Service/Component	Communication Protocol	Authentication	Authorization
Classifier	SSL	SSL certificate verification with Puppet CA, token authentication	RBAC user-based authorization
Console Services UI	SSL	Session-based authentication	RBAC user-based authorization
Orchestrator	HTTPS, Secure web sockets	RBAC token authentication	RBAC user-based authorization
PXP agent	Secure web sockets	SSL certificate verification with Puppet CA	n/a
PCP broker	Secure web sockets	SSL certificate verification with Puppet CA	trapperkeeper-auth
File sync	HTTPS, SSL/TLS	SSL certificate verification with Puppet CA	trapperkeeper-auth
Code Manager	HTTPS; can fetch code remotely via HTTP, HTTPS, and SSH (via Git)	RBAC token authentication; for remote module sources, HTTP(S) Basic or SSH keys	RBAC user-based authorization; for remote module sources, HTTP(S) Basic or SSH keys
Razor	HTTPS, SSL	Apache Shiro authentication	Apache Shiro users and roles

Component versions in recent PE releases

These tables show which components are in Puppet Enterprise (PE) releases, covering recent long-term supported (LTS) releases. To see component version tables for a release that has passed its support phase, switch to the docs site for that release.

Puppet Enterprise agent and server components

This table shows the components installed on all agent nodes.

Note: 5 is a backwards-compatible evolution of , which is built into 4.9.0 and higher. To provide some backwards-compatible features, it uses the classic 3.x.x codebase version listed in this table.

Version	Agent				Resource API		Ruby	OpenSSL
2019.2.2	6.10.1	6.10.1	3.14.5	3.6.0	1.8.9	N/A	2.5.7	1.1.1d
2019.2.1	6.10.1	6.10.1	3.14.5	3.6.0	1.8.9	N/A	2.5.7	1.1.1d
2019.2.0	6.10.1	6.10.1	3.14.5	3.6.0	1.8.9	N/A	2.5.7	1.1.1d
2019.1.3	6.4.4	6.4.4	3.13.4	3.5.0	1.8.9	N/A	2.5.7	1.1.1d
2019.1.1	6.4.3	6.4.3	3.13.3	3.5.0	1.8.6	N/A	2.5.3	1.1.1a
2019.1.0	6.4.2	6.4.2	3.13.2	3.5.0	1.8.2	N/A	2.5.3	1.1.1a
2019.0.4	6.1.10	6.0.10	3.12.5	3.4.6	1.6.5	N/A	2.5.3	1.1.1a
2019.0.3	6.0.9	6.0.9	3.12.4	3.4.6	1.6.4	N/A	2.5.3	1.1.1a
2019.0.2	6.0.5	6.0.5	3.12.3	3.4.6	1.6.3	N/A	2.5.1	1.0.2n
2019.0.1	6.0.4	6.0.4	3.12.1	3.4.5	1.6.2	N/A	2.5.1	1.0.2n

Version	Agent				Resource API		Ruby		OpenSSL
2019.0.0	6.0.2	6.0.2	3.12.0	3.4.5	1.6.0	N/A	2.5.1		1.1.0h
2018.1.11 (LTS)	5.5.17	5.5.17	3.11.10	3.4.6	N/A	2.12.5	2.4.9		1.0.2t
2018.1.9	5.5.16	5.5.16	3.11.9	3.4.6	N/A	2.12.4	2.4.5		1.0.2r
2018.1.8	5.5.14	5.5.14	3.11.8	3.4.6	N/A	2.12.4	2.4.5		1.0.2r
2018.1.7	5.5.10	5.5.10	3.11.7	3.4.6	N/A	2.12.4	2.4.5		1.0.2n
2018.1.5	5.5.8	5.5.8	3.11.6	3.4.5	N/A	2.12.4	2.4.4		1.0.2n
2018.1.4	5.5.6	5.5.6	3.11.4	3.4.4	N/A	2.12.3	2.4.4		1.0.2n
2018.1.3	5.5.4	5.5.3	3.11.3	3.4.3	N/A	2.12.2	2.4.4		1.0.2n
2018.1.2	5.5.3	5.5.2	3.11.2	3.4.3	N/A	2.12.2	2.4.4		1.0.2n
2018.1.0	5.5.1	5.5.1	3.11.1	3.4.3	N/A	2.12.1	2.4.4		1.0.2n

This table shows components installed on server nodes.

Note: FIPS-compliant version 2019.2 includes version 9.6.

Version	Server				Services	Agentless Catalog Executor (ACE) Services	Java		Nginx	
2019.2.2	6.7.1	6.7.3	3.3.3	1.9.6	1.33.0	1.0.0	11.5	1.8.0	N/A	1.16.1
2019.2.1	6.7.1	6.7.3	3.3.3	1.9.6	1.33.0	1.0.0	11.5	1.8.0	N/A	1.16.1
2019.2.0	6.7.1	6.7.2	3.3.3	1.9.6	1.33.0	1.0.0	11.5	1.8.0	N/A	1.16.1
2019.1.3	6.3.2	6.3.6	3.2.3	1.9.5	1.34.0	1.0.0	9.6.15	1.8.0	N/A	1.16.1
2019.1.1	6.3.1	6.3.4	3.2.0	1.9.5	1.26.0	0.9.1	9.6.13	1.8.0	N/A	1.14.2
2019.1.0	6.3.0	6.3.2	3.2.0	1.9.5	1.17.0	0.9.1	9.6.12	1.8.0	N/A	1.14.2
2019.0.4	6.0.5	6.0.4	3.0.4	1.9.4	1.26.0	N/A	9.6.13	1.8.0	N/A	1.14.2
2019.0.3	6.0.4	6.0.3	3.0.3	1.9.4	1.15.0	N/A	9.6.12	1.8.0	N/A	1.14.2
2019.0.2	6.0.3	6.0.2	3.0.3	1.9.3	1.10.0	N/A	9.6.10	1.8.0	N/A	1.14.0
2019.0.1	6.0.2	6.0.1	3.0.3	1.9.3	1.1.0	N/A	9.6.10	1.8.0	N/A	1.14.0
2019.0.0	6.0.1	6.0.0	3.0.2	1.9.3	0.24.0	N/A	9.6.10	1.8.0	N/A	1.14.0
2018.1.11 (LTS)	5.3.10	5.2.11	2.6.7	1.9.2	N/A	N/A	9.6.15	1.8.0	5.15.5	1.16.1
2018.1.9	5.3.9	5.2.9	2.6.6	1.9.2	N/A	N/A	9.6.13	1.8.0	5.15.5	1.14.2
2018.1.8	5.3.8	5.2.8	2.6.5	1.9.2	N/A	N/A	9.6.12	1.8.0	5.15.5	1.14.2
2018.1.7	5.3.7	5.2.7	2.6.5	1.9.2	N/A	N/A	9.6.10	1.8.0	5.15.5	1.14.0
2018.1.5	5.3.6	5.2.6	2.6.5	1.9.2	N/A	N/A	9.6.10	1.8.0	5.15.5	1.14.0
2018.1.4	5.3.5	5.2.4	2.6.2	1.9.2	N/A	N/A	9.6.10	1.8.0	5.15.3	1.14.0
2018.1.3	5.3.4	5.2.4	2.6.2	1.9.2	N/A	N/A	9.6.8	1.8.0	5.15.3	1.14.0

Version				Server	Services	Agentless Catalog Executor (ACE) Services	Java		Nginx	
2018.1.2	5.3.3	5.2.2	2.6.2	1.9.2	N/A	N/A	9.6.8	1.8.0	5.15.3	1.12.1
2018.1.0	5.3.2	5.2.2	2.6.2	1.8.1	N/A	N/A	9.6.8	1.8.0	5.15.3	1.12.1

Master and agent compatibility

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

Master					
		PE 3.x Puppet 3.x	PE 2015.1 through 2017.2 Puppet 4.x	PE 2017.3 through 2018.1 Puppet 5.x	PE 2019.1 and later Puppet 6.x
Agent	3.x	#	#	#	#
	4.x		#	#	#
	5.x			#	#
	6.x				#

Note:

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

Task compatibility

This table shows which version of the Puppet task specification is compatible with each version of PE.

PE version	Puppet task specification (GitHub)
2019.0.1+	version 1, revision 3
2019.0.0+	version 1, revision 2
2017.3.0+	version 1, revision 1

PE and open source version numbers

In October 2018, the "x.y.z" version numbering system for Puppet Enterprise (PE) changed so that the first number ("x") changes only when PE adopts a new major version of the Puppet Platform.

We made this change to so that it's easier for you to determine whether a release contains potentially high-impact breaking changes. Previously, the "x" part of the version number incremented with each new calendar year, whether or not there were breaking changes in the release.

The "y" and "z" numbers continue to increment with minor and patch releases, respectively.

Type of change	Major "x" releases	Minor "y" releases	Patch "z" releases
Security updates	yes	yes	yes
Bug fixes	yes	yes	yes
Minor improvements	yes	yes	yes
Major improvements	yes	yes	no
New features	yes	yes	no
Low-impact breaking changes	yes	yes	no
High-impact breaking changes	yes	no	no

For example, if you are using PE 2018.1.3, then upgrading to PE 2019 likely introduces changes that involve updating your Puppet code, depending on what features you use and what we added. Upgrading to 2018.1.4 fixes bugs and security problems, and might slightly adjust how a feature looks.

Note: This version of documentation represents the latest update in this release stream. There might be differences in features or functionality from previous releases in this stream.

The first release to use the updated system is Puppet Enterprise 2019.0. Older release streams follow the previous version numbering system.

For information about active PE releases, mainstream and extended support, and end of life dates, see [Puppet Enterprise support lifecycle](#).

Open source version numbers

All of our open source projects—including Puppet, PuppetDB, Facter, and Hieradata—use semantic versioning ("semver"). This means that in an x.y.z version number, the "y" increases if new features are introduced and the "x" increases if existing features change or get removed.

Our semver refers to only the code within that project; it's possible that packaging or interactions with other projects might cause new behavior in a "z" upgrade of Puppet.

Note: In Puppet versions prior to 3.0.0 and Facter versions prior to 1.7.0, we didn't use semver.

PE and FIPS compliance

Beginning with version 2019.2.0, Puppet Enterprise (PE) complies with Federal Information Processing Standard (FIPS) 140-2 standards and is fully operable on select FIPS platforms.

Federal Information Processing Standard (FIPS) 140-2 is a standard defined by the U.S. government that defines the security requirements for cryptographic modules. The standard is published and maintained by the National Institute of Standards and Technology (NIST) and is mandated by the White House Office of Management and Budget (OMB) in [Circular A-130](#) and pursuant to FISMA (44 U.S.C Chapter 35). NIST operates the [Cryptographic Module Validation Program \(CMVP\)](#), which validates cryptographic modules per the requirements defined by the standard. All federal information systems that transmit and store sensitive information must utilize FIPS 140-2-validated cryptography.

As of version 2019.2.0, the cryptographic modules included in Puppet Enterprise are compliant with FIPS 140-2 and fully operable on the FIPS-compliant platforms listed below.

PE component	FIPS-compliant platforms
Master	Red Hat Enterprise Linux (RHEL) 7 in FIPS mode

PE component	FIPS-compliant platforms
Agents	Red Hat Enterprise Linux (RHEL) 7 in FIPS mode Windows Server 2012 R2 and newer versions in FIPS mode Windows 10 in FIPS mode

For information on installing a FIPS-compliant PE master and agents, see [Installing FIPS-compliant Puppet Enterprise](#).

Important: Upgrading from non-FIPS-compliant versions of PE to FIPS-compliant PE version 2019.2 is not supported.

In order to operate on FIPS platforms in compliance mode, PE version 2019.2.0 includes the following changes:

- All components are built and packaged against system OpenSSL (Red Hat Enterprise Linux (RHEL) 7 in FIPS mod for the master), or against OpenSSL built in FIPS mode (Windows 10, Windows Server 2012 R2 and newer versions for agents).
- All use of MD5 hashes for security has been eliminated and replaced.
- Forge and module tooling now use SHA-256 hashes to verify the identity of modules.
- Proper random number generation devices are used on all platforms.
- All Java and Clojure components use FIPS Bouncy Castle encryption providers on FIPS platforms.

Getting support

You can get commercial support for versions of PE in mainstream and extended support. You can also get support from our user community.

Puppet Enterprise support life cycle

Some of our releases have long term support (LTS); others have short term support (STS). For each release, there are three phases of product support: Mainstream support, Extended support, and End of Life (EOL).

For full information about types of releases, support phases and dates for each release, frequency of releases, and recommendations for upgrading, see the [Puppet Enterprise support lifecycle](#) page.

If the latest release with the most up-to-date features is right for you, [Download, try, or upgrade Puppet Enterprise](#). Alternatively, download an older supported release from [Previous releases](#).

Open source tools and libraries

PE uses open source tools and libraries. We use both externally maintained components (such as Ruby, PostgreSQL, and the JVM) and also projects which we own and maintain (such as Factor, Puppet agent, Puppet Server, and PuppetDB.)

Projects which we own and maintain are "upstream" of our commercial releases. Our open source projects move faster and have shorter support life cycles than PE. We might discontinue updates to our open source platform components before their commercial EOL dates. We vet upstream security and feature releases and update supported versions according to customer demand and our [Security policy](#).

The customer support portal

We provide responsive, dependable, quality support to resolve any issues regarding the installation, operation, and use of Puppet Enterprise (PE).

There are two levels of commercial support plans for PE: Standard and Premium. Both allow you to report your support issues to our confidential [customer support portal](#). When you purchase PE, you receive an account and log-on for the portal, which includes access to our knowledge base.

Note: We use the term "standard installation" to refer to a PE installation of up to 4,000 nodes. A Standard Support Plan is not limited to this installation type, however. Standard here refers to the support level, not the size of the PE installation.

Puppet Enterprise support script

When seeking support, you might be asked to run an information-gathering support script. This script collects a large amount of system information and PE diagnostics, compresses the data, and prints the location of the zipped tarball when it finishes running.

The script is provided by the `pe_support_script` module bundled with the installer.

Running the support script

Run the support script on the command line of your PE master or any agent node running Red Hat Enterprise Linux, Ubuntu, or SUSE Linux Enterprise Server operating systems with the command: `/opt/puppetlabs/bin/puppet enterprise support`.

Use these options when you run the support script to modify the output:

Option	Description
<code>--verbose</code>	Logs verbosely.
<code>--debug</code>	Logs debug information.
<code>--classifier</code>	Collects classification data.
<code>--dir <DIRECTORY></code>	Specifies where to save the support script's resulting tarball.
<code>--ticket <NUMBER></code>	Specifies a support ticket number for record-keeping purposes.
<code>--encrypt</code>	Encrypts the support script's resulting tarball with GnuPG encryption. Note: You must have GPG or GPG2 available in your <code>PATH</code> in order to encrypt the tarball.
<code>--log-age</code>	Specifies how many days worth of logs the support script collects. Valid values are positive integers or <code>all</code> to collect all logs, up to 1 GB per log.

The next iteration of the support script, version 3, is currently under development and may be selected by running `/opt/puppetlabs/bin/puppet enterprise support --v3`. This new version of the support script has more options, which can be used in addition to the list in the table above.

Option	Description
<code>--v3</code>	Activate version 3 of the support script. This option is required in order to use any other option listed in this table.
<code>--list</code>	List diagnostics that may be enabled or disabled. Diagnostics labeled "opt-in" must be explicitly enabled. All others are enabled by default.
<code>--enable <LIST></code>	A comma-separated list of diagnostic names to enable. Use the <code>--list</code> option to print available names. The <code>--enable</code> option must be used to activate diagnostics marked as "opt-in."

Option	Description
<code>--disable <LIST></code>	A comma-separated list of diagnostic names to enable. Use the <code>--list</code> option to print available names.
<code>--only <LIST></code>	A comma-separated list of diagnostic names to enable. All other diagnostics will be disabled. Use the <code>--list</code> option to print available names.
<code>--upload</code>	Upload the output tarball to Puppet Support via SFTP. Requires the <code>--ticket <NUMBER></code> option to be used.
<code>--upload_disable_host_key_check</code>	Disable SFTP host key checking. See Support article KB#0305 for a list of current host key values.
<code>--upload_user <USER></code>	Specify a SFTP user to use when uploading. If not specified, a shared write-only account will be used.
<code>--upload_key <FILE></code>	Specify a SFTP key to use with <code>--upload_user</code> .

Here are some examples of using the version 3 flags when running the support script:

```
# Collect diagnostics for just Puppet and Puppet Server
/opt/puppetlabs/bin/puppet enterprise support --v3 --only
puppet,puppetserver

# Enable collection of PE classification
/opt/puppetlabs/bin/puppet enterprise support --v3 --enable
pe.console.classifier-groups

# Disable collection of system logs, upload result to Puppet
/opt/puppetlabs/bin/puppet enterprise support --v3 --disable system.logs --
upload --ticket 12345
```

Descriptions of diagnostics that can be selected using the `--enable`, `--disable`, and `--only` flags are found in the next section.

Information collected by the support script

The following sections describe the information collected by version 1 and version 3 of the support script.

base-status

The `base-status` check collects basic diagnostics about the PE installation. This check is unique in that it is always enabled and is not affected by the `--disable` or `--only` flags.

Information collected by the `base-status` check:

- The version of the support script that is running, the Puppet ticket number, if supplied, and the time at which the script was run.

system

The checks in the `system` scope gather diagnostics, logs, and configuration related to the operating system.

Information collected by the `system.config` check:

- A copy of `/etc/hosts`
- A copy of `/etc/nsswitch.conf`
- A copy of `/etc/resolv.conf`
- Configuration for the `apt`, `yum`, and `dnf` package managers
- The operating system version
- The `umask` in effect

- The status of SELinux
- A list of configured network interfaces
- A list of configured firewall rules
- A list of loaded firewall kernel modules

Information collected by the `system.logs` check:

- A copy of the system log (syslog)
- A copy of the kernel log (dmesg)

Information collected by the `system.status` check:

- Values of variables set in the environment
- A list of running processes
- A list of enabled services
- The uptime of the system
- A list of established network connections
- NTP status
- The IP address and hostname of the node running the script, according to DNS
- Disk usage
- RAM usage

puppet

The checks in the `puppet` scope gather diagnostics, logs, and configuration related to the Puppet agent services.

Information collected by the `puppet.config` check:

- Factor configuration files:
 - `/etc/puppetlabs/facter/facter.conf`
- Puppet configuration files:
 - `/etc/puppetlabs/puppet/device.conf`
 - `/etc/puppetlabs/puppet/hiera.yaml`
 - `/etc/puppetlabs/puppet/puppet.conf`
- PXP agent configuration files:
 - `/etc/puppetlabs/pxp-agent/modules/`
 - `/etc/puppetlabs/pxp-agent/pxp-agent.conf`

Information collected by the `puppet.logs` check:

- Puppet log files:
 - `/var/log/puppetlabs/puppet`
- JournalD logs for the puppet service
- PXP agent log files:
 - `/var/log/puppetlabs/puppet`
- JournalD logs for the pxp-agent service

Information collected by the `puppet.status` check:

- `facter -p` output and debug-level messages
- A list of Ruby gems installed for use by Puppet
- Ping output for the Puppet Server the agent is configured to use
- A copy of the following files and directories from the Puppet `statedir`:
 - `classes.txt`
 - `graphs/`
 - `last_run_summary.yaml`

- A listing of metadata (name, size, etc.) of files present in the following directories:
 - `/etc/puppetlabs`
 - `/var/log/puppetlabs`
 - `/opt/puppetlabs`
- A listing of Puppet and PE packages installed on the system along with verification output for each

puppetserver

The checks in the `puppetserver` scope gather diagnostics, logs, and configuration related to the Puppet Server service.

Information collected by the `puppetserver.config` check:

- Puppet Server configuration files:
 - `/etc/puppetlabs/code/hiera.yaml`
 - `/etc/puppetlabs/puppet/auth.conf`
 - `/etc/puppetlabs/puppet/autosign.conf`
 - `/etc/puppetlabs/puppet/classfier.yaml`
 - `/etc/puppetlabs/puppet/fileserver.conf`
 - `/etc/puppetlabs/puppet/hiera.yaml`
 - `/etc/puppetlabs/puppet/puppet.conf`
 - `/etc/puppetlabs/puppet/puppetdb.conf`
 - `/etc/puppetlabs/puppet/routes.yaml`
 - `/etc/puppetlabs/puppetserver/bootstrap.cfg`
 - `/etc/puppetlabs/puppetserver/code-manager-request-logging.xml`
 - `/etc/puppetlabs/puppetserver/conf.d/`
 - `/etc/puppetlabs/puppetserver/logback.xml`
 - `/etc/puppetlabs/puppetserver/request-logging.xml`
 - `/etc/puppetlabs/r10k/r10k.yaml`
 - `/opt/puppetlabs/server/data/code-manager/r10k.yaml`

Information collected by the `puppetserver.logs` check:

- Puppet Server log files:
 - `/var/log/puppetlabs/puppetserver/`
- JournalD logs for the `pe-puppetserver` service
- `r10k` log files:
 - `/var/log/puppetlabs/r10k/`

Information collected by the `puppetserver.metrics` check:

- Data stored in `/opt/puppetlabs/puppet-metrics-collector/puppetserver`

Information collected by the `puppetserver.status` check:

- A list of certificates issued by the Puppet CA
- A list of Ruby gems installed for use by Puppet Server
- Output from the `status/v1/services` API
- Output from the `puppet/v3/environment_modules` API
- Output from the `puppet/v3/environments` API
- `environment.conf` and `hiera.yaml` files from each Puppet code environment
- The disk space used by Code Manager cache, storage, client, and staging directories
- The disk space used by the server's File Bucket
- The output of `r10k deploy display`

puppetdb

The checks in the `puppetdb` scope gather diagnostics, logs, and configuration related to the PuppetDB service.

Information collected by the `puppetdb.config` check:

- Configuration files:
 - `/etc/puppetlabs/puppetdb/bootstrap.cfg`
 - `/etc/puppetlabs/puppetdb/certificate-whitelist`
 - `/etc/puppetlabs/puppetdb/conf.d/`
 - `/etc/puppetlabs/puppetdb/logback.xml`
 - `/etc/puppetlabs/puppetdb/request-logging.xml`

Information collected by the `puppetdb.logs` check:

- PuppetDB log files: `/var/log/puppetlabs/puppetdb`
- JournalD logs for the `pe-puppetdb` service

Information collected by the `puppetdb.metrics` check:

- Data stored in `/opt/puppetlabs/puppet-metrics-collector/puppetdb`

Information collected by the `puppetdb.status` check:

- Output from the `status/v1/services` API
- Output from the `pdb/admin/v1/summary-stats` API
- A list of active certnames from the PQL query `nodes[certname] {deactivated is null and expired is null}`

pe

The checks in the `pe` scope gather diagnostics, logs, and configuration related to Puppet Enterprise services.

Information collected by the `pe.config` check:

- Installer configuration files:
 - `/etc/puppetlabs/enterprise/conf.d/`
 - `/etc/puppetlabs/enterprise/hiera.yaml`
 - `/etc/puppetlabs/installer/answers.install`
- PE client tools configuration files:
 - `/etc/puppetlabs/client-tools/orchestrator.conf`
 - `/etc/puppetlabs/client-tools/puppet-access.conf`
 - `/etc/puppetlabs/client-tools/puppet-code.conf`
 - `/etc/puppetlabs/client-tools/puppetdb.conf`
 - `/etc/puppetlabs/client-tools/services.conf`

Information collected by the `pe.logs` check:

- PE installer log files:
 - `/var/log/puppetlabs/installer/`
- PE backup and restore log files:
 - `/var/log/puppetlabs/pe-backup-tools/`
 - `/var/log/puppetlabs/puppet_infra_recover_config_cron.log`

Information collected by the `pe.status` check:

- Output from `puppet infra status`
- Current tuning settings from `puppet infra tune`
- Recommended tuning settings from `puppet infra tune`

This check is disabled by default. Information collected by the `pe.file-sync` check when activated by the `--enable` option:

- Puppet manifests and other content from `/etc/puppetlabs/code-staging/`
- Puppet manifests and other content stored in Git repos under `/opt/puppetlabs/server/data/puppetserver/filesync`

pe.console

The checks in the `pe.console` scope gather diagnostics, logs, and configuration related to the Puppet Enterprise console service.

Information collected by the `pe.console.config` check:

- Configuration files:
 - `/etc/puppetlabs/console-services/bootstrap.cfg`
 - `/etc/puppetlabs/console-services/conf.d/`
 - `/etc/puppetlabs/console-services/logback.xml`
 - `/etc/puppetlabs/console-services/rbac-certificate-whitelist`
 - `/etc/puppetlabs/console-services/request-logging.xml`
 - `/etc/puppetlabs/nginx/conf.d/`
 - `/etc/puppetlabs/nginx/nginx.conf`

Information collected by the `pe.console.logs` check:

- Console log files:
 - `/var/log/puppetlabs/console-services/`
 - `/var/log/puppetlabs/nginx/`
- JournalD logs for the `pe-puppetdb` and `pe-nginx` services

Information collected by the `pe.console.status` check:

- Output from the `/status/v1/services` API
- Directory service connection configuration, with passwords removed

This check is disabled by default. Information collected by the `pe.console.classifier-groups` check when activated by the `--enable` option:

- All classification data provided by the `/v1/groups` API endpoint

pe.orchestration

The checks in the `pe.orchestration` scope gather diagnostics, logs, and configuration related to the Puppet Enterprise orchestration services.

Information collected by the `pe.orchestration.config` check:

- ACE server configuration files:
 - `/etc/puppetlabs/puppet/ace-server/conf.d/`
- Bolt server configuration files:
 - `/etc/puppetlabs/puppet/bolt-server/conf.d/`

- Orchestration service configuration files:
 - `/etc/puppetlabs/puppet/orchestration-services/bootstrap.cfg`
 - `/etc/puppetlabs/puppet/orchestration-services/conf.d/analytics.conf`
 - `/etc/puppetlabs/puppet/orchestration-services/conf.d/auth.conf`
 - `/etc/puppetlabs/puppet/orchestration-services/conf.d/global.conf`
 - `/etc/puppetlabs/puppet/orchestration-services/conf.d/inventory.conf`
 - `/etc/puppetlabs/puppet/orchestration-services/conf.d/metrics.conf`
 - `/etc/puppetlabs/puppet/orchestration-services/conf.d/orchestrator.conf`
 - `/etc/puppetlabs/puppet/orchestration-services/conf.d/pcp-broker.conf`
 - `/etc/puppetlabs/puppet/orchestration-services/conf.d/web-routes.conf`
 - `/etc/puppetlabs/puppet/orchestration-services/conf.d/webserver.conf`
 - `/etc/puppetlabs/puppet/orchestration-services/logback.xml`
 - `/etc/puppetlabs/puppet/orchestration-services/request-logging.xml`

Information collected by the `pe.orchestration.logs` check:

- ACE server log files: `/var/log/puppetlabs/ace-server/`
- JournalD logs for the `pe-ace-server` service
- Bolt server log files: `/var/log/puppetlabs/bolt-server/`
- JournalD logs for the `pe-bolt-server` service
- Orchestrator log files: `/var/log/puppetlabs/orchestration-services/`
- JournalD logs for the `pe-orchestration-services` service

Information collected by the `pe.orchestration.metrics` check:

- Data stored in `/opt/puppetlabs/puppet-metrics-collector/orchestrator/`

Information collected by the `pe.orchestration.status` check:

- Output from the `/status/v1/services` API

pe.postgres

The checks in the `pe.postgres` scope gather diagnostics, logs, and configuration related to the Puppet Enterprise PostgreSQL database.

Information collected by the `pe.postgres.config` check:

- Configuration files:
 - `/opt/puppetlabs/server/data/postgresql/*/data/postgresql.conf`
 - `/opt/puppetlabs/server/data/postgresql/*/data/postmaster.opts`
 - `/opt/puppetlabs/server/data/postgresql/*/data/pg_ident.conf`
 - `/opt/puppetlabs/server/data/postgresql/*/data/pg_hba.conf`

Information collected by the `pe.postgres.logs` check:

- PostgreSQL log files:
 - `/var/log/puppetlabs/postgresql/*/`
 - `/opt/puppetlabs/server/data/postgresql/pg_upgrade_internal.log`
 - `/opt/puppetlabs/server/data/postgresql/pg_upgrade_server.log`
 - `/opt/puppetlabs/server/data/postgresql/pg_upgrade_utility.log`
- JournalD logs for the `pe-postgresql` service

Information collected by the `pe.postgres.status` check:

- A list of setting values that the database is using while running
- A list of currently established database connections and the queries being executed
- A distribution of Puppet run start times for thundering herd detection

- The status of any configured replication slots
- The status of any active replication connections
- The size of database directories on disk
- The size of databases as reported by the database service
- The size of tables and indices within databases

Getting support from the Puppet community

As a Puppet Enterprise customer you are more than welcome to participate in our large and helpful open source community as well as report issues against the open source project.

- Join the [Puppet Enterprise user group](#). Your request to join is sent to Puppet, Inc. for authorization and you receive an email when you've been added to the user group.
 - Click on "Sign in and apply for membership."
 - Click on "Enter your email address to access the document."
 - Enter your email address.
- Join the open source [Puppet user group](#).
- Join the [Puppet developers group](#).
- Report issues with the [open source Puppet project](#).

API index

APIs allow you to interact with Puppet and Puppet Enterprise (PE) applications from your own code or application integration hooks.

Puppet Enterprise APIs

For information on port requirements, see the [System Configuration](#) documents.

API	Useful for
Node inventory API on page 410	<ul style="list-style-type: none"> • Making HTTP(S) requests to the Puppet inventory service API. • Creating and deleting connection entries in the inventory service database. • Listing the connections entries in the inventory database.
RBAC service API v1	<ul style="list-style-type: none"> • Managing access to Puppet Enterprise. • Connecting to external directories. • Generating authentication tokens. • Managing users, user roles, user groups, and user permissions.
RBAC service API v2	<ul style="list-style-type: none"> • Revoking authentication tokens.
Node classifier service API	<ul style="list-style-type: none"> • Querying the groups that a node matches. • Querying the classes, parameters, and variables that have been assigned to a node or group. • Querying the environment that a node is in.
Orchestrator API	<ul style="list-style-type: none"> • Gathering details about the orchestrator jobs you run. • Inspecting applications and applications instances in your Puppet environments.

API	Useful for
Code Manager API	<ul style="list-style-type: none"> • Creating a webhook to trigger Code Manager. • Queueing Puppet code deployments. • Checking Code Manager and file sync status.
Status API	<ul style="list-style-type: none"> • Checking the health status of PE services.
Activity service API	<ul style="list-style-type: none"> • Querying PE service and user events logged by the activity service.
Razor API	<ul style="list-style-type: none"> • Provisioning bare-metal machines.

Open source Puppet Server, Puppet, PuppetDB, and Forge APIs

API	Useful for
Puppet Server administrative API endpoints <ul style="list-style-type: none"> • environment-cache • jruby-pool 	<ul style="list-style-type: none"> • Deleting environment caches created by a Puppet master. • Deleting the Puppet Server pool of JRuby instances.
Server-specific Puppet API <ul style="list-style-type: none"> • Environment classes • Environment modules • Static file content 	<ul style="list-style-type: none"> • Getting the classes and parameter information that is associated with an environment, with cache support. • Getting information about what modules are installed in an environment. • Getting the contents of a specific version of a file in a specific environment.
Puppet Server status API	<ul style="list-style-type: none"> • Checking the state, memory usage, and uptime of the services running on Puppet Server.
Puppet Server metrics API <ul style="list-style-type: none"> • v1 metrics (deprecated) • v2 metrics (Jolokia) 	<ul style="list-style-type: none"> • Querying Puppet Server performance and usage metrics.
Puppet HTTP API	<ul style="list-style-type: none"> • Retrieving a catalog for a node • Accessing environment information <p>For information on how this API is used internally by Puppet, see Agent/Master HTTPs Communications page</p>
Certificate Authority (CA) API	<ul style="list-style-type: none"> • Used internally by Puppet to manage agent certificates. <p>See Agent/Master HTTPs Communications for details. See Puppet's Commands page for information about the Puppet Cert command line tool.</p>
PuppetDB APIs	<ul style="list-style-type: none"> • Querying the data that PuppetDB collects from Puppet • Importing and exporting PuppetDB archives • Changing the PuppetDB model of a population • Querying information about the PuppetDB server • Querying PuppetDB metrics

API	Useful for
Forge API	<ul style="list-style-type: none"> Finding information about modules and users on the Forge Writing scripts and tools that interact with the Forge website

Puppet platform documentation for PE

Puppet Enterprise (PE) is built on the Puppet platform which has several components: Puppet, Puppet Server, Facter, Hieradata, and PuppetDB. This page describes each of these platform components, and links to the component docs.

Puppet

- [Puppet docs](#)

Puppet is the core of our configuration management platform. It consists of a programming language for describing desired system states, an agent that can enforce desired states, and several other tools and services.

Right now, you're reading the PE manual; the Puppet reference manual is a separate section of our docs site. After you've followed a link there, you can use the navigation sidebar to browse other sections of the manual.

Note: The Puppet manual has information about installing the open source release of Puppet. As a PE user, ignore those pages.

The following pages are good starting points for getting familiar with Puppet:

Language

- [An outline of how the Puppet language works.](#)
 - [Resources](#), [variables](#), [conditional statements](#), and [relationships and ordering](#) are the fundamental pieces of the Puppet language.
- [Classes](#) and [defined types](#) are how you organize Puppet code into useful chunks. Classes are the main unit of Puppet code you'll be interacting with on a daily basis. You can assign classes to nodes in the PE console.
- [Facts and built-in variables](#) explains the special variables you can use in your Puppet manifests.

Modules

- Most Puppet code goes in modules. We explain how modules work [here](#).
- There are also guides to [installing modules](#) and [publishing modules](#) on the Forge.
- Use the code management features included in PE to control your modules instead of installing by hand. See Managing and deploying Puppet code (in the PE manual) for more details.

Services and commands

- [An overview of Puppet's architecture.](#)
- [A list of the main services and commands you'll interact with.](#)
- [Notes on running Puppet's commands on Windows.](#)

Built-in resource types and functions

- [The resource type reference](#) has info about all of the built-in Puppet resource types.
- [The function reference](#) does the same for the built-in functions.

Important directories and files

- Most of your Puppet content goes in environments. Find out more about environments [here](#).
- The [codedir](#) contains code and data and the [confdir](#) contains config files. The [modulepath](#) and the [main manifest](#) both depend on the current environment.

Configuration

- The main config file for Puppet is `/etc/puppetlabs/puppet/puppet.conf`. Learn more about [Puppet's settings](#), and [about puppet.conf](#) itself.
- There are also a bunch of other config files used for special purposes. Go to the [page about puppet.conf](#) and check the navigation sidebar for a full list.

Puppet Server

- [Puppet Server docs](#)

Puppet Server is the JVM application that provides the core Puppet HTTPS services. Whenever Puppet agent checks in to request a configuration catalog for a node, it contacts Puppet Server.

For the most part, PE users don't need to directly manage Puppet Server, and the Puppet reference manual (above) has all the important info about how Puppet Server evaluates the Puppet language and loads environments and modules. However, some users might need to access the [environment cache](#) and [JRuby pool](#) administrative APIs, and there's lots of interesting background information in the rest of the Puppet Server docs.

Note: The Puppet Server manual has information about installing the open source release of Puppet Server. As a PE user, ignore those pages. Additionally, the Puppet Server config files in PE are managed with a built-in Puppet module; to change most settings, set the appropriate class parameters in the console.

Facter

- [Facter docs](#)

Facter is a system profiling tool. Puppet agent uses it to send important system info to Puppet Server, which can access that info when compiling that node's catalog.

- For a list of variables you can use in your code, check out the [core facts reference](#).
- You can also write your own custom facts. See the [custom fact overview](#) and the [custom fact walkthrough](#).

Hiera

- [Hiera docs](#)

Hiera is a hierarchical data lookup tool. You can use it to configure your Puppet classes.

Start with the [overview](#) and use the navigation sidebar to get around.

Note: Hiera 5 is a backwards-compatible evolution of Hiera, which is built into Puppet. To provide some backwards-compatible features, it uses the classic Hiera 3 codebase. This means "Hiera" is still version 3.x, even though this Puppet Enterprise version uses Hiera 5.

PuppetDB

- [PuppetDB docs](#)

PuppetDB collects the data Puppet generates, and offers a powerful query API for analyzing that data. It's the foundation of the PE console, and you can also use the API to build your own applications.

If you're interacting with PuppetDB directly, you'll mostly be using the query API.

- [The query tutorial page](#) walks you through the process of building and executing a query.
- [The query structure page](#) explains the fundamentals of using the query API.
- [The cURL tips page](#) has useful information about testing the API from the command line.
- You can use the navigation sidebar to browse the rest of the query API docs.

Note: The PuppetDB manual has information about installing the open source release of PuppetDB. As a PE user, ignore those pages.

Related information

[Managing and deploying Puppet code](#) on page 550

Puppet Enterprise (PE) includes built-in tools for managing and deploying your Puppet infrastructure. Code Manager and r10k are code management tools that automatically install modules, create and maintain environments, and deploy new code to your masters, all based on version control of your Puppet code and data.

Help us localize docs

The Technical Publications team at Puppet is pursuing a long-term plan to translate our user documentation into as many languages as possible, but we need your help!

If you are interested in contributing to localization of our documentation into your native language, you can either join an existing language team, or request the formation of a new language team.

We currently have the following language teams:

- Japanese
- Spanish
- German

Create a localization account

These steps describe how to create an account in our translation management tool, and get started translating content.

1. Select the project you are interested in contributing to.

Currently we have these:

- Open source Puppet documentation
 - Puppet Enterprise documentation
2. To access these projects, [create an account with Transifex](#), our translation management software, and then navigate to either [Open source Puppet](#) or [Puppet Enterprise](#) documentation.
 3. Click **Help Translate (project name)** and get started translating.

Join an existing language team

Puppet has some existing language projects already set up.

After you've logged in to Transifex and accessed the project, click **Join team** and select the language team that you're interested in.

Request a new language team

You can request a new language if we are not yet supporting your language.

If your language doesn't exist as one of our options yet, log in to Transifex, follow the links in **Create an account** to access your project of interest, and click **Request language**.

Questions?

Contact us with your localization questions.

If you have questions about contributing to localization at Puppet, please contact our localization team at localization@puppet.com.

Release notes

These release notes contain important information about Puppet Enterprise® 2019.2.

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

Note: This version of documentation represents the latest update in this release stream. There might be differences in features or functionality from previous releases in this stream.

PE uses certain components of open source Puppet. Refer to the component release notes for information about those releases.

Open source component	Version used in PE
and the agent	6.10.1
	6.7.3
	6.7.1
(optional)	latest

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

Documentation for end-of-life and superseded product versions are archived at <https://github.com/puppetlabs/docs-archive>.

PE release notes

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

PE 2019.2.2

Resolved issues

Orchestration service shuts down after code deploy

The initial code deploy after upgrade could cause the orchestration service to shut down if there were errors with the JRuby interpreter. Improved error handling now ensures that the orchestration service remains operable for Puppet run and task run functionality even if there are JRuby errors.

The notice function is now available for plans

The `notice` function did not work for the initial release of plans for PE. Now, `notice` is fully functional with plans.

PE 2019.2.1

Resolved issues

Upgrade failed if PuppetDB was offline

When upgrading to this version, installation failed during validation with an error about connecting to PuppetDB.

Upgrade could fail in environments with external PE-PostgreSQL

When you ran the installer in an environment with external PE-PostgreSQL, if the orchestrator database username was not `pe-orchestrator`, upgrade on the master failed.

Upgrade failed if IPv6 was disabled

If IPv6 was disabled on your system, upgrade failed with a `Protocol family unavailable` error. With this release, upgrade uses IPv4 by default. If you prefer to use IPv6, you can modify `"puppet_enterprise::ip_version": 6` in your `pe.conf` file before you upgrade.

PE 2019.2.0

New features

Manage network devices with Puppet Enterprise

You can add network devices to a PE deployment using the **Inventory** option on the console. This feature requires that you have installed modules for device transports in your production environment. By adding connections to those devices to your inventory, you can manage switches and firewalls, and run Puppet and tasks on them, just like other agentless nodes in your infrastructure. For more information on how to add devices and run jobs on them, see

- [Add devices to the inventory](#) on page 318
- [Run a task on a node list](#) on page 452
- [Run Puppet on a node list](#) on page 434



CAUTION: This initial implementation of network device configuration management is limited in how many device connections it can handle. Managing more than 100 devices can cause performance issues on the master. We plan to gather feedback from this release and improve the scaling and performance capabilities. In the meantime, avoid impacting master performance by limiting the total number of network device connections you make to 100.

Plans in PE

You can run plans in the PE console and on the command line. For more information, see [Running plans in PE](#) on page 463.

Agent installation from the console

You can install Puppet agents on target nodes from the **Inventory** option on the console. For more information, see [Install agents from the console](#) on page 53.

View plan content

You can view plan details and information about plan jobs you have run using the `/plan_jobs` [endpoints](#).

Added 'check node-connections endpoints

Added the endpoints `/ssh/check_node_connections` and `/winrm/check_node_connections` to `bolt-server` to support the `wait_until_available` feature in the console for plans. See the [developer docs](#) for more information.

FIPS compliance

This version of Puppet Enterprise complies with Federal Information Processing Standard (FIPS) 140-2 standards and is fully operable on select FIPS platforms. For more information on FIPS-compatible PE, see [PE and FIPS compliance](#).

Enhancements

PostgreSQL 11 upgrade

PE version 2019.2 upgrades PostgreSQL to version 11. If you are upgrading to PE version 2019.2, carefully review the information on datastore migration in [PostgreSQL 11 upgrade in PE 2019.2](#) before you begin. If you have an external PostgreSQL instance, you must upgrade it before you upgrade your master, compilers, and agents.

List node groups by name for a given node

Using an updated classification endpoint for the node classifier API v2, you can generate an array of the groups that a node was classified into and sort the list by group name and ID. Previously group lists were limited to ID.

```
"groups": [ { "id": "9c0c7d07-a199-48b7-9999-3cdf7654e0bf",
              "name": "a group" },
            { "id": "96d1a058-225d-48e2-a1a8-80819d31751d",
              "name": "b group" } ],
```

In addition to the new classifier endpoint, a new top-scope variable is available, `$pe_node_groups`, to use in your Puppet code. This variable contains the same content as the new groups value in the classifier, the names and group identifiers for the groups that were used to classify the nodes.

Important: `$pe_node_groups` cannot be interpolated when used as a classifier in class variable values.

For more information, see [POST /v2/classified/nodes/<name>](#) on page 407.

Specify alternate DNS names when regenerating certificates

When regenerating agent or master certificates, you can now pass an optional `dns_alt_names` parameter to the Bolt task.

Job export file contains new error messages column

You can now view full error messages for failed job runs on the job export file. The new column is called **Job node status detail**.

More RBAC control for plan permissions

This version implements `plan:run<planname>` and `plan:run:*` (all plans) to allow for more control over permissions in plans.

Additional settings can be configured in the console

The orchestrator now contains a new JRuby pool to facilitate plans in PE. New configuration options are now available as orchestrator settings to provide control over the JRuby pool.

You can now access the following settings in the orchestrator:

- `jruby_borrow_timeout` - how long to wait for a JRuby instance before timing out
- `jruby_compile_mode` - the compile mode that JRuby is in, can be 'jit' or 'off'
- `jruby_max_active_instances` - the max number of concurrently running JRuby instances at any one time.
- `jruby_max_requests_per_instance` - the max number of times a JRuby instance is re-used before it is destroyed and replaced by a new one.
- `reserved_code_cache` - a JVM argument that has been exposed to help better configure the JVM arguments in orchestrator.

Execute Bolt actions for pxp-agent

Previously, running a command over PCP via pxp-agent required wrapping the command in a temporary task wrapper via `bolt_shim` because pxp-agent was not able to respond to any other types of PCP requests for Bolt actions.

Now, pxp-agents can respond to the following requests over PCP without formatting them as tasks:

- `command run`
 - Please note, the executable `/opt/puppetlabs/puppet/bin/task_wrapper` has moved to `/opt/puppetlabs/puppet/bin/execution_wrapper`. Anyone using this executable directly should update their paths accordingly. Its functionality remains the same as before -- this is only a name change to reflect that it is used for more than one type of Bolt action. Formerly, this was just `task run`, now it is `command run` as well.
- `download file`
- `run script`

CLI update for `puppet plan run`

This change modifies `puppet-plan` CLI to use the same flag names as `puppet-task` for options that are the same across both tools. It also modifies `puppet-plan` so that it is able to read configured values for the CA cert path, the token, and the orchestrator service URL from files created by `pe-client-tools`.

FIPS Bouncy Castle for Puppet Server

Puppet Server will now use the FIPS version Bouncy Castle when running on systems with FIPS enabled.

Support script version 3

PE version 2019.2 introduces version 3 of the Puppet Enterprise support script. By adding the optional `--v3` flag to the support script command, you can access new options that allow you to fine-tune the information you gather and send to the Support team. For more information, see [Puppet Enterprise support script](#).

Architecture terminology changes

We've updated the terms we use to talk about PE reference architecture types, and are no longer using the term "monolithic."

Configuration	Nodes	Former name	New name
All infrastructure components on the master.	Up to 4,000	Monolithic	Standard
All infrastructure components on the master, plus one or more compilers and a load balancer.	4,000 to 20,000	Monolithic with compilers	Large
All infrastructure components on the master, plus one or more compilers and a load balancer, plus a separate node which hosts the PE-PostgreSQL instance.	More than 20,000	Monolithic with compilers and standalone PE-PostgreSQL	Extra large

Please note that a Standard Support Plan, one of the two levels of commercial support available from Puppet, is not restricted to PE instances using a standard reference architecture. "Standard" in this case refers to the support level, not the size of the PE installation.

Platform support

This version adds support for these platforms.

Master

- FIPS 140-2 compliant Enterprise Linux 7

Agent

- Debian Buster (10)

Deprecations and removals

Split and large environment installations

Split and large environment installations, where the master, console, and PuppetDB were installed on separate nodes, are no longer supported, and you can't upgrade to this version with a split install. For instructions on migrating from a split installation to a standard (formerly called monolithic) installation, see [Migrate from a split to a standard installation](#).

Web installation

The web-based method for installing infrastructure components has been removed. For a simplified installation, use the [express install](#) method instead.

Console admin password reset script

The Ruby script for resetting the console administrator password has been removed in favor of the new `puppet infrastructure console_password` command.

Use of `pe_ini_subsettings` to manage Java arguments

Java arguments on all Trapperkeeper services are now fully managed by the `puppet_enterprise` module, and use of `pe_ini_subsettings` to manage Java arguments is no longer supported. If you add Java arguments to the `java_args` environment variable using `pe_ini_subsettings`, the arguments will be removed during the next Puppet run.

Deprecated platforms

These platforms are deprecated in this release and will not be supported in the next major version of PE.

Master

- Enterprise Linux 6
- Ubuntu 16.04

Platforms reaching end of support

These platforms have reached end-of-life and are no longer supported.

Agent

- Ubuntu 14.04

Resolved issues

New Tasks fails to show status in the console

Occasionally when running a new task, the target node did not appear on the **Jobs** page with a status until you refreshed the console. This was the result of a time sync issue between the node event and the local orchestrator time, and has been fixed.

Package versions were not reset correctly after failed upgrade

If an agent run was in progress as an upgrade began, and consequently the installer failed because it could not acquire an agent lock, the installer did not roll back the relevant packages to the pre-upgrade PE version.

Replicas tried to query PuppetDB on the primary master

In high availability installations, the replica was incorrectly configured to first send queries to the PuppetDB service on the primary master. The failover list has been corrected so that the replica now queries its own PuppetDB service first.

Rerunning the installer created the All Environments node group

If the installer was run for a second time due an issue such as a failed upgrade or a faulty agent lock, the All Environments node group was mistakenly created for the installation. This issue has been resolved, and the All Environments node group is only created for new installations.

`puppet infrastructure run` plans caused Puppet agent service settings to be ignored

In some cases, plans used in `puppet infrastructure run` commands forced the Puppet agent service to run after the plan was complete, even if you had previously disabled the service. The impacted plans now reset the Puppet agent service to the state it was in before the plan was run.

Certificate backup directories could be overwritten

When a certificate regeneration command was run multiple times, certificate backup directories could be unintentionally overwritten. To solve this issue, certificate backup directories are now uniquely named using a time stamp.

Setting node group environment required `Edit configuration data` permission

To allow a user role to set a node group environment, users had to add the permission `Edit configuration data` in addition to `Set environment`. Now, the permission `Set environment` alone is enough to allow a user to change the environment.

Upgrade attempts failed when a Puppet run was in progress

If a Puppet run began while the installer was attempting to upgrade PE, conflicts and failures occurred. The installer now checks for Puppet runs before beginning an upgrade, and stops the upgrade if one is in progress.

Analytics can report on modules without metadata

Previously, modules without a `metadata.json` file installed on the Master would break analytical submissions and leave a schema error in the log. Now, analytics will report correctly on installed modules without metadata.

Run a task on agentless node using `ed25519` key

You can now use the `ed25519` SSH keys to run tasks against agentless SSH nodes.

PE known issues

These are the known issues in PE 2019.2.

Installation and upgrade known issues

These are the known issues for installation and upgrade in this release.

Missing package dependencies for SUSE Linux Enterprise Server agent nodes

On agent nodes running SUSE Linux Enterprise Server 15, the `libyaml-cpp` package and operating system packages prefixed with `libboost_` are no longer bundled with Puppet agent, and might not be included in the operating system. See [SUSE Linux Enterprise Server dependencies](#) for the full list of required packages. The `libyaml-cpp` package is in the Desktop Applications SUSE Linux Enterprise Server packaging module, and `libboost` packages are in the Basesystem module. If you encounter issues when installing Puppet agent on SUSE Linux Enterprise Server 15 nodes, use `zypper package-search` or the SUSE Linux Enterprise Server Customer Center packages list at <https://scc.suse.com/packages> to locate the packages and instructions on how to install them manually.

`puppet infrastructure run` commands can fail if the agent is run with cron

`puppet infrastructure run` commands, such as those used for certain installation, upgrade, and certificate management tasks, can fail if the Puppet agent is run with cron. The failure occurs if the command conflicts with a Puppet run. As a workaround, you can either disable the agent running out of cron on the nodes the plan affects, or be careful that cron doesn't execute an agent run while the command is running.

Installer can fail due to SSL errors with AmazonAWS

In some cases when attempting to install PE, some master platforms have received SSL errors when attempting to connect to `s3.amazonaws.com`, and thus have been unable retrieve puppet-agent packages needed for installation. In most cases, you should be able to properly install after updating the CA cert bundle on the master platform. To update the bundle, run the following commands:

```
rm /etc/ssl/certs/ca-bundle.crt
yum reinstall ca-certificates
```

After updating the CA bundle, run the PE installer again.

Upgrades can fail when using custom database certificate parameters

When upgrading an infrastructure with an unmanaged PostgreSQL database to this version of PE, the upgrade can fail and result in downtime if the databases use custom certificates not issued by the PE certificate authority. This is the case even if you configured the `database_properties` parameters in `pe.conf` to set a custom `sslrootcert`. The upgrader ignores these custom certificate settings.

To manually override the upgrader's certificate settings, run these commands, replacing `<NEW CERT>` with the name of your custom certificate.

```
rpm -iUv /opt/puppetlabs/server/data/packages/public/2019.3.0/el-7-
x86_64-6.12.0/pe-modules-2019.3.0.2-1.el7.x86_64.rpm

sed -i 's#/etc/puppetlabs/puppet/ssl/certs/ca.pem#/etc/puppetlabs/puppet/
ssl/<NEW CERT>.pem#'
/opt/puppetlabs/server/data/environments/enterprise/modules/pe_manager/lib/
puppet_x/puppetlabs/meep/configure/postgres.rb

./puppet-enterprise-installer - --force
```

Note: The first step in the code block above includes paths specific to the PE version you're upgrading to. Replace these version information as needed.

Supported platforms known issues

These are the known issues with supported platforms in this release.

Errors when using `puppet code` and `puppet db` commands on FIPS-enabled hardware

When the `pe-client-tools` packages are run on FIPS-enabled hardware, `puppet code` and `puppet db` commands fail with SSL handshake errors. To use `puppet db` commands on a FIPS-enabled machine, install the [puppetdb_cli](#) Ruby gem with the following command:

```
/opt/puppetlabs/puppet/bin/gem install puppetdb_cli --bindir /opt/puppetlabs/bin/
```

To use `puppet code` commands on a FIPS-enabled machine, use the Code Manager API. Alternately, you can use `pe-client-tools` on a non-FIPS-enabled machine to access a FIPS-enabled master.

FIPS installations do not support Razor

The FIPS platform does not support the Razor component in PE.

Configuration and maintenance known issues

These are the known issues for configuration and maintenance in this release.

`puppet infrastructure recover_configuration` misreports success if specified environment doesn't exist

If you specify an invalid environment when running `puppet infrastructure recover_configuration`, the system erroneously reports that the environment's configuration was saved.

Restoring the `pe-rbac` database fails with the `puppet-backup restore` command

When restoring the `pe-rbac` database, the restore process exits with errors about a duplicate operator family, `citext_ops`.

To work around this issue:

1. Log into your existing PostgreSQL instance:

```
sudo su - pe-postgres -s /bin/bash -c "/opt/puppetlabs/server/bin/psql pe-rbac"
```

2. Issue these commands:

```
ALTER EXTENSION citext ADD operator family citext_ops using btree;
ALTER EXTENSION citext ADD operator family citext_ops using hash
```

3. Exit the PostgreSQL shell and re-run the backup utility.

`puppet-backup` fails if gems are missing from the master's agent environment

The `puppet-backup create` command might fail if any gem installed on the Puppet Server isn't present on the agent environment on the master. If the gem is either absent or of a different version on the master's agent environment, you get the error "command `puppet infrastructure recover_configuration` failed".

To fix this, you must install any missing or incorrectly versioned gems on the master's agent environment. To find which gems are causing the error, check the backup logs for any gem incompatibility issues with the error message. PE creates backup logs as a `report.txt` whenever you run a `puppet-backup` command.

To see what gems and their versions you have installed on your Puppet Server, run the command `puppetserver gem list`. To see what gems are installed in the agent environment on your master, run `/opt/puppetlabs/puppet/bin/gem list`.

Console and console services known issues

These are the known issues for the console and console services in this release.

Console is inaccessible on macOS Catalina using default certificates

Enhanced security requirements in macOS Catalina prevent accessing the console using the default certificate generated during installation. As a workaround, you can [specify a custom SSL certificate](#), editing parameters of the `puppet_enterprise::profile::console` class in Hiera instead of in the console. For example, in `/etc/puppetlabs/code/environments/production/data/common.yaml` add:

```
puppet_enterprise::profile::console::browser_ssl_cert: /opt/puppetlabs/
server/data/console-services/certs/public-console.cert.pem
puppet_enterprise::profile::console::browser_ssl_private_key: /opt/
puppetlabs/server/data/console-services/certs/public-console.private_key.pem
```

For more details about certificate guidelines in macOS Catalina, see the Apple support article about [requirements for trusted certificates in macOS 10.15](#).

Mismatch between classifier classification and matching nodes for regexp rules

PuppetDB's regular expression matching has surprising behaviors for structured fact value comparisons. For example, the structured fact `os` is a rule that matches `["~", "os", ":"]`. PuppetDB would unintentionally match every node that has the `os` structured fact because the regular expression is applied to the JSON encoded version of the fact value.

The classifier does not use PuppetDB for determining classification and regular expressions in the classifier rules syntax only support direct value comparisons for string types.

This has caused issues in the console where the node list and counts for the "matching nodes" display sometimes indicated that nodes were matching even though the classifier would not consider them matching.

After the fix, the same criteria will be applied to the displays and counts that the classifier uses. The output of the classifier's rule translation endpoints will also make queries that match the classifier behavior.

Note: This fix does not change the way nodes are classified, it only corrects how the GUI displays matching nodes.

Orchestration services known issues

These are the known issues for the orchestration services in this release.

Orchestrator fails when rerunning tasks on agentless nodes

When you rerun tasks from the Job details page, target nodes that do not have the Puppet agent installed are miscategorized as PuppetDB nodes. This causes the orchestrator to fail on those nodes.

Target name `localhost` unavailable for plans in PE

The special target name `localhost` is not available in plans for PE since the definition is ambiguous.

Some Bolt stdlib functions fail in PE

The following Bolt stdlib functions will return an error if they are run in PE:

- `apply_prep`
- `file::read`
- `file::write`

- `file::readable`
- `file::exists`
- `set_features`
- `add_to_group`

For more information, see [Plan caveats and limitations](#) on page 464

No metadata in `puppet plan show <PLAN NAME>` function

There is a known issue where the `puppet plan show <PLAN NAME>` function does not return plan metadata. To see plan metadata, view the plan source code.

The function `puppet plan show` displays invalid plans when Code Manager is disabled

If Code Manager is disabled, the `puppet plan show` function and the PE console will still list plans that are contained in a control repo, but those plans cannot actually be run. Enable Code Manager to run plans.

SSL and certificate known issues

These are the known issues for SSL and certificates in this release.

Regenerating agent certificates fails with autosign enabled

The `puppet infrastructure run regenerate_agent_certificate` command includes a step for signing the node's certificate. With autosign enabled, an unsigned certificate can't be found, and the command errors out. As a workaround, temporarily disable autosign before running `puppet infrastructure run regenerate_agent_certificate`.

Code management known issues

These are the known issues for Code Manager, r10k, and file sync in this release.

Default SSH URL with TFS fails with rugged error

Using the default SSH URL with Microsoft Team Foundation Server (TFS) with the rugged provider causes an error of "unable to determine current branches for Git source." This is because the rugged provider expects an @ symbol in the URL format.

To work around this error, replace `ssh://` in the default URL with `git@`

For example, change:

```
ssh://tfs.puppet.com:22/tfs/DefaultCollection/Puppet/_git/control-repo
```

to

```
git@tfs.puppet.com:22/tfs/DefaultCollection/Puppet/_git/control-repo
```

GitHub security updates might cause errors with `shellgit`

GitHub has disabled TLSv1, TLSv1.1 and some SSH cipher suites, which can cause automation using older crypto libraries to start failing. If you are using Code Manager or r10k with the `shellgit` provider enabled, you might see negotiation errors on some platforms when fetching modules from the Forge. To resolve these errors, switch your configuration to use the rugged provider, or fix `shellgit` by updating your OS package.

Timeouts when using `--wait` with large deployments or geographically dispersed compilers

Because the `--wait` flag now deploys code to all compilers before returning results, some deployments with a large node count or compilers spread across a large geographic area might experience a timeout. Work around this issue by adjusting the `timeouts_sync` parameter.

r10k with the Rugged provider can develop a bloated cache

If you use the Rugged provider for r10k, repository pruning is not supported. As a result, if you use many short-lived branches, over time the local r10k cache can become bloated and take up significant disk space.

If you encounter this issue, run `git-gc` periodically on any cached repo that is using a large amount of disk space in the `cachedir`. Alternately, use the `shellgit` provider, which automatically garbage collects the repos according to the normal Git CLI rules.

Code Manager and r10k do not identify the default branch for module repositories

When you use Code Manager or r10k to deploy modules from a Git source, the default branch of the source repository is always assumed to be `master`. If the module repository uses a default branch that is *not* `master`, an error occurs. To work around this issue, specify the default branch with the `ref:` key in your Puppetfile.

After an error during the initial run of file sync, Puppet Server won't start

The first time you run Code Manager and file sync on a master, an error can occur that prevents Puppet Server from starting. To work around this issue:

1. Stop the `pe-puppetserver` service.
2. Locate the `data-dir` variable in `/etc/puppetlabs/puppetserver/conf.d/file-sync.conf`.
3. Remove the directory.
4. Start the `pe-puppetserver` service.

Repeat these steps on each master exhibiting the same symptoms, including any compilers.

Puppet Server crashes if file sync can't write to the live code directory

If the live code directory contains content that file sync didn't expect to find there (for example, someone has made changes directly to the live code directory), Puppet Server crashes.

The following error appears in `puppetserver.log`:

```
2016-05-05 11:57:06,042 ERROR [clojure-agent-send-off-pool-0] [p.e.s.f.file-
sync-client-core] Fatal error during file sync, requesting shutdown.
org.eclipse.jgit.api.errors.JGitInternalException: Could not delete file /
etc/puppetlabs/code/environments/development
    at org.eclipse.jgit.api.CleanCommand.call(CleanCommand.java:138)
    ~[puppet-server-release.jar:na]
```

To recover from this error:

1. Delete the environments in code dir: `find /etc/puppetlabs/code -mindepth 1 -delete`.
2. Start the `pe-puppetserver` service: `puppet resource service pe-puppetserver ensure=running`
3. Trigger a Code Manager run by your usual method.

Code manager cannot deploy forge modules with a proxy (2019.2.0 and 2019.2.1)

The commands `puppet code deploy` and `r10k` fail when behind a proxy. The commands do not use the configured proxy settings and using them can result in issues downloading modules from the Puppet forge. This is due to an issue in a dependency gem.

To work around this issue, downgrade the gem to the previous version.

```
/opt/puppetlabs/puppet/bin/gem install faraday -v 0.12.2
/opt/puppetlabs/puppet/bin/gem uninstall faraday -v 0.13.1
puppet resource service pe-puppetserver ensure=stopped
puppet resource service pe-puppetserver ensure=running
```


Internationalization known issues

These are the known issues for internationalization and UTF-8 support in this release.

ASCII limitations

Certain elements of Puppet and PE are limited to ASCII characters only, or work best with ASCII characters, including:

- Names for environments, variables, classes, resource types, modules, parameters, and tags in the Puppet language.
- File names, which can generate errors when referenced as a `source` in a file resource or concat fragment.
- The `title` and `namevar` for certain resource types, on certain operating systems. For example, the user and group resources on RHEL and CentOS might contain only ASCII characters in `title` and `namevar`.
- The console password.

Ruby can corrupt the `path` fact and environment variable on Windows

There is a bug in Ruby that can corrupt the environment variable names and values. This bug causes corruption for only some codepages. This bug might be triggered when environment names or values contain UTF-8 characters that can't be translated to the current codepage.

The same bug can cause the `path` fact to be cached in a corrupt state.

Getting started with Puppet Enterprise on *nix

Puppet Enterprise (PE) is automation software that can help you and your organization be productive and agile while managing your IT infrastructure.

What is Puppet Enterprise (PE)?

Puppet Enterprise (PE) blends together Puppet, Bolt, and a robust selection of tools to help you automatically and continuously manage your large infrastructure.

PE is a commercial version of Puppet, our original open source product often used by individuals managing smaller infrastructures. It has all the best parts of Puppet, plus other features like a graphical user interface, orchestration services, role-based access control, reporting, and the capability to manage thousands of nodes at once. PE also incorporates other tools and products within Puppet to create comprehensive configuration management software.

Here is a little more information about what goes into PE.

The Puppet language for maintaining a solid infrastructure

Puppet is a declarative language. This means that instead of you actively managing your configurations, you tell Puppet what your desired infrastructure looks like and it works to keep it that way. This is the opposite of an imperative language, like Java, which focuses on what things are required to get to an end result.

Bolt for when you need it done now

Bolt is another Puppet product that helps users do one-off tasks without disrupting anything in their configurations. PE allows you to run Bolt tasks and plans from the console or on the command line to quickly do things like restart services or inspect a fact on your system.

CD4PE and the PDK for helping you manage code

PE works best in conjunction with two of Puppet's coding tools - [Continuous Delivery for PE \(CD4PE\)](#) and the [Puppet Development Kit \(PDK\)](#). CD4PE is a tool that help you streamline testing and deployment of Puppet code and the PDK is a tool for building quality modules.

Together, they make your Puppet code easier to manage.

What can PE do?

You know a little about what PE is and what parts of Puppet went into creating it, but you still might be wondering what problems it can help you solve.

Here are just a few examples of things PE can do for you.

- Automatically synchronize the clocks on all of your servers to reduce failures and errors.
- Deploy IT permissions changes to entire job roles or departments.
- Alert you of any unauthorized changes to your infrastructure.
- Quickly make profile changes to all user accounts at once.
- Allow you to view and filter information about nodes.
- Schedule reports.

Check out our [video](#) for more information.

Key components of PE

Here are notes about some the key components within PE that will help you along the way.

The Puppet master

The Puppet master, or master, is the central hub of activity and process in Puppet Enterprise. This is where PE compiles code to create agent catalogs and where SSL certificates are verified and signed.

Puppet Server

Puppet Server is an application that runs on the Java Virtual Machine (JVM) on the master. In addition to hosting endpoints for the certificate authority service, it also powers the catalog compiler.

Catalog

To configure a managed node, the agent uses a document called a catalog. The catalog describes the desired state for each resource that should be managed on the node. A resource is a configurable unit of information, like a user or service.

Agents

Agents are responsible for ensuring your configuration stays in the desired state by periodically checking it against the catalog. If there are changes, the agent sends a report to the master or compiler and changes it back to the desired state based on what the catalog says.

Role-based access control (RBAC)

RBAC is the way you manage user permissions in PE. With RBAC, you define a role within your infrastructure, configure the permissions within the role, and assign users who need those permissions to that role. This way, you do not have to manage individual user permissions.

Node classifier (NC)

PE comes with its own node classifier, which is built into the console. Classification is when you configure your managed nodes by assigning classes to them. **Classes** provide the Puppet code—distributed in modules—that enables you to define the function of a managed node, or apply specific settings and values to it.

Modules

Modules are self-contained, shareable bundle of code and data. Each module manages a specific task in your infrastructure, like installing and configuring a piece of software. Most Puppet code is written within modules. You can write your own modules or download pre-built modules from the Forge.

Forge

The [Forge](#) is a Puppet-run repository for storing, sharing, and installing modules. Some modules are approved by Puppet, meaning they have been tested rigorously by Puppet and Puppet maintains them. Other modules are built and submitted by users or organizations.

Puppet Development Kit (PDK)

The [PDK](#) is downloadable tool for building modules. It provides a command line interface and integrated testing features to help you develop high-quality code. We highly recommend checking out the PDK when you begin writing your own code.

Install and administer Puppet Enterprise

Whether you're setting up a PE installation for actual deployment or want to learn some fundamentals of configuration management with PE, we'll provide you with the steps you need to get up and running relatively quickly.

You will go through the setup of a standard installation and learn how to perform some common PE tasks to configure important parts of your infrastructure, like synchronizing your clocks or setting up your DNS nameserver.

The steps and concepts are organized in the order that you would most likely perform or encounter them.

Start installing PE on *nix

The express install of PE is ideal for trying out PE with up to 10 nodes and can be used to manage up to 4,000 nodes.

Here are some general guidelines for what is required depending on your system capacity to download and install PE.

On a single *nix machine, you install:

- The master, the central hub of activity, where Puppet code is compiled to create agent catalogs, and where SSL certificates are verified and signed.
- The PE console, the graphical web interface, which features numerous configuration and reporting tools.
- PuppetDB, which collects data generated throughout your Puppet infrastructure.

Download and verify the installation package

PE is distributed in downloadable packages specific to supported operating system versions and architectures. Installation packages include the full installation tarball and a GPG signature (.asc) file used to verify authenticity.

Before you begin

GnuPG is an open source program that allows you to safely encrypt and sign digital communications. You must have GnuPG installed to sign for the release key. Visit the [GnuPG website](#) for more information or to download GnuPG.

1. [Download](#) the tarball appropriate to your operating system and architecture.
2. Import the Puppet public key.

```
wget -O - https://downloads.puppetlabs.com/puppet-gpg-signing-key.pub |
gpg --import
```

3. Print the fingerprint of the key.

```
gpg --fingerprint 0x7F438280EF8D349F
```

The primary key fingerprint displays: 6F6B 1550 9CF8 E59E 6E46 9F32 7F43 8280 EF8D 349F.

4. Verify the release signature of the installation package.

```
$ gpg --verify puppet-enterprise-<version>-<platform>.tar.gz.asc
```

The result is similar to:

```
gpg: Signature made Tue 18 Sep 2016 10:05:25 AM PDT using RSA key ID
EF8D349F
gpg: Good signature from "Puppet, Inc. Release Key (Puppet, Inc. Release
Key)"
```

Note: If you don't have a trusted path to one of the signatures on the release key, you receive a warning that a valid path to the key couldn't be found.

Install using express install

Express installation uses default settings to install PE, so you don't have to edit a `pe.conf` file before or during installation. At the end of the installation process, you're prompted to provide a console administrator password, which is the only user-required value.

Log in as a root user to perform these steps.

1. Unpack the installation tarball:

```
tar -xf <TARBALL_FILENAME>
```

2. From the installer directory, run the installer:

```
sudo ./puppet-enterprise-installer
```

3. When prompted, select express install mode.
4. After installation completes, follow the prompts to specify a password, or run: `puppet infrastructure console_password --password=<MY_PASSWORD>`
5. Run Puppet twice: `puppet agent -t`.

You must restart the shell before you can use PE client tool subcommands.

Log into the PE console

The console is a graphical interface that allows you to manage parts your PE infrastructure without relying on the command line.

Log in for the first time.

1. When you navigate to the hostname of the master in your browser, you'll receive a browser warning about an untrusted certificate. This is because you were the signing authority for the console's certificate, and your Puppet Enterprise deployment is not known to your browser as a valid signing authority. Ignore the warning and accept the certificate.
2. On the login page for the console, log in with the username `admin` and the password you created when installing the Puppet master. Keep track of this login as you will use it later.

Congratulations, you've successfully installed the Puppet master node! Next, learn how to install an agent on a node using the console.

Start installing *nix agents

A common way to manage nodes in PE is to install an agent on them.

In this guide you install a *nixPuppet agent and how to add agentless nodes to your inventory.

These instructions assume you've installed PE.

Install agents from the console

You can use the console to leverage tasks that install *nix or Windows agents on target nodes.

Before you begin

Add classes to the **PE Master** node group for each agent platform used in your environment. For example, `pe_repo::platform::el_7_x86_64`.

Make sure you have permission to run the appropriate task to install agents on all nodes:

- *nix targets use the task `pe_bootstrap::linux`
- Windows targets use the task `pe_bootstrap::windows`

For Windows targets, this task requires:

- Windows 2008 SP2 or newer
- PowerShell version 3 or higher
- Microsoft .NET Framework 4.5 or higher

1. In the console, click **Inventory**.
2. Select the node type **Nodes with Puppet agents**.
3. Select a transport method.
 - **SSH** for *nix targets
 - **WinRM** for Windows targets
4. Enter the target host names and the credentials required to access them. If you use an SSH key, include begin and end tags.
5. Click **Add nodes**.

Tip: Click **Installation job started** to view the job details for the task.

Agents are installed on the target nodes and then automatically submit certificate signing requests (CSR) to the master. The list of unsigned certificates is updated with new targets.

Related information

[Managing certificate signing requests in the console](#) on page 54

Open certificate signing requests appear in the console on the **Unsigned certs** page. Accept or reject submitted requests individually or in a batch.

[Managing certificate signing requests on the command line](#) on page 54

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

Add agentless nodes to the inventory

Add nodes that don't have the Puppet agent installed to the inventory so you can run tasks on them.

Before you begin

Add classes to the **PE Master** node group for each agent platform used in your environment. For example, `pe_repo::platform::el_7_x86_64`.

Make sure you have the permission **Nodes: Add and delete connection information from inventory service**.

1. In the console, click **Inventory**.
2. Select the node type **Nodes without Puppet agents**.

3. Select a transport method.
 - **SSH** for *nix targets
 - **WinRM** for Windows targets
4. Enter target host names and the credentials required to access them. If you use an SSH key, include begin and end tags.
5. Optional: Select additional target options. For example, to add a target port number, select **Target Port** from the drop-down list, enter the number, and click **Add**. For details, see [Transport configuration options](#) on page 317, below.
6. Click **Add nodes**.

After the nodes have been added to the inventory, they are added to PuppetDB, and you can view them from the **Nodes** page. Nodes in the inventory can be added to an inventory node list when you set up a job to run tasks. To review each inventory node's connection options, or to remove the node from inventory, go to the **Connections** tab on the node's details page.

Managing certificate signing requests

When you install a Puppet agent on a node, the agent automatically submits a certificate signing request (CSR) to the master. You must accept this request to bring before the node under PE management can be added your deployment. This allows Puppet to run on the node and enforce your configuration, which in turn adds node information to PuppetDB and makes the node available throughout the console.

You can approve certificate requests from the PE console or the command line. If DNS altnames are set up for agent nodes, you must approve the CSRs on use the command line interface .

Note: Specific user permissions are required to manage certificate requests:

- To accept or reject CSRs in the console or on the command line, you need the permission **Certificate requests: Accept and reject**.
- To manage certificate requests in the console, you also need the permission **Console: View**.

Related information

[Installing agents](#) on page 147

You can install Puppet Enterprise agents on *nix, Windows, and macOS.

Managing certificate signing requests in the console

Open certificate signing requests appear in the console on the **Unsigned certs** page. Accept or reject submitted requests individually or in a batch.

- To manage requests individually, click **Accept** or **Reject**.
- To manage the entire list of requests, click **Accept All** or **Reject All**. Nodes are processed in batches. If you close the browser window or navigate to another website while processing is in progress, only the current batch is processed.

The node appears in the PE console after the next Puppet run. To make a node available immediately after you approve the request, run Puppet on demand.

Related information

[Running Puppet on demand](#) on page 434

The orchestrator gives you the ability to set up jobs in the console or on the command line to trigger on-demand Puppet runs.

Managing certificate signing requests on the command line

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

To view pending node requests on the command line:

```
$ sudo puppetserver ca list
```

To sign a pending request:

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

To sign pending requests for nodes with DNS altnames:

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

Manage your infrastructure

Learn how to use PE to configure and manage essential parts of your infrastructure, such as NTP, DNS, and Apache.

Restart a service using the console

The console is the graphical interface for Puppet Enterprise (PE).

Use the console to:

- Manage node requests to join the Puppet deployment.
- Assign Puppet classes to nodes and groups.
- Run Puppet on specific groups of nodes.
- View reports and activity graphs.
- Run tasks and plans.
- Browse and compare resources on your nodes.
- View package and inventory data.
- Manage console users and their access privileges.

You will learn how to accept the console certificate, log in, and run a task to restart a service.

There will be many more console functionalities covered later, which include things like managing permissions and creating node groups. These are taught in tandem with some system configuration tasks you might need to complete to get up and running.

Once you configure the basics and have added more nodes, roles, and users, we encourage you to revisit the console to get a deeper understanding of some of its core features.

Reaching the console

The console is served as a website over SSL, on whichever port you chose when installing the console component.

Let's say your console server is `console.domain.com`. If you chose to use the default port (443), you can omit the port from the URL and reach the console by navigating to `https://console.domain.com`.

If you chose to use port 8443, you reach the console at `https://console.domain.com:8443`.

Remember: Always use the `https` protocol handler. You cannot reach the console over plain `http`.

Accepting the console's certificate

The console uses an SSL certificate created by your own local Puppet certificate authority. Because this authority is specific to your site, web browsers won't know it or trust it, and you must add a security exception in order to access the console.

Adding a security exception for the console is safe to do. Your web browser warns you that the console's identity hasn't been verified by one of the external authorities it knows of, but that doesn't mean it's untrustworthy. Because you or another administrator at your site is in full control of which certificates the Puppet certificate authority signs, the authority verifying the site is *you*.

When your browser warns you that the certificate authority is invalid or unknown:

- In Chrome, click **Advanced**, then **Proceed to <CONSOLE ADDRESS>**.
- In Firefox, click **Advanced**, then **Add exception**.

- In Internet Explorer or Microsoft Edge, click **Continue to this website (not recommended)**.
- In Safari, click **Continue**.

Logging in

Accessing the console requires a username and password.

If you are an administrator setting up the console or accessing it for the first time, use the username and password you chose when you installed the console. Otherwise, get credentials from your site's administrator.

Because the console is the main point of control for your infrastructure, it is a good idea to prohibit your browser from storing the login credentials.

Use a task to restart a service on one node

Puppet Enterprise allows you to run ad-hoc Bolt tasks using the console.

Before you begin

Make sure you have installed PE and logged into the console.

A task is a single action you can take on one or more targeted machines. It allows you to do things like upgrade packages and restart services without changing anything in your infrastructure. PE comes with a few tasks installed, such as `package`, `service`, and `puppet_conf`, but you can download more tasks from the Forge, which will be covered later.

In this section, you will learn how to access the tasks page in the console and run a job to execute the `service` task, which will instruct PE to restart the `nginx` service. It is important to be able to restart services when systems get hung up or when new applications are installed.

1. In the console, in the **Run** section, click **Task**.
2. In the **Task** field, select a task to run. For this example, use `service`.

Note: If the task you want to run is not available, you either do not have it installed or do not have permission to run it.

3. Optional: In the **Job description** field, provide a description. What you enter here appears on the job list and job details pages.
4. Set parameters and values for the task. Click **Add parameter** for each parameter-value pair you add.

For the `service` task, you have two required parameters. For **action**, you can choose `restart`. For **service**, enter `nginx`.

Note: Parameters marked with an asterisk (*) are required. Optional parameters are available in a dropdown. If there are no required or optional parameters, you can add arbitrary parameters provided those parameters are allowed in the task.

Express values as strings, arrays, objects, integers, or booleans (true or false). You must express empty strings as two double quotes with no space (" "). Wrap boolean values in double quotes (for example, "false").

5. Under **Schedule**, select **Now**.

You can also schedule a task by selecting **Later** and setting a chosen time, day, and time zone.

6. Under **Select targets**, choose **Node list** from the drop down.

a) In the **Inventory nodes** field, start typing the name of the node you want to target and click **Search** to find it.

7. Once you have selected your target node, click **Run job** to execute the task.

8. Navigate to the **Jobs** page to view the status of the job and any output it produces.

Congratulations! You have run a task using the PE console. To learn more about tasks, including how to install them from the Forge or how to write your own, visit the [Installing tasks](#) and [Writing tasks](#) section of the docs.

Next, install and manage the Apache server.

Manage Apache services

Install and manage your Apache server using a module from the Forge.

Apache overview

Apache, also called the HTTP, is common web server software responsible for bridging web server files with the actual content web users see on the internet, creating a readable website. Since Apache is crucial for a lot of websites to function, it is important to ensure it is installed and configured correctly.

Apache is the first of a several system configurations you will learn how to manage and will introduce you to several types of key administrative tasks you might perform frequently in PE.

You will:

- Install a module
- Create a node group
- Classify nodes
- Edit class parameters
- Assign permissions to users
- Write a module

Install the Apache module

The Puppet Forge contains thousands of modules that you can use in your own environment.

Before you begin

These instructions assume you've installed PE, at least one *nix agent node.

Modules are self-contained, shareable bundles of code and data. Each module manages a specific task in your infrastructure, like installing and configuring a piece of software. You can write your own modules or download pre-built modules from the Forge. While you can use any module available on the Forge, PE customers can take advantage of supported modules. These modules are designed to make common services easier, and are tested and maintained by Puppet. A lot of your infrastructure will be supported by modules, so it is important to learn how to install, build, and use them.

In this guide, you will install the pre-built `puppetlabs-apache`, which will help you install, configure, and manage Apache services.

1. Install the Apache module by running `puppet module install puppetlabs-apache`.

The results look like 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/code/environments/production/modules
### puppetlabs-apache (v1.1.1)
```

2. Visit the Puppet Forge to learn more about the [apache module](#).

You have just installed a Puppet module!

Next, use the `apache` class included in the `puppetlabs-apache` module to manage Apache with Puppet Enterprise.

About module directories

By default, Puppet keeps modules in `/etc/puppetlabs/code/environments/production/modules`. This includes modules that you download from the Forge and those you write yourself.

PE also creates two other module directories: `/opt/puppetlabs/puppet/modules` and `/etc/puppetlabs/staging-code/modules`. For this guide, don't modify or add anything to either of these directories.

There are some resources about modules and the creation of modules that you can reference.

- [Welcome to Puppet modules](#)
- [Module fundamentals](#)
- [The Forge](#)

Let a node manage Apache by applying a class

Classification is a way to configure nodes by creating node groups, adding nodes to the groups, and adding classes that specify what types of resources and code can be used by nodes in the group. Allow your agent node to manage Apache by adding the `apache` class to it.

Before you begin

These instructions assume you've installed PE, at least one *nix agent node, and the `puppetlabs-apache` module.

Every module contains one or more classes -- a named chunk of Puppet code. Classes are the primary means by which Puppet Enterprise configures nodes. The `puppetlabs-apache` module you installed in the last section contains a class called `apache`. In this section, you apply the `apache` class to your agent node

Step 1: Create a node group

Node groups enable you to assign classes to more than one node at a time.

You could assign classes to individual nodes one at a time, but chances are, each of your classes needs to be applied to more than one node. By creating a node group, you can apply a class to many nodes at one time.

For this section, we will create the node group `apache_example`.

1. In the console, click **Classification**, and click **Add group**.
2. Specify options for the new node group:
 - **Parent name** : select **All nodes**
 - **Group name** : Type `apache_example`
 - **Environment**: select **production**
 - **Environment group**: don't select this option
3. Click **Add**.

Step 2: Add nodes to the node group

Add nodes to a node group to manage them more efficiently.

To add nodes to a node group, create rules that define which nodes to include in the group.

1. Click the `apache_example` group you created on the **Classification** page.
2. In the **Rules** tab, in the **Certname** area, in the **Node name** field, enter the name of your PE agent node.
3. Click **Pin node**.
4. Commit changes.
5. Repeat these steps for any other nodes you want to add to the node group.

Step 3: Add the `apache` class to the node group

After you create a node group, add classes to give the matching nodes purpose.

1. Unless you have navigated elsewhere in the console, the `apache_example` node group is still displayed in the **Classification** page. On the **Configuration** tab, in the **Class name** field, begin typing `apache`, and select it from the autocomplete list.
2. Click **Add class**, and commit changes.

The `apache` class now appears in the list of classes for your agent. You can see this list by clicking **Inventory** and then clicking your node in the **Inventory** list. When the page opens with your node's details, click the **Configuration** tab.

3. Apply your changes. From the command line on your agent, run `puppet agent -t`.

4. To see your changes in action, navigate to `/var/www/html/`, and create a file named `index.html`.
5. Open `index.html` with the text editor of your choice and add some content (for example, "Hello, World!").
6. From the command line of your agent node, run `puppet agent -t`. This configures the node using the newly assigned class.
7. Wait one or two minutes.
8. Open a browser and enter the IP address for the agent node, adding port 80 on the end, as in `http://myagentnodeIP:80/`. The contents of `/var/www/html/index.html` are displayed.

Step 4: Edit class parameters in the console

You can use the console to modify the values of a class's parameters without editing the module code directly.

1. In the console, click **Classification**, and then find and select the `apache_example`.
2. On the **Configuration** tab, find `apache` in the list of classes.
3. From the **Parameter name** drop-down list, choose the parameter you want to edit. For this example, select `docroot`.

Note: The grey text that appears as values for some parameters is the default value, which can be either a literal value or a Puppet variable. You can restore this value by selecting **Discard changes** after you have added the parameter.

4. In the **Value** field, enter `/var/www`.
5. Click **Add parameter**, and commit changes.
6. From the command line of your PE-managed node, run `puppet agent -t`.

This triggers a Puppet run, and Puppet Enterprise creates the new configuration.

You have set the Apache web server's root directory to `/var/www` instead of its default `/var/www/html`. If you refresh `http://myagentnodeIP:80/` in your web browser, it shows the list of files in `/var/www`. If you click `html`, the browser again shows the contents of `/var/www/html/index.html`.

You have learned how to create and classify node groups and edit class parameters while instructing Puppet Enterprise to manage the default Apache vhost on your agent node. Next, learn how to manage users with PE, and how to set permissions for each user or group of users.

Determine who can access Apache configurations

Role-based access control (RBAC) is used to manage user permissions. Permissions define what actions users can perform on designated objects. There are multiple steps involved in an RBAC workflow, which can be adapted to fit your needs.

Before you begin

These instructions assume you've installed PE, at least one *nix agent node, and the `puppetlabs-apache` module.

In this section, you will learn how to create a role, give the role access to the Apache node group, and add users to the role.

Note: Roles are deletable by API, not in the console. Therefore, it's best to try out these steps on a virtual machine.

Step 1: Create a user role

User roles are sets of permissions you can apply to multiple users. You can't assign permissions to single users in PE, only to user roles. This allows you to have more consistent permissions among users and be able to make changes efficiently.

1. In the console, from the **Access control** drop down, click **User roles**.
2. For **Name**, type `Web developers`, and then for **Description**, type a description for the Web developers role, such as `Members of the web development team`.
3. Click **Add role**.

Step 2: Give the user role access to view a node group

Users get permissions when the user roles they belong to are granted those permissions.

One of the permissions you can grant to user roles is the ability to access (view, create, or edit) node groups.

1. From the **User Roles** page, select `Web developers`, and then click the **Permissions tab**.
2. In the **Type** box, select **Node groups**.
3. In the **Permission** box, select **View**.
4. In the **Object** box, select `apache_example`.
5. Click **Add permission**, and then click the **commit** button.

You have given members of the `Web developers` role permission to view the `apache_example` node group.

Step 3: Create a new user

You can add local users, or import users and groups from a directory.

These steps add a local user. You can also import users and groups from an external directory, so you don't have to recreate users one at a time.

1. In the console, click **Users**.
2. In the **Full name** field, type in a user name.
3. In the **Login** field, type the user's login information.
4. Click **Add local user**.

Related information

[Connecting external directory services to PE](#) on page 237

Puppet Enterprise connects to external Lightweight Directory Access Protocol (LDAP) directory services through its role-based access control (RBAC) service. Because PE integrates with cloud LDAP service providers such as Okta, you can use existing users and user groups that have been set up in your external directory service.

Step 4: Enable the new user to log in

When you create new local users, you need to send them a password reset token so that they can log in for the first time.

1. Click the new local user in the **Users** list.
2. On the upper-right of the page, click **Generate password reset**. A **Password reset link** message box opens.
3. Copy the link provided in the message and send it to the new user. Then you can close the message.

Step 5: Assign the user role to the new user

Users must be assigned to one or more roles before they can log in and use PE.

When you add users to a role, the user gains the permissions that are applied to that role.

1. Click **User Roles** and then click **Web developers**.
2. On the **Member users** tab, on the **User name** list, select the new user you created, and then click **Add user** and click the **commit** button.

You're now managing a user with RBAC. By using permissions and user roles, you give the appropriate level of access and agency to each user or user group who works with PE.

Next, learn more about the basics of writing modules of your own, so you can begin customizing your deployment and getting work done.

Write and modify modules for Apache

Modify existing modules in the Forge to better meet your needs. You can also write your own modules, using the Puppet Development Kit to create and unit test it. Work through some examples of modifying and writing modules using your Apache configurations.

Modules are reusable chunks of Puppet code and are the basic building blocks of any Puppet Enterprise (PE) deployment. Some modules from the Forge might precisely fit your needs, but many are *almost* --- but not quite ---

what you need. You might want to adapt pre-written modules to suit your deployment's requirements. In other cases, you might need to write your own modules from scratch.



CAUTION: The directories and some of the commands in this guide do not work correctly with `.`. If you are using `.`, do not follow the steps in this guide.

Module location

By default, Puppet keeps modules in `/etc/puppetlabs/code/environments/production/modules`. This includes modules installed by PE, those that you download from the Forge, and those you write yourself. You can configure this path with the `modulepath` setting in `puppet.conf`.

Note: PE also creates another module directory at `/opt/puppetlabs/puppet/modules`. Don't modify anything in this directory or add modules of your own to it.

Module structure

Modules are directory trees that contain manifests, templates, and other files.

These files are in the `puppetlabs-apache` module:

- `apache/` (the module name)
 - `manifests/`
 - `init.pp` (contains the `apache` class)
 - `php.pp` (contains the `php` class to install PHP for Apache)
 - `vhost.pp` (contains the Apache virtual hosts class)
 - `templates/`
 - `vhost/`
 - `_file_header.erb` (contains the `vhost` template, managed by PE)

Every manifest (`.pp`) file contains a single class. File names map to class names in a predictable way: `init.pp` contains a class with the same name as the module, in this case `apache`. `<NAME>.pp` contains a class called `<MODULE NAME>::<NAME>`. `<NAME>/<OTHER NAME>.pp` contains `<MODULE NAME>::<NAME>::<OTHER NAME>`.

Many modules, including Apache, contain directories other than `manifests` and `templates`. For simplicity's sake, we do not cover them in this introductory guide.

Edit a module's manifest to populate facts

Modules from the Forge provide a great basis for common tasks. You can configure them to meet your specific needs.

Before you begin

These instructions assume you've installed PE, at least one *nix agent node, and the `puppetlabs-apache` module.

In this exercise, you modify a template from the `puppetlabs-apache` module, specifically `vhost.conf.erb`, to include some simple variables automatically populated with facts about your node.

Note: Log in as root or administrator on your nodes.

1. On the Puppet master, navigate to the modules directory by running `cd /etc/puppetlabs/code/environments/production/modules`
2. Run `ls` to view the currently installed modules and note that `apache` is present.
3. Open `apache/templates/vhost/_file_header.erb` in a text editor. (Avoid using Notepad because it can introduce errors.) The `_file_header.erb` file contains the following header:

```
# *****
# Vhost template in module puppetlabs-apache
# Managed by Puppet
# *****
```

4. Use the PE lookup tool, Factor, to collect the following facts about your agent node:
 - run `factor operatingsystem` (this returns your agent node's OS)
 - run `factor id` (this returns the id of the currently logged in user)
5. Edit the header of `_file_header.erb` so that it contains the following variables for Factor lookups:

```
# *****
# Vhost template in module puppetlabs-apache
# Managed by Puppet
#
# This file is authorized for deployment by <%= scope.lookupvar('::id')
%>.
#
# This file is authorized for deployment ONLY on
<%= scope.lookupvar('::operatingsystem') %> <%=
scope.lookupvar('::operatingsystemmajrelease') %>.
#
# Deployment by any other user or on any other system is strictly
prohibited.
# *****
```

6. In the console, add `apache` to the available classes, and then add that class to your agent node. Refer to the Adding classes getting started guide if you need help with these steps.

When Puppet runs, it configures Apache and starts the `httpd` service. When this happens, a default Apache vhost is created based on the contents of `_file_header.erb`.

7. On the agent node, navigate to one of the following locations, depending on your operating system:

- Redhat-based: `/etc/httpd/conf.d`
- Debian-based: `/etc/apache2/sites-available`

8. View `15-default.conf`; depending on the node's OS, the header shows some variation of the following contents:

```
# *****
# Vhost template in module puppetlabs-apache
# Managed by Puppet
#
# This file is authorized for deployment by root.
#
# This file is authorized for deployment ONLY on Redhat 6.
#
# Deployment by any other user or on any other system is strictly
prohibited.
# *****
```

PE has used Factor to retrieve some key facts about your node, and then used those facts to populate the header of your vhost template.

Write a new module

Write, validate, and test an example module for PE on *nix systems with Puppet Development Kit (PDK).

Before you begin

Make sure you've installed:

- PE
- At least one *nix agent node
- The `puppetlabs-apache` module
- Puppet Development Kit on your master. See PDK [installation](#) instructions.

You must be logged in as root or administrator on your nodes.

Tip: PDK is a standalone development kit that doesn't require Puppet on your development machine. For simplicity in this example, you create your module on your master, but normally, you would develop modules on a separate development workstation and then move your module to the master.

Create a `pe_getstarted_app` module that manages a PHP-based web app running on an Apache virtual host.

1. Make sure you're in the module's directory by running `cd /etc/puppetlabs/code/environments/production/modules`.
2. On the master, from the command line, run `pdk new module pe_getstarted_app --skip-interview`.

By default, PDK asks a series of metadata questions before it creates your module. The `--skip-interview` option bypasses the interview, and PDK uses default values for the module's metadata.

PDK creates the new module's directory with the same name you gave the module. It also creates the module's metadata, subdirectories, and testing template files.

3. Change into the module directory by running `cd pe_getstarted_app`.
4. Validate your module's code syntax and style by running `pdk validate`.

The `pdk validate` command checks the Puppet and Ruby code style and syntax in your module. Validate every time you add new code.

5. Unit test your module by running `pdk test unit` to ensure that the testing directories and templates were correctly created.

On a newly generated module, the `pdk test unit` command checks that the testing directories and templates were correctly created. As you add new code to your module, write and run unit tests to make sure your code works.

Create a module class

Write, test, and deploy a module class for an example module.

Before you begin

Make sure you have generated the `pe_getstarted_app` module.

1. In the `pe_getstarted_app` directory, create the main class for your module by running `pdk new class pe_getstarted_app`.

When you create a class with the same name as your module, PDK creates the `init.pp` file. This file contains the main class of a module, and it's the only class with a file name that's different from the class name. PDK also creates testing directories and templates for the class.

2. Validate your class's code syntax and style by running `pdk validate`.
3. Unit test your class by running `pdk test unit`.
4. Open the `init.pp` file in your text editor and add the following Puppet code to it. Save the file and exit the editor.

```
class pe_getstarted_app (
  $content = "<?php phpinfo() ?>\n",
) {

  class { 'apache':
    mpm_module => 'prefork',
  }

  include apache::mod::php

  apache::vhost { 'pe_getstarted_app':
    port      => '80',
    docroot   => '/var/www/pe_getstarted_app',
    priority  => '10',
  }
```

```

file { ['/var/www/pe_getstarted_app/index.php']:
  ensure => file,
  content => $content,
  mode    => '0644',
}

```

Additional details about the code in your new class:

- The class `apache` is modified to include the `mpm_module` attribute. This attribute determines which multi-process module is configured and loaded for the Apache (HTTPD) process. In this case, the value is set to `'prefork'`.
- `include apache::mod::php` indicates that your new class relies on those classes to function correctly. PE understands that your node needs to be classified with these classes and completes that work automatically when you classify your node with the `pe_getstarted_app` class; in other words, you don't need to worry about classifying your nodes with Apache and Apache PHP.
- The priority attribute of `'10'` ensures that your app has a higher priority on port 80 than the default Apache `vhost` app.
- The file `/var/pe_getstarted_app/index.php` contains whatever is specified in the `content` attribute. This is the content you see when you launch your app. PE uses the `ensure` attribute to create that file the first time the class is applied.

After creating your class, you are ready to validate and unit test it.

Validate and unit test a class

Validate and unit test a class in an example module.

When you add code to a module, you should always validate and unit test it. PDK creates unit test directories and templates, and then you write the unit tests you need. For this example, we've provided an `rspec` unit test. To learn more about how to write unit tests, see [rspec-puppet](#) documentation.

1. In the `pe_getstarted_app` directory, create a `.fixtures.yml` file. This file installs module dependencies for unit testing.
2. Open the `.fixtures.yml` file and edit it to include the following code. Save the file and exit the editor.

```

fixtures:
  forge_modules:
    apache: "puppetlabs/apache"
    stdlib: "puppetlabs/stdlib"
    concat: "puppetlabs/concat"

```

3. Change into the spec tests directory by running `cd spec/classes`.
4. Open the `pe_getstarted_app_spec.rb` in your text editor and add the following code. Save and exit the file.

```

require 'spec_helper'

describe 'pe_getstarted_app', type: :class do
  let(:facts) do
    {
      operatingsystemrelease: '18.04',
      osfamily: 'Debian',
      operatingsystem: 'Ubuntu',
      lsbdistrelease: 'Bionic',
    }
  end

  describe 'standard content' do

```



```

    it { is_expected.to contain_class('apache').with('mpm_module' =>
'prefork') }

    it {
      is_expected.to contain_apache__vhost('pe_getstarted_app').with(
        'port' => '80',
      )
    }

    it {
      is_expected.to contain_file('/var/www/pe_getstarted_app/
index.php').with(
        'ensure' => 'file',
        'content' => "<?php phpinfo() ?>\n",
        'mode' => '0644'
      )
    }
  end

  describe 'custom content' do
    let(:params) do
      { 'content' => "custom\n", }
    end

    it {
      is_expected.to contain_file('/var/www/pe_getstarted_app/
index.php').with(
        'ensure' => 'file',
        'content' => "custom\n",
        'mode' => '0644'
      )
    }
  end
end

```

5. Run `pdk validate` to check the code style and syntax of your class.

6. Run `pdk test unit` to run your unit test against your class.

Adding your module class to nodes

Add the class from your example module to nodes in the console.

You can now add your new module's class to the console and apply it to nodes, following the workflow in the "Apply the Apache class to your agent node" section.

Congratulations! Your first module is up and running. There are plenty of additional resources about modules and the creation of modules that you can reference. Check out documentation about [Puppet Development Kit](#), [module fundamentals](#), [the modulepath](#), the [Beginner's guide to modules](#), and the [Puppet Forge](#).

Synchronize your clocks with NTP

Make sure your clocks stay in sync by using a module from the Forge to configure NTP.

NTP overview

The clocks on your servers need to synchronize with something to let them know what the right time is. NTP is a protocol that synchronizes computer clocks over a network to within a millisecond, using Coordinated Universal Time (UTC). Follow this section to ensure all your nodes and processes in your infrastructure are syncing correctly.

NTP is one of the most crucial, yet easy, services to configure and manage with Puppet Enterprise. Using the `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.

You will:

- Install a module
- Create a node group
- Classify nodes
- Set parameters

About module directories

By default, Puppet keeps modules in `/etc/puppetlabs/code/environments/production/modules`. This includes modules that you download from the Forge and those you write yourself.

PE also creates two other module directories: `/opt/puppetlabs/puppet/modules` and `/etc/puppetlabs/staging-code/modules`. For this guide, don't modify or add anything to either of these directories.

There are some resources about modules and the creation of modules that you can reference.

- [Welcome to Puppet modules](#)
- [Module fundamentals](#)
- [The Forge](#)

Install the NTP module

Install the `puppetlabs-ntp` module, which helps manage your NTP service.

From the command line of your Puppet master, run `puppet module install puppetlabs-ntp`

The output looks similar to the following:

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

That's it! You've just installed the `ntp` module. You must wait a short time for the Puppet server to refresh before the classes are available to add to your agent nodes.

Add the `ntp` class from the module

Using the PE console, you add the module's `ntp` class to a node group that you create, called **NTP**, which contains all of your nodes. Depending on your needs or infrastructure, you might have a different group that you assign NTP to, but these same instructions apply.

Create the NTP node group

The role of a classification node group is to assign classification data, such as classes, parameters, and variables, to nodes .

1. In the console, click **Classification**, and click **Add group**.
2. Specify options for the new node group:
 - **Parent name** : select **All nodes**
 - **Group name** : enter a name that describes the role of this environment node group
 - **Environment**: select **production**
 - **Environment group**: don't select this option
3. Click **Add**
4. Click the **NTP** group, and select the **Rules** tab.
5. In the **Fact** field, enter `name`.
6. From the **Operator** drop-down list, select `~` (matches regex).

7. In the **Value** field, enter `. *`.
8. Click **Add rule**.

This rule "dynamically" pins all nodes to the **NTP** group. Note that this rule is for testing purposes and that decisions about pinning nodes to groups in a production environment vary from user to user.

Add the `ntp` class to the **NTP** group

Node groups contain classes and other elements.

1. In the console, click **Classification**, and find and select the **NTP** group.
2. On the **Configuration** tab, in the **Add new class** field, select `ntp`.

Tip: You need to add only the main `ntp` class; it contains the other classes from the module.

3. Click **Add class**, and commit changes.

Note: The `ntp` class now appears in the list of classes for the **NTP** group, but it has not yet been configured on your nodes. For that to happen, you need to kick off a Puppet run.

4. From the command line of your Puppet master, run `puppet agent -t`.
5. From the command line of each PE-managed node, run `puppet agent -t`.

This configures the nodes using the newly-assigned classes.

Success! Puppet Enterprise is now managing NTP on the nodes in the **NTP** group. So, for example, if you forget to restart the NTP service on one of those nodes after running `ntpdate`, PE automatically restarts it on the next Puppet run.

View `ntp` changes in the PE console

You can view and research infrastructure changes and events on the console's Events page. After applying the `ntp` class, check the Events page to confirm that changes were indeed made to your infrastructure.

Note that in the summary pane on the left, one event, a successful change, has been recorded for Nodes: with events. However, there are two changes for Classes: with events and Resources: with events. This is because the `ntp` class loaded from the `ntp` module contains additional classes---a class that handles the configuration of NTP (`Ntp::Config`) and a class that handles the NTP service (`Ntp::Service`).

1. Click Intentional changes in the **Classes: with events** summary view. The main pane shows you that the `Ntp::Config` and `Ntp::Service` classes were successfully added when you ran PE after adding the main `ntp` class.
2. Navigate through further levels to see more data.

If you continue to navigate down, you end up at a run summary that shows you the details of the event. For example, you can see exactly which piece of Puppet code was responsible for generating the event. In this case, it was line 15 of the `service.pp` manifest and line 21 of the `config.pp` manifest from the `puppetlabs-ntp` module.

If there had been a problem applying this class, this information would tell you exactly which piece of code you need to fix. In this case, the **Events** page lets you confirm that PE is now managing NTP.

In the upper right corner of the detail pane is a link to a run report, which contains information about the Puppet run that made the change, including logs and metrics about the run. See [Infrastructure reports](#) for more information.

For more information about using the Events page, see [Working with the Events page](#).

Edit parameters of the `ntp` class

You can edit or add class parameters in the PE console without needing to edit the module code directly.

The NTP module, by default, uses public NTP servers. But what if your infrastructure runs an internal pool of NTP servers? You can change the server parameter of the `ntp` class in a few steps using the PE console.

1. In the console, click **Classification**, and find and select the **NTP** group.
2. On the **Configuration** tab, find `ntp` in the list of classes.

3. From the **Parameter name** drop-down list, choose `servers`.

Note: The grey text that appears as values for some parameters is the default value, which can be either a literal value or a Puppet variable. You can restore this value by selecting **Discard changes** after you have added the parameter.

4. In the **Value** field, enter the new server name (for example, ["time.apple.com"]). Note that this must be an array, in JSON format.
5. Click **Add parameter**, and commit changes.
6. From the command line of your Puppet master, run `puppet agent -t`.
7. From the command line of each PE-managed node, run `puppet agent -t`.

This triggers a Puppet run that causes Puppet Enterprise to create the new configuration.

Puppet Enterprise now uses the NTP server you specified for that node.

Tip: Remember to check the **Events** page to be sure the changes were correctly applied to your nodes!

Learn more about NTP

You can learn more about NTP from the Puppet blog.

- [Automating with Puppet: Standardizing NTP](#)

Next, learn about managing your DNS nameserver with PE.

Resolve nodes to your internal nameserver

To manage your nameserver, write your own module and configure it in the console.

DNS overview

A nameserver ensures that the human-readable names you type in your browser (for example, google.com) can be resolved to IP addresses that computers can read.

Sysadmins typically need to manage a nameserver file for internal resources that aren't published in public nameservers. For example, let's say 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 (say at 10.16.22.10), and then use PE to ensure all the servers in your infrastructure have access to it.

Note: You can add the DNS nameserver class to as many agents as needed. For ease of explanation, our instructions might show only one agent.

You will:

- Write a module
- Create a node group
- Classify nodes
- Set parameters

About module directories

By default, Puppet keeps modules in `/etc/puppetlabs/code/environments/production/modules`. This includes modules that you download from the Forge and those you write yourself.

PE also creates two other module directories: `/opt/puppetlabs/puppet/modules` and `/etc/puppetlabs/staging-code/modules`. For this guide, don't modify or add anything to either of these directories.

There are some resources about modules and the creation of modules that you can reference.

- [Welcome to Puppet modules](#)
- [Module fundamentals](#)
- [The Forge](#)

Write the resolver module

Write a small module to ensure that your nodes resolve to your internal nameserver.

This module contains just one class and one template. Modules are directory trees. For this task, you create the following files:

- `resolver/` (the module name)
 - `manifests/`
 - `init.pp` (contains the `resolver` class)
 - `templates/`
 - `resolve.conf.erb` (contains the template for the `/etc/resolv.conf` template, the contents of which are populated after you add the class and run PE.)
1. From the command line on the Puppet master, navigate to the modules directory: `cd /etc/puppetlabs/code/environments/production/modules`
 2. Run `mkdir -p resolver/manifests`
 3. From the `manifests` directory, use your text editor to create the `init.pp` file, and edit it so it contains the following Puppet code.

```
class resolver (
  $nameservers,
) {

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

4. Save and exit the file.
5. Run `mkdir -p resolver/templates` to create the templates directory.
6. Use your text editor to create the `resolver/templates/resolv.conf.erb` file.
7. Edit the `resolv.conf.erb` so that it contains the following Ruby code.

```
# Resolv.conf generated by Puppet

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

# Other values can be added or hard-coded into the template as needed.
```

8. Save and exit the file.

That's it! You've written a module that contains a class that, after applied, ensures your nodes resolve to your internal nameserver. You must wait a short time for the Puppet server to refresh before the classes are available to add to your agents.

Note the following about your new class:

- The class `resolver` 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`. You set this content in the next task using the PE console.

Create the DNS node group

To manage DNS on your nodes, create a new node group that contains all of your nodes.

Create the DNS node group to contain all the nodes in your deployment (including the master). You can create your own groups or add the classes to individual nodes, depending on your needs.

1. In the console, click **Classification**, then **Add group**.
2. Specify options for the new node group:
 - **Parent name** – Select **default**
 - **Group name** - Enter a name that describes the role of this environment node group, for example, DNS.
 - **Environment** - Select **Production**
 - **Environment group** - Don't select this option
3. Click **Add**.
4. Click the **DNS** group, and select the **Rules** tab.
5. In the **Fact** field, enter `name`.
6. From the **Operator** drop-down list, select `~` (matches regex).
7. In the **Value** field, enter `. * .`
8. Click **Add rule**.

This rule "dynamically" pins all nodes to the DNS group. This rule is for testing purposes; decisions about pinning nodes to groups in a production environment vary from user to user.

Add the resolver class to the DNS group

After you create a group, add a class to it.

Next, add the `resolver` class to your new DNS node group.

1. In the console, select **Classification**, and then find and select the **DNS** group.
2. On the **Configuration** tab, in the **Class name** field, select **resolver**.
3. Click **Add class**, and commit changes.

Note: The `resolver` class now appears in the list of classes for the DNS group, but it has not yet been configured on your nodes. For that to happen, you need to kick off a Puppet run.

4. From the command line of your Puppet master, run `puppet agent -t`.
5. From the command line of each PE-managed node, run `puppet agent -t`.

This configures the nodes using the newly-assigned classes. Wait one or two minutes.

You're not done just yet! The `resolver` class now appears in the list of classes for your DNS group, but it has not yet been fully configured. You still need to add the nameserver IP address parameter for the `resolver` class to use. You can do this by adding a parameter right in the console.

Add the nameserver IP address parameter in the console

You can add class parameter values to the code in your module, but it's easier to add those parameter values to your classes using the PE console.

1. In the console, select **Classification**, and then find and select the **DNS** group.
2. On the **Configuration** tab, find `resolver` in the list of classes.
3. From the **parameter** drop-down list, select `nameservers`.
4. In the **Value** field, enter the nameserver IP address you'd like to use (for example, `8.8.8.8`).

Note: The grey text that appears as values for some parameters is the default value, which can be either a literal value or a Puppet variable. You can restore this value by selecting **Discard changes** after you have added the parameter.

5. Click **Add parameter**, and commit changes.
6. From the command line of your Puppet master, run `puppet agent -t`.

7. From the command line of each PE-managed node, run `puppet agent -t`.

This triggers a Puppet run to have Puppet Enterprise create the new configuration.

8. Navigate to `/etc/resolv.conf`. This file now contains the contents of the `resolv.conf.erb` template and the nameserver IP address you added in step 5.

Puppet Enterprise now uses the nameserver IP address you specified for that node.

Viewing DNS changes on the Events page

The **Events** page lets you view and research changes. You can view changes by class, resource, or node.

After applying the `resolver` class, you can use the **Events** page to confirm that changes were indeed made to your infrastructure, most notably that the class created `/etc/resolv.conf` and set the contents as specified by the module's template.

The further you drill down in this page, the more detail you receive. If there had been a problem applying the `resolver` class, this information tells you exactly where that problem occurred or which piece of code you need to fix.

You can click **Reports**, which contains information about the changes made during Puppet runs, including logs and metrics about the run. See Infrastructure reports for more info.

For more information about using the **Events** page, see Working with the Events page.

Check that PE enforces the desired state of the resolver class

If your infrastructure changes from what you've specified, PE corrects that change. To test this, make a manual infrastructure change and then run Puppet.

When you set up DNS nameserver management, you set the nameserver IP address. If a member of your team changes the contents of `/etc/resolv.conf` to use a different nameserver, blocking access to internal resources, the next Puppet run corrects this. You can test this by manually changing the `resolv.conf` file.

1. On any agent to which you applied the `resolv.conf`, edit `/etc/resolv.conf` to be any nameserver IP address other than the one you want to use.
2. Save and exit the file.
3. After Puppet runs, navigate to
4. Puppet runs.
5. Navigate to `/etc/resolv.conf`, and notice that PE has enforced the desired state you specified for the nameserver IP address.

That's it! PE has enforced the desired state of your agent node. And remember, review the changes to the class or node using the **Events** page.

Learn more about DNS

Check out the Puppet blog for more information on DNS.

- [Dealing with name resolution issues](#)

Next, allow encryption by enabling SSH.

Manage SSH keys and permissions

Add SSH capabilities to your infrastructure using a module from the Forge.

SSH overview

Secure Shell (SSH) is a protocol that enables encrypted connections between nodes on a network for administrative purposes. This guide provides instructions for getting started managing SSH across your PE deployment using a module from the Puppet Forge.

SSH is an important part of an IT infrastructure. With it, you can access remote systems securely, even through insecure networks, by using keys. This allows you to manage your nodes from one machine.

Typically, the first time you attempt to SSH into a host you've never connected to before, you get a warning similar to the following:

```
The authenticity of host '10.10.10.9 (10.10.10.9)' can't be established.  
RSA key fingerprint is 05:75:12:9a:64:2f:29:27:39:35:a6:92:2b:54:79:5f.  
Are you sure you want to continue connecting (yes/no)?
```

If you select yes, the public key for that host is added to your SSH `known_hosts` file, and you won't have to authenticate it again unless that host's key changes.

You will:

- Install a module
- Create a node group
- Classify nodes
- Set parameters

About module directories

By default, Puppet keeps modules in `/etc/puppetlabs/code/environments/production/modules`. This includes modules that you download from the Forge and those you write yourself.

PE also creates two other module directories: `/opt/puppetlabs/puppet/modules` and `/etc/puppetlabs/staging-code/modules`. For this guide, don't modify or add anything to either of these directories.

There are some resources about modules and the creation of modules that you can reference.

- [Welcome to Puppet modules](#)
- [Module fundamentals](#)
- [The Forge](#)

Install the `ghoneycutt-ssh` module

The `ghoneycutt-ssh` module manages SSH keys and removes SSH keys that you aren't managing with Puppet.

This module, available on the Puppet Forge, is one of many modules written by our user community. It contains one class: the `ssh` class. You can learn more about the `ghoneycutt-ssh` module by visiting the Forge.

On the command line, on the PE master, run `puppet module install ghoneycutt-ssh -v 3.40.0`.

The output looks similar to the following:

```
Notice: Preparing to install into /etc/puppetlabs/code/environments/  
production/modules ...  
Notice: Downloading from https://forgeapi.puppetlabs.com ...  
Notice: Installing -- do not interrupt ...  
/etc/puppetlabs/code/environments/production/modules  
### ghoneycutt-ssh (v3.40.0)  
### ghoneycutt-common (v1.6.0)  
### puppetlabs-firewall (v1.8.1)  
### puppetlabs-stdlib (v4.9.1)
```

That's it! You've just installed the `ghoneycutt-ssh` module. You must wait a short time for the Puppet server to refresh before the classes are available to add to your agent nodes.

Create the SSH node group

In the console, create a group called `ssh_example` to designate which nodes Puppet must manage SSH on.

This group contains all of your nodes. Depending on your needs or infrastructure, you might have a different group that you assign SSH to.

1. In the console, click **Classification**, and click **Add group**.

2. Specify options for the new node group:
 - **Parent name** : select **All nodes**
 - **Group name** : enter a name that describes the role of this environment node group
 - **Environment**: select **production**
 - **Environment group**: don't select this option
3. Click **Add**
4. Click the `ssh_example` group, then select the **Rules** tab.
5. In the **Fact** field, enter `name`.
6. From the **Operator** drop-down list, select `~` (matches regex).
7. In the **Value** field, enter `. * .`
8. Click **Add rule**.

This rule "dynamically" pins all nodes to the `ssh_example` group. Note that this rule is for testing purposes, and that decisions about pinning nodes to groups in a production environment vary.

Related topics:

Adding nodes dynamically

Add classes from the ssh module

Add the `ssh` class to the `ssh_example` node group to add the necessary resources that allow Puppet to manage SSH.

Depending on your needs or infrastructure, you might have a different group that you assign SSH to, but these same instructions apply.

After you apply the `ssh` class and run Puppet, the public key for each agent node is exported and then disseminated to the `known_hosts` files of the other agent nodes in the group, and you are no longer asked to authenticate those nodes on future SSH attempts.

1. In the console, click **Classification**, and find and select `ssh_example` group.
2. On the **Configuration** tab, in the **Class name** field, select `ssh`.
3. Click **Add class**, and commit changes.

Note: The `ssh` class now appears in the list of classes for the `ssh_example` group, but it has not yet been configured on your nodes. For that to happen, kick off a Puppet run.

4. From the command line of your master, run `puppet agent -t`.
5. From the command line of each -managed node, run `puppet agent -t`.

This configures the nodes using the newly assigned classes. Wait one or two minutes.

Important: You need to run Puppet a second time due to the round-robin nature of the key sharing. In other words, the first server that ran on the first Puppet run was only able to share its key, but it was not able to retrieve the keys from the other agents. It collects the other keys on the second Puppet run.

View changes made by the ssh class

To confirm that the `ssh` class made changes to your infrastructure, check the **Events** console page. This page lets you view and research changes and other events.

For example, after applying the `ssh` class, you can use the **Events** page to confirm that changes were indeed made to your infrastructure.

Note that in the summary pane on the left, one event, a successful change, has been recorded for **Classes: with events**. However, there are three changes for **Classes: with events** and six changes for **Resources: with events**.

1. Click **With changes** in the **Classes: with events** summary view.

The main pane shows you that the `ssh` class was successfully added when you ran PE. This class sets the `known_hosts` entries after it collects the public keys from agents nodes in your deployment .

2. Click **Changed** in the **Resources: with events** summary view.

The main page shows you that public key resources for each agent in our example has now been brought under PE management. The further down you navigate, the more information you receive about the event. For example, in this case, you see that the the SSH rsa key for `agent1.example.com` has been created and is now present in the `known_hosts` file for `master.example.com`.

If there had been a problem applying any piece of the `ssh` class, the information found here could tell you exactly which piece of code you need to fix. In this case, the **Events** page simply lets you confirm that PE is now managing SSH keys.

In the upper right corner of the detail pane is a link to a run report which contains information about the Puppet run that made the change, including logs and metrics about the run. See Infrastructure reports for more information.

For more information about using the **Events** page, see Working with the Events page.

Edit root login parameters of the `ssh` class

Edit or add class parameters from the `ssh` class in the PE console without needing to edit the module code directly.

The `ghoneycutt-ssh` module, by default, allows root login over SSH. But if your compliance protocols do not allow this, you can change this parameter of the `ssh` class in a few steps using the PE console.

1. In the console, click **Classification**, and find and select the `ssh_example` group.
2. On the **Configuration** tab, find `ssh` in the list of classes.
3. From the **parameter** drop-down menu, choose **permit_root_login**.

Note: The grey text that appears as values for some parameters is the default value, which can be either a literal value or a Puppet variable. You can restore this value by clicking **Discard changes** after you have added the parameter.

4. In the **Value** field, enter `no`.
5. Click **Add parameter**, and commit changes.
6. From the command line of your master, run `puppet agent -t`.
7. From the command line of each -managed node, run `puppet agent -t`.

Puppet Enterprise is now managing the root login parameter for your SSH configuration. You can see this setting in `/etc/ssh/sshd_config`. For fun, change the `PermitRootLogin` parameter to `yes`, run PE, and then recheck this file. As long as the parameter is set to `no` in the PE console, the parameter in this file is set back to `no` on every Puppet run if it is ever changed.

You can use the console to manage other SSH parameters, such as agent forwarding, X11 forwarding, and password authentication.

Learn more about SSH

Check out a few Puppet blog posts to learn more about SSH.

- [How I stopped worrying and learned to love public key authentication for SSH](#)
- [Speed up SSH by reusing connections](#)
- [Using Puppet to address new SSH client vulnerability](#)

Next, learn how to create and manage sudo privileges across your infrastructure.

Control sudo privileges

Add sudo capabilities to your nodes and control sudo permissions using a module from the Forge in conjunction with a simple module you write.

Sudo overview

Sudo is a program that allows users to work using elevated privileges of another user, usually called a superuser.

With sudo, you can more easily protect information or configurations that are most vulnerable to security exploitations by only allowing sudo users with elevated privileges to access them. Only those who can log in as a sudo user can make changes to or see this information. Use PE to manage sudo privileges across your nodes and ensure the correct users have elevated privileges for the right systems.

You will:

- Install a module
- Write a module
- Create a node group
- Classify nodes

About module directories

By default, Puppet keeps modules in `/etc/puppetlabs/code/environments/production/modules`. This includes modules that you download from the Forge and those you write yourself.

PE also creates two other module directories: `/opt/puppetlabs/puppet/modules` and `/etc/puppetlabs/staging-code/modules`. For this guide, don't modify or add anything to either of these directories.

There are some resources about modules and the creation of modules that you can reference.

- [Welcome to Puppet modules](#)
- [Module fundamentals](#)
- [The Forge](#)

Install the saz-sudo module

To start managing sudo configuration with Puppet Enterprise, install the `saz-sudo` module.

From the master, run `puppet module install saz-sudo`.

You should see output similar to the following:

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

That's it! You've just installed the `saz-sudo` module. Wait a short time for the Puppet server to refresh before the classes are available to add to your agents.

Write the privileges module

Manage sudo privileges with Puppet Enterprise, by writing a `privileges` module.

The `privileges` module contains the following files:

- **privileges/** (the module name)
 - **manifests/**
 - **init.pp** (contains the privileges class)

1. From the command line on the master, navigate to the modules directory:

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

2. Run `mkdir -p privileges/manifests` to create the new module directory and its manifests directory.
3. From the `manifests` directory, use your text editor to create the `init.pp` file, and edit it so that it contains the following code.

```
class privileges { user { 'root': ensure => present, password =>
  '$1$oST1TkX7$p21hU2qzMkR4Iy7HK6zWq0', shell => '/bin/bash', uid => '0', }
  sudo::conf { 'admins': ensure => present, content => '%admin ALL=(ALL)
  ALL', } sudo::conf { 'wheel': ensure => present, content => '%wheel
  ALL=(ALL) ALL', } }
```

4. Save and exit the file.

That's it! You've written a module that contains a class that, when applied, ensures that your agent nodes have the correct sudo privileges set for the root user and the "admin" and "wheel" groups. You add this class at the same time you add the `saz-sudo` module.

To learn more about what the resources in this class do, see the related topic about the resources in the `privileges` class.

About the resources in the privileges class

The `privileges` module you wrote for managing sudo privileges in your deployment contains just one class, but several resources. Each resource has a specific job.

The `privileges` module contains the following resources:

- `user 'root'`: This resource ensures that the root user has a centrally defined password and shell. Puppet enforces this configuration and report on and remediate any drift detected, such as if a rogue admin logs in and changes the password on an agent node.
- `sudo::conf 'admins'`: Create a sudoers rule to ensure that members of the admin group have the ability to run any command using sudo. This resource creates configuration fragment file to define this rule in `/etc/sudoers.d/`. It is usually called something like `10_admins`.
- `sudo::conf 'wheel'`: Create a sudoers rule to ensure that members of the wheel group have the ability to run any command using sudo. This resource creates a configuration fragment to define this rule in `/etc/sudoers.d/`. It is usually called something like `10_wheel`.

Create the Sudo node group

To specify which nodes you want to manage sudo on, set up a designated node group.

This group, called Sudo, contains all of your nodes. Depending on your needs or infrastructure, your group might be different.

1. In the console, click **Classification**, and click **Add group**.
2. Specify options for the new node group:
 - **Parent name** : select **All nodes**
 - **Group name** : enter a name that describes the role of this environment node group
 - **Environment**: select **production**
 - **Environment group**: don't select this option
3. Click **Add**
4. Click the Sudo group and select the **Rules** tab.
5. In the **Fact** field, enter `name`.
6. From the **Operator** drop-down list, select `~` (matches regex).
7. In the **Value** field, enter `. * .`
8. Click **Add rule**.

This rule "dynamically" pins all nodes to the Sudo group. Note that this rule is for testing purposes and that decisions about pinning nodes to groups in a production environment vary. To learn more, see the related topic about dynamically pinning nodes.

Related topics: Adding nodes "dynamically"

Add the privileges and sudo classes

To manage sudo configuration and privileges for the nodes in your Sudo group, add the `privileges` and `sudo` classes to your node group.

The `privileges` module you wrote has only one class (`privileges`), but the `saz-sudo` module contains several classes. If you don't want to add these classes to all of your nodes, you can pin the nodes "statically" or write a different rule to add them "dynamically", depending on your needs. See the related topics about adding nodes dynamically or statically for more information.

1. In the console, click **Classification**, and find and select the Sudo group.

2. On the **Configuration** tab, in the **Class name** field, enter `sudo`.
3. Click **Add class**, and commit changes.

Note: The `sudo` class now appears in the list of classes for the Sudo group, but it has not yet been configured on your nodes. For that to happen, you need to kick off a Puppet run.

4. Repeat steps 2 and 3 to add the `privileges` class.
5. From the command line of your master, run `puppet agent -t`.
6. From the command line of each -managed node, run `puppet agent -t`.

This configures the nodes using the newly-assigned classes. Wait one or two minutes.

Congratulations! You've just created the `privileges` class that you can use to define and enforce a sudoers configuration across your PE-managed infrastructure.

Learn more about sudo

Learn more about the `sudo` module in the Puppet blog.

- [Module of the Week: saz/sudo - Manage sudo configuration](#)

Next steps

Now that you have your system up and running with PE, here are some recommendations on where to go next.

- Check out the [Forge](#) to download additional modules and start managing other things like [IIS](#) sites or machines running on [Azure](#).
- Learn how to develop high-quality modules with the [Puppet VSCode extension](#) and the [Puppet Development Kit \(PDK\)](#).
- See the [Configuring Puppet Enterprise](#) on page 189 docs to fine-tune things like the console, orchestration services, java, and proxy settings.
- See the docs on [Managing nodes](#) on page 314 to add more nodes to your inventory and classify them.
- Learn about [Managing and deploying Puppet code](#) on page 550.
- Check out our [Youtube channel](#).

Getting started with Puppet Enterprise on Windows

Puppet Enterprise (PE) is automation software that can help you and your organization be productive and agile while managing your IT infrastructure.

Important: For the most part, interacting with Puppet is the same regardless of your operating system. The key difference between other operating systems and Windows is that you cannot configure a Windows machine to be a Puppet master. Agent components can be installed on Windows machines and you can manage those machines with your Linux master.

What is Puppet Enterprise (PE)?

Puppet Enterprise (PE) blends together Puppet, Bolt, and a robust selection of tools to help you automatically and continuously manage your large infrastructure.

PE is a commercial version of Puppet, our original open source product often used by individuals managing smaller infrastructures. It has all the best parts of Puppet, plus other features like a graphical user interface, orchestration services, role-based access control, reporting, and the capability to manage thousands of nodes at once. PE also incorporates other tools and products within Puppet to create comprehensive configuration management software.

Here is a little more information about what goes into PE.

The Puppet language for maintaining a solid infrastructure

Puppet is a declarative language. This means that instead of you actively managing your configurations, you tell Puppet what your desired infrastructure looks like and it works to keep it that way. This is the opposite of an imperative language, like Java, which focuses on what things are required to get to an end result.

Bolt for when you need it done now

Bolt is another Puppet product that helps users do one-off tasks without disrupting anything in their configurations. PE allows you to run Bolt tasks and plans from the console or on the command line to quickly do things like restart services or inspect a fact on your system.

CD4PE and the PDK for helping you manage code

PE works best in conjunction with two of Puppet's coding tools -[Continuous Delivery for PE \(CD4PE\)](#) and the [Puppet Development Kit \(PDK\)](#). CD4PE is a tool that help you streamline testing and deployment of Puppet code and the PDK is a tool for building quality modules.

Together, they make your Puppet code easier to manage.

What can PE do?

You know a little about what PE is and what parts of Puppet went into creating it, but you still might be wondering what problems it can help you solve.

Here are just a few examples of things PE can do for you.

- Automatically synchronize the clocks on all of your servers to reduce failures and errors.
- Deploy IT permissions changes to entire job roles or departments.
- Alert you of any unauthorized changes to your infrastructure.
- Quickly make profile changes to all user accounts at once.
- Allow you to view and filter information about nodes.
- Schedule reports.

Check out our [video](#) for more information.

Key components of PE

Here are notes about some the key components within PE that will help you along the way.

The Puppet master

The Puppet master, or master, is the central hub of activity and process in Puppet Enterprise. This is where PE compiles code to create agent catalogs and where SSL certificates are verified and signed.

Puppet Server

Puppet Server is an application that runs on the Java Virtual Machine (JVM) on the master. In addition to hosting endpoints for the certificate authority service, it also powers the catalog compiler.

Catalog

To configure a managed node, the agent uses a document called a catalog. The catalog describes the desired state for each resource that should be managed on the node. A resource is a configurable unit of information, like a user or service.

Agents

Agents are responsible for ensuring your configuration stays in the desired state by periodically checking it against the catalog. If there are changes, the agent sends a report to the master or compiler and changes it back to the desired state based on what the catalog says.

Role-based access control (RBAC)

RBAC is the way you manage user permissions in PE. With RBAC, you define a role within your infrastructure, configure the permissions within the role, and assign users who need those permissions to that role. This way, you do not have to manage individual user permissions.

Node classifier (NC)

PE comes with its own node classifier, which is built into the console. Classification is when you configure your managed nodes by assigning classes to them. **Classes** provide the Puppet code—distributed in modules—that enables you to define the function of a managed node, or apply specific settings and values to it.

Modules

Modules are self-contained, shareable bundle of code and data. Each module manages a specific task in your infrastructure, like installing and configuring a piece of software. Most Puppet code is written within modules. You can write your own modules or download pre-built modules from the Forge.

Forge

The [Forge](#) is a Puppet-run repository for storing, sharing, and installing modules. Some modules are approved by Puppet, meaning they have been tested rigorously by Puppet and Puppet maintains them. Other modules are built and submitted by users or organizations.

Puppet Development Kit (PDK)

The [PDK](#) is downloadable tool for building modules. It provides a command line interface and integrated testing features to help you develop high-quality code. We highly recommend checking out the PDK when you begin writing your own code.

Install and administer Puppet Enterprise on Windows

Whether you're setting up a PE installation for actual deployment or want to learn some fundamentals of configuration management with PE, we'll provide you with the steps you need to get up and running relatively quickly.

The steps and concepts are organized in the order that you would most likely perform or encounter them.

Start installing PE in a Windows environment

The express install of PE is ideal for trying out PE with up to 10 nodes and can be used to manage up to 4,000 nodes.

Important: The Puppet master can run only on *nix machines, but Windows machines can run as Puppet agents and you can manage those nodes with your *nix Puppet master. This getting started guide assumes you want to access the master *nix machine remotely from a Windows machine. If you plan to install directly onto a *nix node, follow the [installing PE instructions in the *nix getting started guide](#).

On a single *nix machine, you install:

- The Puppet master, the central hub of activity, where Puppet code is compiled to create agent catalogs, and where SSL certificates are verified and signed.
- The console, PE's web interface, which features numerous configuration and reporting tools.
- PuppetDB, which collects data generated throughout your Puppet infrastructure.

Step 1: Review installation prerequisites on your Linux server

Before getting started, review this checklist to make sure you're ready to install PE.

Note: The examples in this guide use a Linux server running Red Hat Enterprise Linux (RHEL) 6.

1. Be aware that you must work as the `root` user on the command line throughout the installation process.
2. Make sure you meet the [hardware recommendations](#) for 10 or fewer nodes.

You can download and install Puppet Enterprise on up to 10 nodes at no charge.

3. Make sure that DNS is properly configured on the server you're installing on.
 - All nodes must know their own hostnames, which you can achieve by properly configuring reverse DNS on your local DNS server, or by setting the hostname explicitly. Setting the hostname usually involves the `hostname` command and one or more configuration files, but the exact method varies by platform.
 - All nodes must be able to reach each other by name, which you can achieve with a local DNS server.
4. Know the fully qualified domain name (FQDN) of the server you're installing PE on, for example, `master.example.com`.

Related information

[Hardware requirements](#) on page 114

These hardware requirements are based on internal testing at Puppet and are meant only as guidelines to help you determine your hardware needs.

Step 2: Prepare your Windows System

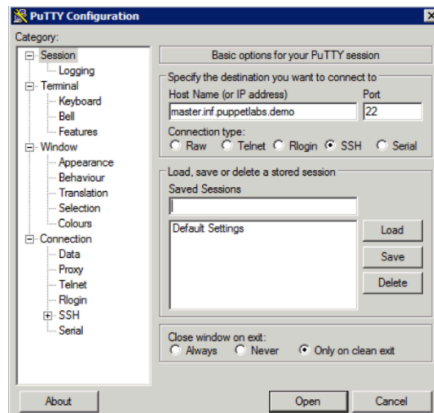
Some extra software is necessary so that you can communicate and work between your Windows and Linux machines.

On your Windows machine set up an SSH Client. The examples below use the Putty SSH client.

Step 3: Install the master on your Linux machine

These steps show you how to use a Windows machine to install a Puppet master on a Linux server.

1. Open the SSH client, and enter the hostname or IP address and port of the Linux machine that you want to use as your master. You can then open an SSH session to the master.



2. When prompted in the terminal, log into the Linux node as the `root` user.
3. Download the PE installer to the server that you intend to use as your master by copying the appropriate download URL from the [downloads page](#) and running:

```
wget --content-disposition '<DOWNLOAD_URL>'
```

4. Run the following command to unpack the tarball:

```
tar -xvf <TARBALL>
```

Note: You need about 1 GB of space.

5. Change directories to the installer directory, and run the installer:

```
sudo ./puppet-enterprise-installer
```

6. When prompted, select express install mode.
7. After installation completes, follow the prompts to specify a password, or run:

```
puppet infrastructure console_password --password=<MY_PASSWORD>
```

8. Run Puppet twice: `puppet agent -t`.

You must restart the shell before you can use PE client tool subcommands.

Note: From this point the examples refer to your Linux server as the master.

Related information

[Accepting the console's certificate](#) on page 55

The console uses an SSL certificate created by your own local Puppet certificate authority. Because this authority is specific to your site, web browsers won't know it or trust it, and you must add a security exception in order to access the console.

Log into the PE console

The console is a graphical interface that allows you to manage parts your PE infrastructure without relying on the command line.

Log in for the first time.

1. When you navigate to the hostname of the master in your browser, you'll receive a browser warning about an untrusted certificate. This is because you were the signing authority for the console's certificate, and your Puppet Enterprise deployment is not known to your browser as a valid signing authority. Ignore the warning and accept the certificate.
2. On the login page for the console, log in with the username `admin` and the password you created when installing the Puppet master. Keep track of this login as you will use it later.

Congratulations, you've successfully installed the Puppet master node! Next, learn how to install an agent on a node using the console.

Start installing Windows agents

When a node is managed by Puppet, it runs a Puppet agent application, commonly called an agent. In this section you install a Windows Puppet agent, which regularly pulls configuration catalogs from a Puppet master and applies them locally. These instructions include how to sign the agent certificate request in the console.

These instructions assume that you've installed PE.

How does a Puppet agent work?

Agents ensure that resources in a node stay in their desired state and is one of the ways you can manage nodes using PE.

Puppet is a declarative language. This means that instead of telling Puppet to do something, you tell it what a normal configuration looks like to you and Puppet works to maintain that state. If something changes, Puppet changes it back. Agents are responsible for catching any changes, fixing them, and submitting reports about what happened.

In the following section, you will install your first agent.

Step 1: Install an agent on your Windows machine

To install a Windows agent with PE package management, you use the `pe_repo` class to distribute an installation package to agents. You can use this method with or without internet access.

You must use PowerShell 2.0 or later to install Windows agents with PE package management.

Note: The <MASTER_HOSTNAME> portion of the installer script—as provided in the following example—refers to the FQDN of the master. The FQDN must be fully resolvable by the machine on which you're installing or upgrading the agent.

1. In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Master** group.
2. On the **Configuration** tab in the **Class name** field, select **pe_repo** and select the appropriate repo class from the list of classes.

- 64-bit (x86_64) — **pe_repo::platform::windows_x86_64**
- 32-bit (i386) — **pe_repo::platform::windows_i386**

3. Click **Add class** and commit changes.
4. On the master, run Puppet to configure the newly assigned class.

The new repository is created on the master at `/opt/puppetlabs/server/data/packages/public/<PE_VERSION>/<PLATFORM>/`.

5. On the node, open an administrative PowerShell window, and install:

```
[Net.ServicePointManager]::ServerCertificateValidationCallback
= {$true}; $webClient = New-Object System.Net.WebClient;
$webClient.DownloadFile('https://<MASTER_HOSTNAME>:8140/packages/current/
install.ps1', 'install.ps1'); .\install.ps1
```

Microsoft Windows Server 2008r2 only:

```
[Net.ServicePointManager]::ServerCertificateValidationCallback
= {$true}; $webClient = New-Object System.Net.WebClient;
$webClient.DownloadFile('https://cyzokd1lm79sd0j.<MASTER_HOST>:8140/
packages/current/install.ps1', 'install.ps1'); .\install.ps1 -v
```

After running the installer, the following output indicates the agent was successfully installed.

```
Notice: /Service[puppet]/ensure: ensure changed 'stopped' to
'running'service { 'puppet': ensure => 'running', enable => 'true', }
```

Approve the certificate request

During installation, the agent node contacts the Puppet master and requests a certificate. To add the node to the console and to start managing its configuration, you must approve its certificate request.

1. In the console, load a list of currently pending node requests by clicking **Unsigned certs**.
2. Click the **Accept All** button to approve the request and add the node.

The Puppet agent can now retrieve configurations from the master the next time Puppet runs.

Step 3: Test the Puppet agent node

You can wait until Puppet runs automatically, or you can manually trigger Puppet runs.

By default, the agent fetches configurations from the Puppet master every 30 minutes. (You can configure this interval in the `puppet.conf` file with the `runinterval` setting.) However, you can manually trigger a Puppet run from the command line at any time.

On the agent, log in as root and run `puppet agent -test` on the SSH client.

This triggers a single Puppet run on the agent with verbose logging.

Note the long string of log messages, ending with this message: Notice: Applied catalog in <N> seconds.

You are now fully managing the agent node! It has checked in with the Puppet master for the first time and received its configuration info. It continues to check in and fetch new configurations every 30 minutes.

Next, get the PE console up and running.

Start installing modules

Modules are shareable, reusable units of Puppet code that extend Puppet across your infrastructure by automating tasks such as setting up a database, web server, or mail server. In this section, you install a module from the Puppet Forge.

The Puppet Forge is a repository for modules created by Puppet and the Puppet community. It contains thousands of modules submitted by users and Puppet developers for you to use. In addition, PE customers can take advantage of supported modules, which are designed to make common services easier and are tested and maintained by Puppet.

In this section, you install the `puppetlabs-wsus_client` module, a Puppet Enterprise supported module. In a subsequent section, [Writing modules for Windows](#), you learn more about modules, including how to write your own.

The process for installing a module is the same on both Windows and *nix operating systems.

These instructions assume you've installed PE and at least one Windows agent node.

Step 1: Create the modules directory

By default, Puppet keeps modules in `C:\ProgramData\PuppetLabs\code\environments\production\modules` or, on *nix `/etc/puppetlabs/code/environments/production/modules`. This includes modules installed by PE, those that you download from the Forge, and those you write yourself. In a fresh installation, you need to create this modules subdirectory yourself.

Note: PE creates two other module directories: `/opt/puppetlabs/puppet/modules` and `/etc/puppetlabs/staging-code/modules`. Don't modify anything in or add modules of your own to `/opt/puppetlabs/puppet/modules`. The `/etc/puppetlabs/staging-code/modules` directory is for file sync use only; if you are not using Code Manager or file sync, do not add code to this directory.

Navigate to the production directory and run: `mkdir modules`

Step 2: Install a Forge module

The `wsus_client` module, or Windows Server Update Service (WSUS) module, lets Windows administrators manage operating system updates using their own servers instead of Microsoft's Windows Update servers.

Run `puppet module install puppetlabs-wsus_client`

If you're installing a module directly on your Windows machine, the output looks like this:

```
PS C:\Users\Administrator> puppet module install puppetlabs-wsus_client
Notice: Preparing to install into
C:/ProgramData/PuppetLabs/code/environments/production/modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
C:/ProgramData/PuppetLabs/code/environments/production/modules
+-- puppetlabs-wsus_client (v1.0.1)
+-- puppetlabs-registry (v1.1.3)
+-- puppetlabs-stdlib (v4.11.0)
```

This is the output (on a *nix machine):

```
Preparing to install into /etc/puppetlabs/code/environments/production/
modules ...
Notice: Downloading from http://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/etc/puppetlabs/code/environments/production/modules
### puppetlabs-wsus_client (v1.0.1)
### puppetlabs-registry (v1.1.3)
### puppetlabs-stdlib (v4.11.0)
```

You have just installed a module. You can read about this module in the [module readme](#). All of the classes in the module are now available to be added to the console and assigned to nodes.

Next: Adding classes to Windows agent nodes.

Start adding classes

In this section, you use the console to add a class to your Puppet agent. Classes are named chunks of Puppet code that are stored in modules. The class you assign in this exercise is derived from the module you installed previously.

The Puppet `wsus_client` module contains a class called `wsus_client`. In this exercise, you use the `wsus_client` class to supply the types and providers necessary to schedule updates using a WSUS server with Puppet.

To prepare the class, you create a group called `windows_example` and add the `wsus_client` class to it.

Note: The process for adding classes to agent nodes in the console is the same on both Windows and *nix operating systems.

Step 1: Create the `windows_example` group

To classify a node is to add classification data (classes, parameters, and variables) for the node group to the node.

Before you begin

These instructions assume you have installed PE, at least one Windows agent node, and the `puppetlabs-wsus_client` module.

1. In the console, click **Classification**, and click **Add group**.
2. In the **Group name** field, name your group, for example, `windows_example`, and click **Add**.
3. Click **Add membership rules, classes, and variables**.
4. On the **Rules** tab, in the **Certname** field, enter the name of the managed node you want to add to this group, and click **Pin node**.

Repeat this step for any additional nodes you want to add.

Note: Pinning a node adds the node to the group regardless of any rules specified for a node group. A pinned node remains in the node group until you manually remove it. Adding nodes dynamically describes how to use rules to add nodes to a node group.

5. Commit your changes.

Step 2: Add the `wsus_client` class to the example group

Adding the class to the group and then running Puppet configures the group to use that class.

1. In the console, click **Classification**, and find and select the `windows_example` group.
2. On the **Configuration** tab, in the **Add new class** field, select `wsus_client`.

If `wsus_client` doesn't appear in the list, you might have to click **Refresh**.

3. Click **Add class**, and commit changes.

The `wsus_client` class now appears in the list of classes for your agent node.

4. Puppet runs, which configures the `windows_example` group using the newly-assigned class. Wait one or two minutes.

Start assigning user access

The console enables you to import users and groups, create user roles, and assign users to roles. In this exercise, you create a user role, and give the role view permissions on the node group you previously created. Then you create a local user, and assign a user role to that user.

You can connect Puppet Enterprise (PE) with an external directory, such as Active Directory or OpenLDAP, and import users and groups, rather than creating and maintaining users and groups in multiple locations. You can create user roles, and assign imported users to those roles. Roles are granted permissions, such as permission to act on node groups. When you assign roles to users or user groups, you are granting users permissions in a more organized way.

This exercise doesn't cover connecting with an OpenLDAP or Active Directory.

Note: Users and user groups are not currently deletable. And roles are deletable by API, not in the console. Therefore, we recommend that you try out these steps on a virtual machine.

Step 1: Create a user role

Add a user role so you can manage permissions for groups of users at one time.

Before you begin

Ensure you have installed PE, at least one Windows agent node, the `puppetlabs-wsus_client` module, and that you've classified a node.

You must have admin permissions to complete these steps, which include assigning a user to a role.

1. In the console, click **User Roles**.
2. For **Name**, type `Windows users`, and then for **Description**, type a description for the role, such as `Windows users`.
3. Click **Add role**.

Step 2: Create a user and add the user to your role

These steps demonstrate how to create a new local user.

1. In the console, click **Users**.
2. In the **Full name** field, type in a user name.
3. In the **Login** field, type a username for the user.
4. Click **Add local user**.

Note: When you create new local users, you need to send them a login token. Do this by clicking the new user's name in the **User list** and then on the upper-right of the user's page, click **Generate password reset**. A message opens with a link that you must copy and send to the new user.

5. Click **User Roles** and then click **users**.
6. On the **Member users** tab, on the **User name** list, select the new user you created, and then click **Add user** and click the **Commit** button.

Step 3: Enable a user to log in

When you create new local users, you need to send them a password reset token so that they can log in for the first time.

1. On the **Users** page, click the new local user. The new user's page opens.
2. On the upper-right of the page, click **Generate password reset**. A **Password reset link** message box opens.
3. Copy the link provided in the message and send it to the new user. Then you can close the message.

Step 4: Give your role access to the node group you created

You must give the role access to the group, so that the `Windows users` role can view the `windows_example` node group.

1. From the **Windows users role** page, click the **Permissions** tab.
2. In the **Type** box, select **Node groups**.
3. In the **Permission** box, select **View**.
4. In the **Object** box, select `windows_example`.
5. Click **Add permission**, and then click the commit button.

Start writing modules for Windows

In this section, learn about Puppet modules and module development by writing your own basic module. Create a site module and use the console to apply your new module's class to a group.

Note: This guide assumes that you are not using r10k for Code Manager. If you are using r10k, any modules not managed by r10k are destroyed.

Module basics

Modules are directory trees. Many modules contain more than one directory.

By default, modules are stored in `/etc/puppetlabs/code/environments/production/modules`. If you're working from a Windows machine, the path is, `C:\ProgramData\PuppetLabs\code\environments\production\modules`. You can configure this path with the `modulepath` setting in `puppet.conf`.

The manifest directory of the Puppet `wsus_client` module contains the following files:

- `wsus_client/` (the module name)
 - `manifests/`
 - `init.pp` (contains the `wsus_client` class)
 - `service.pp` (defines `wsus_client::service`)

Every manifest (`.pp`) file contains a single class. File names map to class names in a predictable way: `init.pp` contains a class with the same name as the module; `<NAME>.pp` contains a class called `<MODULE NAME>::<NAME>`; and `<NAME>/<OTHER NAME>.pp` contains `<MODULE NAME>::<NAME>::<OTHER NAME>`.

Many modules contain directories other than `manifests`; for simplicity's sake, we do not cover them in this introductory section.

Write a Puppet module

Puppet modules save time, but at some point most users also need to write their own modules.

These instructions assume you have completed all of the preceding sections in *Getting started with Puppet Enterprise for Windows users*.

Step 1: Write a class in a module

Follow these steps to create a module with a single class called `critical_policy` that manages a collection of important settings and options in your Windows registry, most notably the legal caption and text users see before the login screen.

The new class has these characteristics:

- The `registry::value` defined resource type allows you to use Puppet to manage the parent key for a particular value automatically.
 - The `key` parameter specifies the path the key the values must be in.
 - The `value` parameter lists the name of the registry values to manage. This is copied from the resource title if not specified.
 - The `type` parameter determines the type of the registry values. Defaults to 'string'. Valid values are 'string', 'array', 'dword', 'qword', 'binary', or 'expand'.
 - `data` Lists the data inside the registry value.
1. On the Puppet master, make sure you're still in the modules directory, `cd /etc/puppetlabs/code/environments/production/modules`, and then run `mkdir -p critical_policy/manifests` to create the new module directory and its manifests directory.
 2. Use a text editor, for example Visual Studio Code (VS Code), to create and open the `critical_policy/manifests/init.pp` file. Note that Puppet has an [extension](#) for VS Code that supports syntax highlighting of the Puppet language.

3. Edit the `init.pp` file so it contains the following Puppet code, and then save it and exit the editor:

```
class critical_policy {

  registry::value { 'Legal notice caption':
    key    => 'HKLM\Software\Microsoft\Windows\CurrentVersion\Policies\
\System',
    value  => 'legalnoticecaption',
    data   => 'Legal Notice',
  }

  registry::value { 'Legal notice text':
    key    => 'HKLM\Software\Microsoft\Windows\CurrentVersion\Policies\
\System',
    value  => 'legalnoticetext',
    data   => 'Login constitutes acceptance of the End User Agreement',
  }
}
```

For more information about writing classes, refer to the following documentation:

- To learn how to write resource declarations, conditionals, and classes in a guided tour format, start at the beginning of [Learning Puppet](#).
- For a complete but succinct guide to the Puppet language's syntax, see the [Puppet language reference](#).
- For complete documentation of the available resource types, see the [type reference](#).
- For short, printable references, see the [modules cheat sheet](#) and the [core types cheat sheet](#).

Step 2: Use your custom module in the console

Puppet recognizes when you create a custom class, and it can be added to the console and assigned to your Windows nodes.

1. In the console, click **Classification**, and select the node group you want to add your module to (for example, `Windows_example`).
2. On the **Configuration** tab, in the **Add new class** field, enter `critical_policy`.
You might need to wait a moment or two for the class to show up in the list. You can also click Refresh.
3. Click **Add class**, and commit changes.

Step 3: Test out your module

Check to make sure your module does what you expect it to: display the legal caption and text before you log in.

1. On the Windows agent node, manually set the data values of `legalnoticecaption` and `legalnoticetext` to some other values. For example, set `legalnoticecaption` to “Larry’s Computer” and set `legalnoticetext` to “This is Larry’s computer.”
2. On the Windows agent node, refresh the registry and note that the values of `legalnoticecaption` and `legalnoticetext` have been returned to the values specified in your `critical_policy` manifest.
3. Reboot your Windows machine to see the legal caption and text before you log in again. You have created a new class from scratch and used it to manage registry settings on your Windows server.

Step 4: Use a site module

Many users create a "site" module for a type of machine.

Instead of describing smaller units of a configuration, the classes in a site module describe a complete configuration for a given type of machine. For example, a site module might contain:

- A `site::basic` class, for nodes that require security management but haven't been given a specialized role yet.
- A `site::webserver` class for nodes that serve web content.
- A `site::dbserver` class for nodes that provide a database server to other applications.

Site modules hide complexity so you can more easily divide labor at your site. System architects can create the site classes, and junior admins can create new machines and assign a single "role" class to them in the console. In this workflow, the console controls policy, not fine-grained implementation.

1. On the Puppet master, create the `/etc/puppetlabs/code/environments/production/modules/site/manifests/basic.pp` file by running `mkdir -p site/manifests`. Then, edit it to contain the following:

```
class site::basic {
  if $osfamily == 'windows' {
    include critical_policy
  }
  else {
    include motd
    include core_permissions
  }
}
```

2. Run `puppet agent -t` to ensure `site::basic` is created. This class declares other classes with the `include` function.

Note: The "if" conditional sets different classes for different operating systems using the `$osfamily` fact. In this example, if an agent node is not a Windows agent, Puppet applies the `motd` and `core_permissions` classes. For more information about declaring classes, see the modules and classes chapters of Learning Puppet.

3. In the console, remove all of the previous example classes from your nodes and groups (for example, `wsus_client` and `critical_policy`). Be sure to leave the `pe_*` classes in place.
4. Add the `site::basic` class to the console with **Add new class**.
5. Assign the `site::basic` class to the `windows_example` group. Your nodes are now receiving the same configurations as before, but with a simplified interface in the console. Instead of deciding which classes a new node should receive, you can decide what type of node it is and take advantage of decisions you made earlier.

Summary

Writing modules enables you to manage your PE configurations.

In this section, you have performed the core workflows of an intermediate Puppet user.

In the course of their normal work, intermediate users:

- Download and modify Forge modules to fit their deployment's needs.
- Create new modules and write new classes to manage many types of resources, including files, services, packages, user accounts, and more.
- Build and curate a site module to safely empower junior admins and simplify the decisions involved in deploying new machines.
- Monitor and troubleshoot events that affect their infrastructure.

Manage your Windows infrastructure

Learn how to use PE to configure and manage essential parts of your infrastructure.

Using the Windows module pack

This guide covers creating a managed permission with ACL, creating managed registry keys and values with `registry`, and installing and creating your own packages with `chocolatey`.

Windows module pack

The Windows module pack is a group of modules available on the Forge curated to help you complete common Windows tasks.

The Forge is an online community of Puppet modules submitted by Puppet and community members. The Forge makes it easier for you to manage Puppet and can save you time by using pre-written modules, rather than writing your own. In addition to being rated by the community, modules in the Forge can be Puppet Approved or Puppet Supported. The major difference is that Approved modules are not available for Puppet Enterprise support services, but are still tested and adhere to a standard for style and quality.

The Windows module pack includes several Windows compatible modules that help you complete common specific tasks. You can find more Windows modules by searching the Forge. While the module pack itself is not supported, the modules by Puppet contained in the pack are individually supported with PE. The rest have been reviewed and recommended by Puppet but are not eligible for commercial support.

The Windows module pack enables you to do the following:

- Read, create, and write registry keys with `registry`.
- Interact with PowerShell through the Puppet DSL with `powershell`.
- Manage Windows PowerShell DSC (Desired State Configuration) resources using `dsc` and `dsc_lite`.
- Reboot Windows as part of management as necessary through `reboot`.
- Enforce fine-grained access control permissions using `acl`.
- Manage Windows Server Update Service configs on client nodes `wsus_client`.
- Install or remove Windows features with `windowsfeature`.
- Download files for use during management via `download_file`.
- Build IIS sites and virtual applications with `iis`.
- Install packages with `chocolatey`.
- Manage environment variables with `windows_env`.

Install the Windows module pack

These steps show you how to install the module pack locally, but you can also install it on the master and `pluginsync` pushes the module pack to all of your nodes.

1. Open the Puppet command prompt. If you haven't opened the command line interface before, enter `Command Prompt` `Puppet` in your **Start Menu**, and click **Start Command Prompt with Puppet**.
2. To list modules that you currently have installed, enter `puppet module list` in your **Command Prompt** window. If you're just getting started, you likely have no modules installed yet.
3. Next, to install the `puppetlabs/windows` module pack, type `puppet module install puppetlabs/windows`.

Notice that you get a nice output of everything that's installed.

```
C:\>puppet module install puppetlabs/windows
Notice: Preparing to install into
C:/ProgramData/PuppetLabs/code/environments/production/modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
C:/ProgramData/PuppetLabs/code/environments/production/modules
### puppetlabs-windows (v2.1.0)
### chocolatey-chocolatey (v1.2.0)
# ### badgerious-windows_env (v2.2.2)
### puppet-download_file (v1.2.1)
### puppet-iis (v1.4.1)
### puppet-windowsfeature (v1.1.0)
### puppetlabs-acl (v1.1.1)
```

```

### puppetlabs-powershell (v1.0.5)
### puppetlabs-reboot (v1.2.0)
### puppetlabs-registry (v1.1.2)
# ### puppetlabs-stdlib (v4.9.0)
### puppetlabs-wsus_client (v1.0.0)

```

Manage permissions with `acl`

The `puppetlabs-acl` module helps you manage access control lists (ACLs), which provide a way to interact with permissions for the Windows file system. This module enables you to set basic permissions up to very advanced permissions using SIDs (Security Identifiers) with an access mask, inheritance, and propagation strategies. First, start with querying some existing permissions.

View file permissions with `ACL`

ACL is a custom type and provider, so you can use `puppet resource` to look at existing file and folder permissions.

For some types, you can use the command `puppet resource <TYPE NAME>` to get all instances of that type. However, there could be thousands of ACLs on a Windows system, so it's best to specify the folder you want to review the types in. Here, check `c:\Users` to see what permissions it contains.

In the command prompt, enter `puppet resource acl c:\Users`

```

acl { 'c:\Users':
  inherit_parent_permissions => 'false',
  permissions => [
    {identity => 'SYSTEM', rights=> ['full']},
    {identity => 'Administrators', rights => ['full']},
    {identity => 'Users', rights => ['read', 'execute'], affects =>
'self_only'},
    {identity => 'Users', rights => ['read', 'execute'], affects =>
'children_only'},
    {identity => 'Everyone', rights => ['read', 'execute'], affects =>
'self_only'},
    {identity => 'Everyone', rights => ['read', 'execute'], affects =>
'children_only'}
  ],
}

```

As you can see, this particular folder does not inherit permissions from its parent folder; instead, it sets its own permissions and determines how child files and folders inherit the permissions set here.

- `{'identity' => 'SYSTEM', 'rights'=> ['full']}` states that the “SYSTEM” user has full rights to this folder, and by default all children and grandchildren files and folders (as these are the same defaults when creating permissions in Windows).
- `{'identity' => 'Users', 'rights' => ['read', 'execute'], 'affects' => 'self_only'}` gives read and execute permissions to Users but only on the current directory.
- `{'identity' => 'Everyone', 'rights' => ['read', 'execute'], 'affects' => 'children_only'}` gives read and execute permissions to everyone, but only on subfolders and files.

Note: You might see what appears to be the same permission for a user/group twice (both “Users” and “Everyone” above), where one affects only the folder itself and the other is about children only. They are in fact different permissions.

Create a Puppet managed permission

1. Run this code to create your first Puppet managed permission. Then, save it as `perms.pp`

```

file{'c:/temppperms':
  ensure => directory,
}

```

By default, the `acl` creates an implicit relationship to any

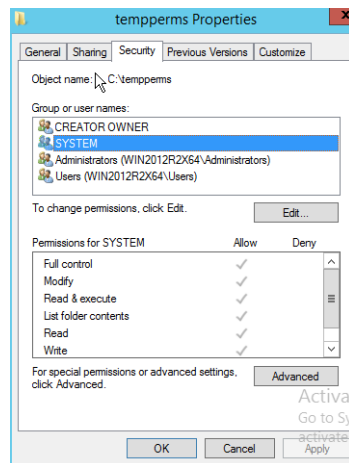
```
# file resources it finds that match the location.
acl {'c:/tempperms':
  permissions => [
    {identity => 'Administrators', rights => ['full']},
    {identity => 'Users', rights => ['read','execute']}
  ],
}
```

2. To validate your manifest, in the command prompt, run `puppet parser validate c:\<FILE PATH>\perms.pp`. If the parser returns nothing, it means validation passed.
3. To apply the manifest, type `puppet apply c:\<FILE PATH>\perms.pp`

Your output should look similar to:

```
Notice: Compiled catalog for win2012r2x64 in environment production in
0.12 seconds
Notice: /Stage[main]/Main/File[c:/tempperms]/ensure: created
Notice: /Stage[main]/Main/Acl[c:/tempperms]/permissions: permissions
changed [
] to [
  { identity => 'BUILTIN\Administrators', rights => ["full"] },
  { identity => 'BUILTIN\Users', rights => ["read", "execute"] }
]
Notice: Applied catalog in 0.05 seconds
```

4. Review the permissions in your Windows UI. In Windows Explorer, right-click **tempperms** and click **Properties**. Then, click the **Security** tab. It should appear similar to the image below.



5. Optional: It might appear that you have more permissions than you were hoping for here. This is because by default Windows inherits parent permissions. In this case, you might not want to do that. Adjust the acl resource to not inherit parent permissions by changing the `perms.pp` file to look like the below by adding `inherit_parent_permissions => false`.

```
acl {'c:/tempperms':
  inherit_parent_permissions => false,
  permissions                => [
    {identity => 'Administrators', rights => ['full']},
    {identity => 'Users', rights => ['read','execute']}
  ],
}
```

6. Save the file, and return the command prompt to run `puppet parser validate c:\<FILE PATH>\perms.pp` again.

7. When it validates, run `puppet apply c:\<FILE PATH>\perms.pp`

You should get output similar to the following:

```
C:\>puppet apply c:\puppet_code\perms.pp
Notice: Compiled catalog for win2012r2x64 in environment production in
0.08 seconds
Notice: /Stage[main]/Main/Acl[c:/temppperms]/inherit_parent_permissions:
inherit_
parent_permissions changed 'true' to 'false'
Notice: Applied catalog in 0.02 seconds
```

8. To check the permissions again, enter `icacls c:\temppperms` in the command prompt. The command, `icacls`, is specifically for displaying and modifying ACLs. The output should be similar to the following:

```
C:\>icacls c:\temppperms
c:\temppperms BUILTIN\Administrators:(OI)(CI)(F)
               BUILTIN\Users:(OI)(CI)(RX)
               NT AUTHORITY\SYSTEM:(OI)(CI)(F)
               BUILTIN\Users:(CI)(AD)
               CREATOR OWNER:(OI)(CI)(IO)(F)
Successfully processed 1 files; Failed processing 0 files
```

The output shows each permission, followed by a list of specific rights in parentheses. This output shows there are more permissions than you specified in `perms.pp`. Puppet manages permissions next to unmanaged or existing permissions. In the case of removing inheritance, by default Windows copies those existing inherited permissions (or Access Control Entries, ACEs) over to the existing ACL so you have some more permissions that you might not want.

9. Remove the extra permissions, so that only the permissions you've specified are on the folder. To do this, in your `perms.pp` set `purge => true` as follows:

```
acl {'c:/temppperms':
  inherit_parent_permissions => false,
  purge                     => true,
  permissions               => [
    {identity => 'Administrators', rights => ['full']},
    {identity => 'Users', rights => ['read', 'execute']}
  ],
}
```

10. Run the parser command as you have before. If it still returns no errors, then you can apply the change.
11. To apply the change, run `puppet apply c:\<FILE PATH>\perms.pp`. The output should be similar to below:

```
C:\>puppet apply c:\puppet_code\perms.pp
Notice: Compiled catalog for win2012r2x64 in environment production in
0.08 seconds
Notice: /Stage[main]/Main/Acl[c:/temppperms]/permissions: permissions
changed [
  { identity => 'BUILTIN\Administrators', rights => ["full"] },
  { identity => 'BUILTIN\Users', rights => ["read", "execute"] },
  { identity => 'NT AUTHORITY\SYSTEM', rights => ["full"] },
  { identity => 'BUILTIN\Users', rights => ["mask_specific"], mask => '4',
    child_types => 'containers' },
  { identity => 'CREATOR OWNER', rights => ["full"], affects =>
    'children_only' }
] to [
  { identity => 'BUILTIN\Administrators', rights => ["full"] },
  { identity => 'BUILTIN\Users', rights => ["read", "execute"] }
]
```

Notice: Applied catalog in 0.05 seconds

Puppet outputs a notice as it is removing each of the permissions.

12. Take a look at the output of `icacls` again with `icacls c:\temppperms`

```
c:\>icacls c:\temppperms
c:\temppperms BUILTIN\Administrators:(OI)(CI)(F)
               BUILTIN\Users:(OI)(CI)(RX)
Successfully processed 1 files; Failed processing 0 files
```

Now the permissions have been set up for this directory. You can get into more advanced permission scenarios if you read the usage scenarios on this module's Forge page.

Create Puppet managed registry keys with `registry`

You might eventually need to use the registry to access and set highly available settings, among other things. The `puppetlabs-registry` module, which is also a Puppet Supported Module enables you to set both registry keys and values.

View registry keys and values with `puppet resource`

`puppetlabs-registry` is a custom type and provider, so you can use `puppet resource` to look at existing registry settings.

It is also somewhat limited, like the `acl` module in that it is restricted to only what is specified.

1. In your command prompt, run: `puppet resource registry_key 'HKLM\Software\Microsoft\Windows'`

```
C:\>puppet resource registry_key 'HKLM\Software\Microsoft\Windows\'
registry_key { 'HKLM\Software\Microsoft\Windows\':
  ensure => 'present',
}
```

Not that interesting, but now take a look at a registry value.

2. Enter `puppet resource registry_value 'HKLM\SYSTEM\CurrentControlSet\Services\BITS\DisplayName'`

```
registry_value { 'HKLM\SYSTEM\CurrentControlSet\Services\BITS\
\DisplayName':
  ensure => 'present',
  data   => ['Background Intelligent Transfer Service'],
  type   => 'string',
}
```

That's a bit more interesting than a registry key.

Keys are like file paths (directories) and values are like files that can have data and be of different types.

Create managed keys

Learn how to make managed registry keys, and see Puppet correct configuration drift when you try and alter them in Registry Editor.

1. Create your first Puppet managed registry keys and values:

```
registry_key { 'HKLM\Software\zTemporaryPuppet':
  ensure => present,
}

# By default the registry creates an implicit relationship to any file
# resources it finds that match the location.
registry_value { 'HKLM\Software\zTemporaryPuppet\StringValue':
  ensure => 'present',
  data   => 'This is a custom value.',
  type   => 'string',
}
```

```

}

#forcing a 32-bit registry view; watch where this is created:
registry_key { '32:HKLM\Software\zTemporaryPuppet':
  ensure => present,
}

registry_value { '32:HKLM\Software\zTemporaryPuppet\StringValue':
  ensure => 'present',
  data   => 'This is a custom 32-bit value.',
  type   => 'expand',
}

```

2. Save the file as `registry.pp`.

3. Validate the manifest. In the command prompt, run `puppet parser validate c:\<FILE PATH>\registry.pp`

If the parser returns nothing, it means validation passed.

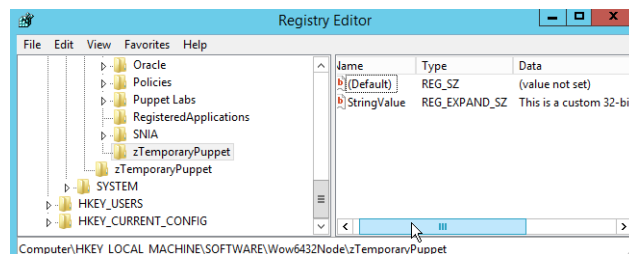
4. Now, apply the manifest by running `puppet apply c:\<FILE PATH>\registry.pp` in the command prompt. Your output should look similar to below.

```

Notice: Compiled catalog for win2012r2x64 in environment production in
0.11 seco
nds
Notice: /Stage[main]/Main/Registry_key[HKLM\Software\zTemporaryPuppet]/
ensure: c
reated
Notice: /Stage[main]/Main/Registry_value[HKLM\Software\zTemporaryPuppet
\StringVa
lue]/ensure: created
Notice: /Stage[main]/Main/Registry_key[ 32:HKLM\Software\zTemporaryPuppet ]/
ensure
: created
Notice: /Stage[main]/Main/Registry_value[ 32:HKLM\Software\zTemporaryPuppet
\Strin
gValue]/ensure: created
Notice: Applied catalog in 0.03 seconds

```

5. Next, inspect the registry and see what you have. Press **Start + R**, then type `regedit` and press **Enter**. Once the **Registry Editor** opens, find your keys under **HKEY_LOCAL_MACHINE**.



Note that the 32-bit keys were created under the 32-bit section of Wow6432Node for Software.

6. Apply the manifest again by running `puppet apply c:\<FILE PATH>\registry.pp`

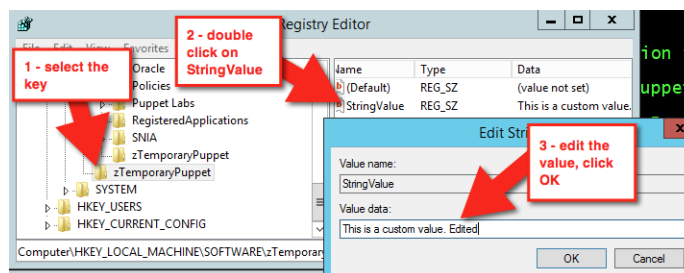
```

Notice: Compiled catalog for win2012r2x64 in environment production in
0.11 seconds
Notice: Applied catalog in 0.02 seconds

```

Nothing changed, so there is no work for Puppet to do.

7. In **Registry Editor**, change the data. Select **HKLM\Software\zTemporaryPuppet** and in the right box, double-click **StringValue**. Edit the value data, and click **OK**.



This time, changes have been made, so running `puppet apply c:\path\to\registry.pp` results in a different output.

```
Notice: Compiled catalog for win2012r2x64 in environment production
in 0.11 seconds
Notice: /Stage[main]/Main/Registry_value[HKLM\Software\zTemporaryPuppet
\StringValue]/data:
data changed 'This is a custom value. Edited' to 'This is a custom value.'
Notice: Applied catalog in 0.03 seconds
```

Puppet automatically corrects the configuration drift.

8. Next, clean up and remove the keys and values. Make your `registry.pp` file look like the below:

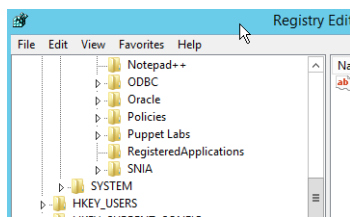
```
registry_key { 'HKLM\Software\zTemporaryPuppet':
  ensure => absent,
}

#forcing a 32 bit registry view, watch where this is created
registry_key { '32:HKLM\Software\zTemporaryPuppet':
  ensure => absent,
}
```

9. Validate it with `puppet parser validate c:\path\to\registry.pp` and apply it again with `puppet apply c:\path\to\registry.pp`

```
Notice: Compiled catalog for win2012r2x64 in environment production in
0.06 seconds
Notice: /Stage[main]/Main/Registry_key[HKLM\Software\zTemporaryPuppet]/
ensure: removed
Notice: /Stage[main]/Main/Registry_key[32:HKLM\Software\zTemporaryPuppet]/
ensure
: removed
Notice: Applied catalog in 0.02 seconds
```

Refresh the view in your **Registry Editor**. The values are gone.



Example

Here's a real world example that disables error reporting:

```
class puppetconf::disable_error_reporting {
```

```

    registry_value { 'HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Windows
Error Reporting\ForceQueue':
      type => dword,
      data => '1',
    }

    registry_value { 'HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Windows
Error Reporting\DontShowUI':
      type => dword,
      data => '1',
    }

    registry_value { 'HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Windows
Error Reporting\DontSendAdditionalData':
      type => dword,
      data => '1',
    }

    registry_key { 'HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Windows Error
Reporting\Consent':
      ensure => present,
    }

    registry_value { 'HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Windows
Error Reporting\Consent\DefaultConsent':
      type => dword,
      data => '2',
    }
  }
}

```

Create, install and repackaging packages with the `chocolatey` module

Chocolatey is a package manager for Windows that is similar in design and execution to package managers on non-Windows systems. The `chocolatey` module is a Puppet Approved Module, so it's not eligible for Puppet Enterprise support services. The module has the capability to install and configure Chocolatey itself, and then manage software on Windows with Chocolatey packages.

View existing packages

Chocolatey has a custom provider for the package resource type, so you can use `puppet resource` to view existing packages.

In the command prompt, run `puppet resource package --param provider | more`

The additional provider parameter in this command outputs all types of installed packages that are detected by multiple providers.

Install Chocolatey

These steps are to install Chocolatey (`choco.exe`) itself. You use the module to ensure Chocolatey is installed.

1. Create a new manifest in the `chocolatey` module called `chocolatey.pp` with the following contents:

```
include chocolatey
```

2. Validate the manifest by running `puppet parser validate c:\<FILE PATH>\chocolatey.pp` in the command prompt. If the parser returns nothing, it means validation passed.
3. Apply the manifest by running `puppet apply c:\<FILE PATH>\chocolatey.pp`

Your output should look similar to below:

```

Notice: Compiled catalog for win2012r2x64 in environment production in
0.58 seconds
Notice: /Stage[main]/Chocolatey::Install/Windows_env[chocolatey_PATH_env]/
ensure
: created

```



```

Notice: /Stage[main]/Chocolatey::Install/
Windows_env[chocolatey_ChocolateyInstal
l_env]/ensure: created
Notice: /Stage[main]/Chocolatey::Install/
Exec[install_chocolatey_official]/retur
ns: executed successfully
Notice: /Stage[main]/Chocolatey::Install/
Exec[install_chocolatey_official]: Trig
gered 'refresh' from 2 events
Notice: Finished catalog run in 13.22 seconds

```

In a production scenario, you're likely to have a Chocolatey.nupkg file somewhere internal. In cases like that, you can use the internal nupkg for Chocolatey installation:

```

class {'chocolatey':
  chocolatey_download_url => 'https://internalurl/to/chocolatey.nupkg',
  use_7zip                 => false,
  log_output              => true,
}

```

Install a package with chocolatey

Normally, when installing packages you copy them locally first, make any required changes to bring everything they download to an internal location, repackage the package with the edits, and build your own packages to host on your internal package repository (feed). For this exercise, however, you directly install a portable Notepad++ from Chocolatey's community feed. The Notepad++ CommandLine package is portable and shouldn't greatly affect an existing system.

1. Update the manifest chocolatey.pp with the following contents:

```

package {'notepadplusplus.commandline':
  ensure => installed,
  provider => chocolatey,
}

```

2. Validate the manifest by running `puppet parser validate c:\<FILE PATH>\chocolatey.pp` in the command prompt. If the parser returns nothing, it means validation passed.
3. Now, apply the manifest with `puppet apply c:\<FILE PATH>\chocolatey.pp`. Your output should look similar to below.

```

Notice: Compiled catalog for win2012r2x64 in environment production in
0.75 seconds
Notice: /Stage[main]/Main/Package[notepadplusplus.commandline]/ensure:
created
Notice: Applied catalog in 15.51 seconds

```

If you want to use this package for a production scenario, you need an internal custom feed. This is simple to set up with the `chocolatey_server` module. You could also use Sonatype Nexus, Artifactory, or a CIFS share if you want to host packages with a non-Windows option, or you can use anything on Windows that exposes a NuGet OData feed (Nuget is the packaging infrastructure that Chocolatey uses). See the [How To Host Feed page of the chocolatey wiki](#) for more in-depth information. You could also store packages on your master and use a file resource to verify they are in a specific local directory prior to ensuring the packages.

Example

The following example ensures that Chocolatey, the Chocolatey Simple Server (an internal Chocolatey package repository), and some packages are installed. It requires the additional [chocolatey/chocolatey_server module](#).

In `c:\<FILE_PATH>\packages` you must have packages for [Chocolatey](#), [Chocolatey.Server](#), [RoundhouseE](#), [Launchy](#), and [Git](#), as well as any of their dependencies for this to work.

```
case $operatingsystem {
  'windows': {
    Package {
      provider => chocolatey,
      source   => 'C:\packages',
    }
  }
}

# include chocolatey
class {'chocolatey':
  chocolatey_download_url => 'file:///C:/packages/
chocolatey.0.9.9.11.nupkg',
  use_7zip                 => false,
  log_output               => true,
}

# This contains the bits to install the custom server.
# include chocolatey_server
class {'chocolatey_server':
  server_package_source => 'C:/packages',
}

package {'roundhouse':
  ensure => '0.8.5.0',
}

package {'launchy':
  ensure           => installed,
  install_options => ['-override', '-installArgs', '', '/VERYSILENT', '/
NORESTART', ''],
}

package {'git':
  ensure => latest,
}
```

Copy an existing package and make it internal (repackaging packages)

To make the existing package local, use these steps.

Chocolatey's community feed has quite a few packages, but they are geared towards community and use the internet for downloading from official distribution sites. However, they are attractive as they have everything necessary to install a piece of software on your machine. Through the repackaging process, by which you take a community package and bring all of the bits internal or embed them into the package, you can completely internalize a package to host on an internal Chocolatey/NuGet repository. This gives you complete control over a package and removes the aforementioned production trust and control issues.

1. Download the Notepad++ package from Chocolatey's community feed by going to the package page and clicking the download link.
2. Rename the downloaded file to end with `.zip` and unpack the file as a regular archive.
3. Delete the `_rels` and `package` folders and the `[Content_Types].xml` file. These are created during `choco pack` and should not be included, because they're regenerated (and their existence leads to issues).

```
notepadplusplus.commandline.6.8.7.nupkg
####_rels    # DELETE
####package # DELETE
```

```
# #####services
####tools
### [Content_Types].xml # DELETE
### notepadplusplus.commandline.nuspec
```

4. Open `tools\chocolateyInstall.ps1`.

```
Install-ChocolateyZipPackage 'notepadplusplus.commandline' 'https://
notepad-plus-plus.org/repository/6.x/6.8.7/npp.6.8.7.bin.zip' "$(Split-
Path -parent $MyInvocation.MyCommand.Definition)"
```

5. Download the zip file and place it in the tools folder of the package.

6. Next, edit `chocolateyInstall.ps1` to point to this embedded file instead of reaching out to the internet (if the size of the file is over 50MB, you might want to put it on a file share somewhere internally for better performance).

```
$toolsDir = "$(Split-Path -parent $MyInvocation.MyCommand.Definition)"
Install-ChocolateyZipPackage 'notepadplusplus.commandline' "$toolsDir
\npp.6.8.7.bin.zip" "$toolsDir"
```

The double quotes allow for string interpolation (meaning variables get interpreted instead of taken literally).

7. Next, open the `*.nuspec` file to view its contents and make any necessary changes.

```
<?xml version="1.0"?>
<package xmlns="http://schemas.microsoft.com/packaging/2010/07/
nuspec.xsd">
  <metadata>
    <id>notepadplusplus.commandline</id>
    <version>6.8.7</version>
    <title>Notepad++ (Portable, CommandLine)</title>
    <authors>Don Ho</authors>
    <owners>Rob Reynolds</owners>
    <projectUrl>https://notepad-plus-plus.org/</projectUrl>
    <iconUrl>https://cdn.rawgit.com/ferrentcoder/chocolatey-
packages/02c21bebe5abb495a56747cbb9b4b5415c933fc0/icons/
notepadplusplus.png</iconUrl>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>Notepad++ is a ... </description>
    <summary>Notepad++ is a free (as in "free speech" and also as in "free
beer") source code editor and Notepad replacement that supports several
languages. </summary>
    <tags>notepad notepadplusplus notepad-plus-plus</tags>
  </metadata>
</package>
```

Some organizations change the version field to denote that this is an edited internal package, for example changing 6.8.7 to 6.8.7.20151202. For now, this is not necessary.

8. Now you can navigate via the command prompt to the folder with the `.nuspec` file (from a Windows machine unless you've installed Mono and built `choco.exe` from source) and use `choco pack`. You can also be more specific and run `choco pack <FILE PATH>\notepadplusplus.commandline.nuspec`. The output should be similar to below.

```
Attempting to build package from 'notepadplusplus.commandline.nuspec'.
Successfully created package 'notepadplusplus.commandline.6.8.7.nupkg'
```

Normally you test on a system to ensure that the package you just built is good prior to pushing the package (just the `*.nupkg`) to your internal repository. This can be done by using `choco.exe` on a test system to install (`choco install notepadplusplus.commandline -source %cd% -change %cd% to $pwd` in PowerShell.exe) and uninstall (`choco uninstall notepadplusplus.commandline`). Another method of testing is to run the manifest pointed to a local source folder, which is what you are going to do.

9. Create `c:\packages` and copy the resulting package file (`notepadplusplus.commandline.6.8.7.nupkg`) into it.

This won't actually install on this system since you just installed the same version from Chocolatey's community feed. So you need to remove the existing package first. To remove it, edit your `chocolatey.pp` to set the package to absent.

```
package {'notepadplusplus.commandline':
  ensure => absent,
  provider => chocolatey,
}
```

10. Validate the manifest with `puppet parser validate path\to\chocolatey.pp`. Apply the manifest to ensure the change `puppet apply c:\path\to\chocolatey.pp`.

You can validate that the package has been removed by checking for it in the package install location or by using `choco list -lo`.

11. Update the manifest (`chocolatey.pp`) to use the custom package.

```
package {'notepadplusplus.commandline':
  ensure => latest,
  provider => chocolatey,
  source => 'c:\packages',
}
```

12. Validate the manifest with the parser and then apply it again. You can see Puppet creating the new install in the output.

```
Notice: Compiled catalog for win2012r2x64 in environment production in
0.79 seconds
Notice: /Stage[main]/Main/Package[notepadplusplus.commandline]/ensure:
created
Notice: Applied catalog in 14.78 seconds
```

13. In an earlier step, you added a `*.zip` file to the package, so that you can inspect it and be sure the custom package was installed. Navigate to `C:\ProgramData\chocolatey\lib\notepadplusplus.commandline\tools` (if you have a default install location for Chocolatey) and see if you can find the `*.zip` file.

You can also validate the `chocolateyInstall.ps1` by opening and viewing it to see that it is the custom file you changed.

Create a package with chocolatey

Creating your own packages is, for some system administrators, surprisingly simple compared to other packaging standards.

Ensure you have at least Chocolatey CLI (`choco.exe`) version `0.9.9.11` or newer for this next part.

1. From the command prompt, enter `choco new -h` to see a help menu of what the available options are.
2. Next, use `choco new vagrant` to create a package named 'vagrant'. The output should be similar to the following:

```
Creating a new package specification at C:\temppackages\vagrant
Generating template to a file
  at 'C:\temppackages\vagrant\vagrant.nuspec'
Generating template to a file
  at 'C:\temppackages\vagrant\tools\chocolateyinstall.ps1'
Generating template to a file
  at 'C:\temppackages\vagrant\tools\chocolateyuninstall.ps1'
Generating template to a file
  at 'C:\temppackages\vagrant\tools\ReadMe.md'
Successfully generated vagrant package specification files
```

```
at 'C:\temppackages\vagrant'
```

It comes with some files already templated for you to fill out (you can also create your own custom templates for later use).

3. Open `vagrant.nuspec`, and edit it to look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<package xmlns="http://schemas.microsoft.com/packaging/2015/06/
nuspec.xsd">
  <metadata>
    <id>vagrant</id>
    <title>Vagrant (Install)</title>
    <version>1.8.4</version>
    <authors>HashiCorp</authors>
    <owners>my company</owners>
    <description>Vagrant - Development environments made easy.</
description>
  </metadata>
  <files>
    <file src="tools\**" target="tools" />
  </files>
</package>
```

Unless you are sharing with the world, you don't need most of what is in the nuspec template file, so only required items are included above. Match the version of the package in this nuspec file to the version of the underlying software as closely as possible. In this example, Vagrant 1.8.4 is being packaged.

4. Open `chocolateyInstall.ps1` and edit it to look like the following:

```
$ErrorActionPreference = 'Stop';

$packageName= 'vagrant'
$toolsDir    = "$(Split-Path -parent $MyInvocation.MyCommand.Definition)"
$fileLocation = Join-Path $toolsDir 'vagrant_1.8.4.msi'

$packageArgs = @{
  packageName   = $packageName
  fileType      = 'msi'
  file          = $fileLocation

  silentArgs    = "/qn /norestart"
  validExitCodes= @(0, 3010, 1641)
}

Install-ChocolateyInstallPackage @packageArgs
```

Note: The above is `Install-ChocolateyINSTALLPackage`, not to be confused with `Install-ChocolateyPackage`. The names are very close to each other, however the latter also downloads software from a URI (URL, ftp, file) which is not necessary for this example.

5. Delete the `ReadMe.md` and `chocolateyUninstall.ps1` files. [Download Vagrant](#) and move it to the tools folder of the package.

Note: Normally if a package is over 100MB, it is recommended to move the software installer/archive to a share drive and point to it instead. For this example, just bundle it as is.

6. Now pack it up by using `choco pack`. Copy the new `vagrant.1.8.4.nupkg` file to `c:\packages`.
7. Open the manifest, and add the new package you just created. Your `chocolatey.pp` file should look like the below.

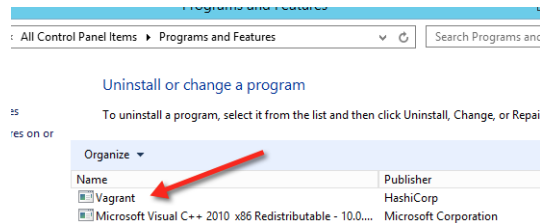
```
package {'vagrant':
  ensure => installed,
  provider => chocolatey,
```

```

    source => 'c:\packages',
  }

```

8. Save the file and make sure to validate with the Puppet parser.
9. Use `puppet apply <FILE PATH>\chocolatey.pp` to run the manifest.
10. Open **Control Panel, Programs and Features** and take a look.



Vagrant is installed!

Uninstall packages with Chocolatey

In addition to installing and creating packages, Chocolatey can also help you uninstall them.

To verify that the `choco autoUninstaller` feature is turned on, use `choco feature` to list the features and their current state. If you're using `include chocolatey` or `class chocolatey` to ensure Chocolatey is installed, the configuration is applied automatically (unless you have explicitly disabled it). Starting in Chocolatey version 0.9.10, it is enabled by default.

1. If you see `autoUninstaller - [Disabled]`, you need to enable it. To do this, in the command prompt, run `choco feature enable -n autoUninstaller`. You should see a similar success message:

You should see a similar success message:

```
Enabled autoUninstaller
```

2. To remove Vagrant, edit your `chocolatey.pp` manifest to ensure `=> absent`. Then save and validate the file.

```

package {'vagrant':
  ensure => absent,
  provider => chocolatey,
  source => 'c:\packages',
}

```

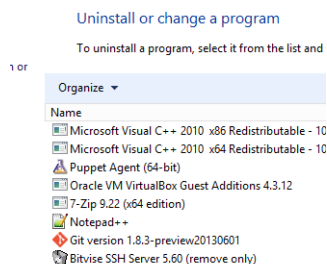
3. Next, run `puppet apply <FILE PATH>\chocolatey.pp` to apply the manifest.

```

Notice: Compiled catalog for win2012r2x64 in environment production in
0.75 seconds
Notice: /Stage[main]/Main/Package[vagrant]/ensure: removed
Notice: Applied catalog in 40.85 seconds

```

You can look in the Control Panel, Programs and Features to see that it's no longer installed!



Managing Windows configurations

This page covers the different ways you can use Puppet Enterprise (PE) to manage your Windows configurations, including creating local group and user accounts.

Basic tasks and concepts in Windows

This section is meant to help familiarize you with several common tasks used in Puppet Enterprise (PE) with Windows agents, and explain the concepts and reasons behind performing them.

Practice tasks

In other locations in the documentation, these can be found as steps in tasks, but they are not explained as thoroughly.

Write a simple manifest

Puppet manifest files are lists of resources that have a unique title and a set of named attributes that describe the desired state.

Before you begin

You need a text editor, for example Visual Studio Code (VS Code), to create manifest files. Puppet has an [extension](#) for VS Code that supports syntax highlighting of the Puppet language. Editors like Notepad++ or Notepad won't highlight Puppet syntax, but can also be used to create manifests.

Manifest files are written in Puppet code, a domain specific language (DSL) that defines the desired state of resources on a system, such as files, users, and packages. Puppet compiles these text-based manifests into catalogs, and uses those to apply configuration changes.

1. Create a file named `file.pp` and save it in `c:\myfiles\`
2. With your text editor of choice, add the following text to the file:

```
file { 'c:\\Temp\\foo.txt':
  ensure => present,
  content => 'This is some text in my file'
}
```

Note the following details in this file resource example:

- Puppet uses a basic syntax of `type { title: }`, where `type` is the resource type — in this case it's `file`.
- The resource title (the value before the `:`) is `C:\\Temp\\foo.txt`. The file resource uses the title to determine where to create the file on disk. A resource title must always be unique within a given manifest.
- The `ensure` parameter is set to `present` to create the file on disk, if it's not already present. For `file` type resources, you can also use the value `absent`, which removes the file from disk if it exists.
- The `content` parameter is set to `This is some text in my file`, which writes that value to the file.

Validate your manifest with puppet parser validate

You can validate that a manifest's syntax is correct by using the command `puppet parser validate`

1. Check your syntax by entering `puppet parser validate c:\myfiles\file.pp` in the Puppet command prompt. If a manifest has no syntax errors, the tool outputs nothing.
2. To see what output occurs when there is an error, temporarily edit the manifest and remove the `:` after the resource title. Run `puppet parser validate c:\myfiles\file.pp` again, and see the following output:

```
Error: Could not parse for environment production: Syntax error at
'ensure' at c:/myfiles/file.pp:2:3
```

Launch the Puppet command prompt

A lot of common interactions with Puppet are done via the command line.

To open the command line interface, enter `Command Prompt Puppet` in your **Start Menu**, and click **Start Command Prompt with Puppet**.

The Puppet command prompt has a few details worth noting:

- Several important batch files live in the current working directory, `C:\Program Files\Puppet Labs\Puppet\bin`. The most important of these batch files is `puppet.bat`. Puppet is a Ruby based application, and `puppet.bat` is a wrapper around executing Puppet code through `ruby.exe`.
- Running the command prompt with Puppet rather than just the default Windows command prompt ensures that all of the Puppet tooling is in `PATH`, even if you change to a different directory.

Simulate a Puppet run with --noop

Puppet has a switch that you can use to test if manifests make the intended changes. This is referred to as non-enforcement or no-op mode.

To simulate changes, run `puppet apply c:\myfiles\file.pp --noop` in the command prompt:

```
C:\Program Files\Puppet Labs\Puppet\bin>puppet apply c:\myfiles\file.pp --
noop
Notice: Compiled catalog for win-User.localdomain in environment production
in 0.45 seconds
Notice: /Stage[main]/MainFile[C:\Temp\foo.txt]/ensure: current value absent,
should be present (noop)
Notice: Class[Main]: Would have triggered 'refresh' from 1 events
Notice: Stage[main]: Would have triggered 'refresh' from 1 events
Notice: Applied catalog in 0.03 seconds
```

Puppet shows you the changes it *would* make, but does not actually make the changes. It *would* create a new file at `C:\Temp\foo.txt`, but it hasn't, because you used `--noop`.

Enforce the desired state with puppet apply

When the output of the simulation shows the changes you intend to make, you can start enforcing these changes with the `puppet apply` command.

Run `puppet apply c:\myfiles\file.pp`.

To see more details about what this command did, you can specify additional options, such as `--trace`, `--debug`, or `--verbose`, which can help you diagnose problematic code. If `puppet apply` fails, Puppet outputs a full stack trace.

Puppet enforces the resource state you've described in `file.pp`, in this case guaranteeing that a file (`c:\Temp\foo.txt`) is present and has the contents `This is some text in my file`.

Understanding idempotency

A key feature of Puppet is its *idempotency*: the ability to repeatedly apply a manifest to guarantee a desired resource state on a system, with the same results every time.

If a given resource is already in the desired state, Puppet performs no actions. If a given resource is not in the desired state, Puppet takes whatever action is necessary to put the resource into the desired state. Idempotency enables Puppet to simulate resource changes without performing them, and lets you set up configuration management one time, fixing configuration drift without recreating resources from scratch each time Puppet runs.

To demonstrate how Puppet can be applied repeatedly to get the same results, change the manifest at `c:\myfiles\file.pp` to the following:

```
file { 'C:\\Temp\\foo.txt':
  ensure => present,
  content => 'I have changed my file content.'
}
```

Apply the manifest by running `puppet apply c:\myfiles\file.pp`. Open `c:\Temp\foo.txt` and notice that Puppet changes the file's contents.

Applying the manifest again with `puppet apply c:\myfiles\file.pp` results in no changes to the system, demonstrating that Puppet behaves idempotently.

Many of the samples in Puppet documentation assume that you have this basic understanding of creating and editing manifest files, and applying them with `puppet apply`.

Additional command line tools

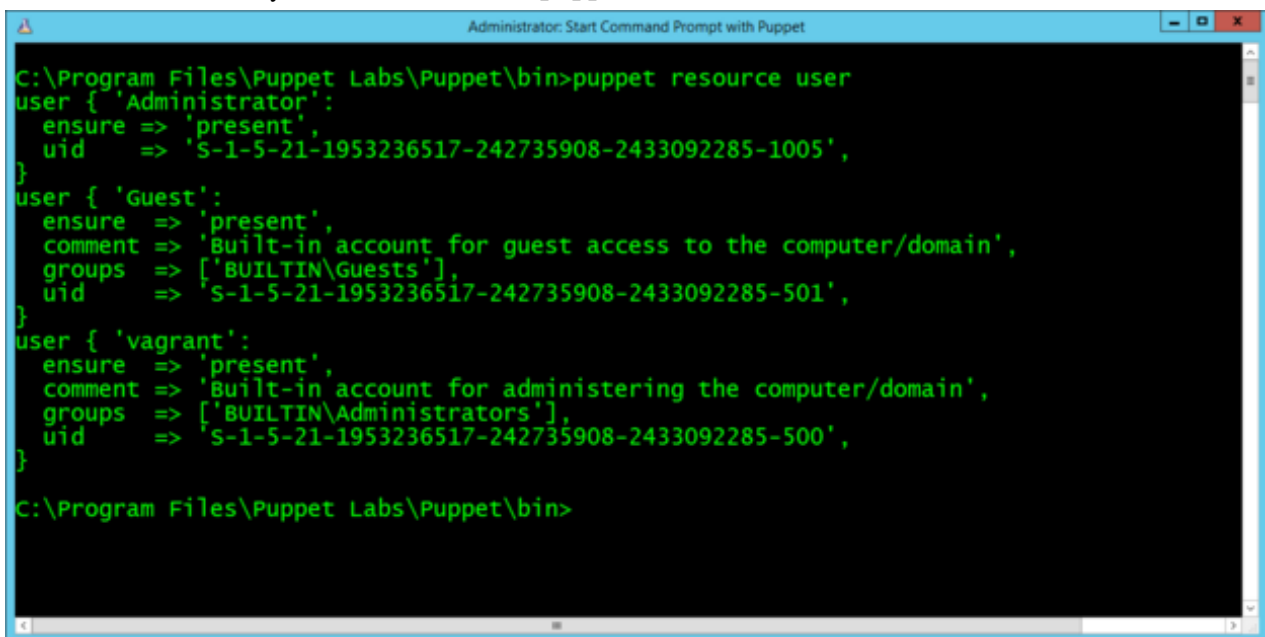
Once you understand how to write manifests, validate them, and use `puppet apply` to enforce your changes, you're ready to use commands such as `puppet agent`, `puppet resource`, and `puppet module install`.

puppet agent

Like `puppet apply`, the `puppet agent` command line tool applies configuration changes to a system. However, `puppet agent` retrieves compiled catalogs from a Puppet Server, and applies them to the local system. Puppet is installed as a Windows service, and by default tries to contact the master every 30 minutes by running `puppet agent` to retrieve new catalogs and apply them locally.

puppet resource

You can run `puppet resource` to query the state of a particular type of resource on the system. For example, to list all of the users on a system, run the command `puppet resource user`.



```
C:\Program Files\Puppet Labs\Puppet\bin>puppet resource user
user { 'Administrator':
  ensure => 'present',
  uid    => 'S-1-5-21-1953236517-242735908-2433092285-1005',
}
user { 'Guest':
  ensure => 'present',
  comment => 'Built-in account for guest access to the computer/domain',
  groups  => ['BUILTIN\Guests'],
  uid     => 'S-1-5-21-1953236517-242735908-2433092285-501',
}
user { 'vagrant':
  ensure => 'present',
  comment => 'Built-in account for administering the computer/domain',
  groups  => ['BUILTIN\Administrators'],
  uid     => 'S-1-5-21-1953236517-242735908-2433092285-500',
}
C:\Program Files\Puppet Labs\Puppet\bin>
```

The computer used for this example has three local user accounts: Administrator, Guest, and vagrant. Note that the output is the same format as a manifest, and you can copy and paste it directly into a manifest.

puppet module install

Puppet includes many core resource types, plus you can extend Puppet by installing modules. Modules contain additional resource definitions and the code necessary to modify a system to create, read, modify, or delete those resources. The Puppet Forge contains modules developed by Puppet and community members available for anyone to use.

Puppet synchronizes modules from a master to agent nodes during `puppet agent` runs. Alternatively, you can use the standalone Puppet module tool, included when you install Puppet, to manage, view, and test modules.

Run `puppet module list` to show the list of modules installed on the system.

To install modules, the Puppet module tool uses the syntax `puppet module install NAMESPACE/MODULENAME`. The `NAMESPACE` is registered to a module, and `MODULE` refers to the specific module name. A very common module to install on Windows is `registry`, under the `puppetlabs` namespace. So, to install the `registry` module, run `puppet module install puppetlabs/registry`.

Manage Windows services

You can use Puppet to manage Windows services, specifically, to start, stop, enable, disable, list, query, and configure services. This way, you can ensure that certain services are always running or are disabled as necessary.

You write Puppet code to manage services in the manifest. When you apply the manifest, the changes you make to the service are applied.

Note: In addition to using manifests to apply configuration changes, you can query system state using the `puppet resource` command, which emits code as well as applying changes.

Ensure a Windows service is running

There are often services that you always want running in your infrastructure.

To have Puppet ensure that a service is running, use the following code:

```
service { '<service name>':
  ensure => 'running'
}
```

Example

For example, the following manifest code ensures the Windows Time service is running:

```
service { 'w32time':
  ensure => 'running'
}
```

Stop a Windows service

Some services can impair performance, or might need to be stopped for regular maintenance.

To disable a service, use the code:

```
service { '<service name>':
  ensure => 'stopped',
  enable => 'false'
}
```

Example

For example, this disables the disk defragmentation service, which can negatively impact service performance.

```
service { 'defragsvc':
  ensure => 'stopped',
  enable => 'false'
}
```

Schedule a recurring task

Regularly scheduled tasks are often necessary on Windows to perform routine system maintenance.

If you need to sync files from another system on the network, perform backups to another disk, or execute log or index maintenance on SQL Server, you can use Puppet to schedule and perform regular tasks. The following shows how to regularly delete files.

To delete all files recursively from `C:\Windows\Temp` at 8 AM each day, create a resource called `scheduled_task` with these attributes:

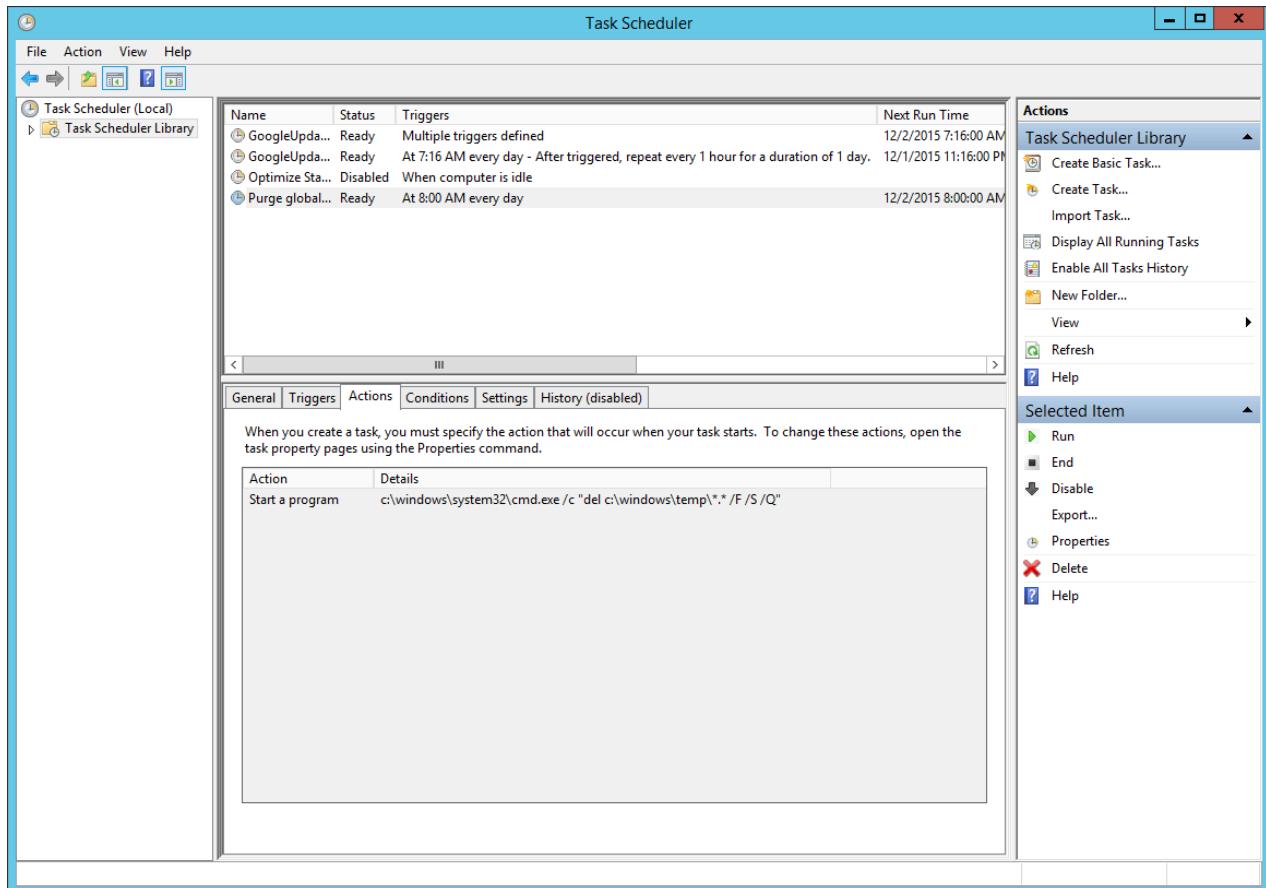
```
scheduled_task { 'Purge global temp files':
  ensure    => present,
  enabled   => true,
  command   => 'c:\\windows\\system32\\cmd.exe',
  arguments => '/c "del c:\\windows\\temp\\*. * /F /S /Q"',
  trigger   => {
    schedule => daily,
```

```

    start_time => '08:00',
  }
}

```

After you set up Puppet to manage this task, the Task Scheduler includes the task you specified:



Example

In addition to creating a trivial daily task at a specified time, the scheduled task resource supports a number of other more advanced scheduling capabilities, including more fine-tuned scheduling. For example, to change the above task to instead perform a disk clean-up every 2 hours, modify the trigger definition:

```

scheduled_task { 'Purge global temp files every 2 hours':
  ensure => present,
  enabled => true,
  command => 'c:\\windows\\system32\\cmd.exe',
  arguments => '/c "del c:\\windows\\temp\\*. * /F /S /Q"',
  trigger => [{
    day_of_week => ['mon', 'tues', 'wed', 'thurs', 'fri'],
    every => '1',
    minutes_interval => '120',
    minutes_duration => '1440',
    schedule => 'weekly',
    start_time => '07:30'
  }],
  user => 'system',
}

```

You can see the corresponding definition reflected in the Task Scheduler GUI:

Manage Windows users and groups

Puppet can be used to create local group and user accounts. Local user accounts are often desirable for isolating applications requiring unique permissions.

Manage administrator accounts

It is often necessary to standardize the local Windows Administrator password across an entire Windows deployment.

To manage administrator accounts with Puppet, create a user resource with 'Administrator' as the resource title like so:

```
user { 'Administrator':
  ensure => present,
  password => 'yabbadabba'
}
```

Note: Securing the password used in the manifest is beyond the scope of this introductory example, but it's common to use Hiera, a key/value lookup tool for configuration, with eyaml to solve this problem. Not only does this solution provide secure storage for the password value, but it also provides parameterization to support reuse, opening the door to easy password rotation policies across an entire network of Windows machines.

Configure an app to use a different account

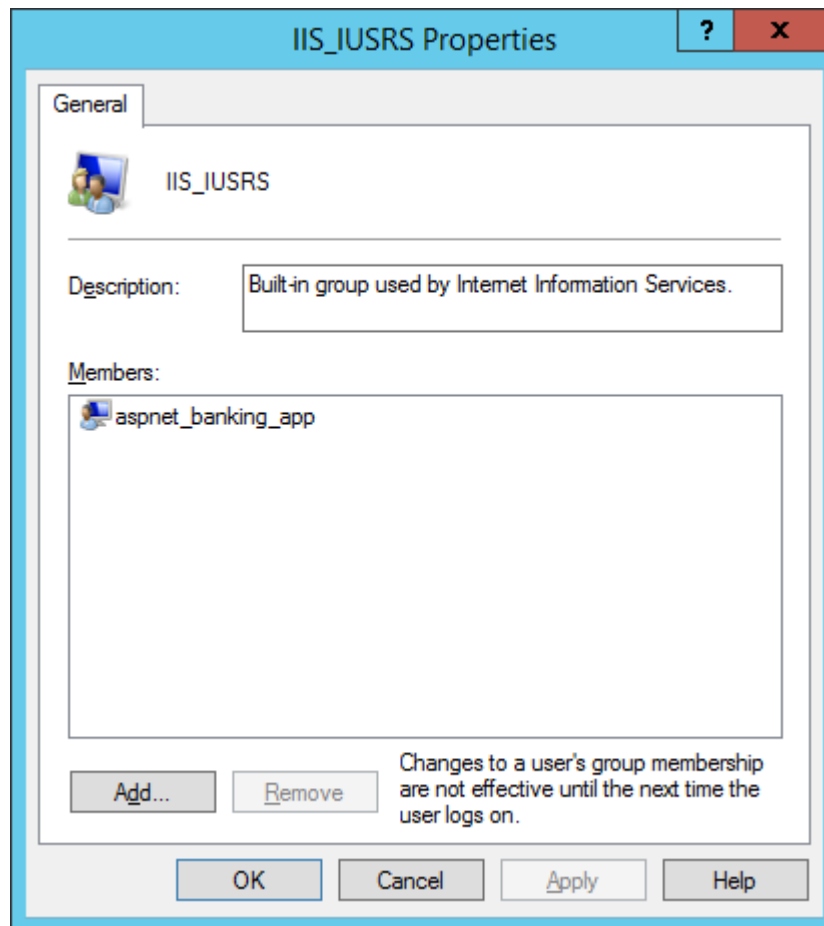
You might not always want to use the default user for an application, you can use Puppet to create users for other applications, like ASP.NET.

To configure ASP.NET apps to use accounts other than the default `Network Service`, create a user and `exec` resource:

```
user { 'aspnet_banking_app':
  ensure      => present,
  managehome  => true,
  comment     => 'ASP.NET Service account for Banking application',
  password    => 'banking_app_password',
  groups      => ['IIS_IUSRS', 'Users'],
  auth_membership => 'minimum',
  notify      => Exec['regiis_aspnet_banking_app']
}

exec { 'regiis_aspnet_banking_app':
  path        => 'c:\\windows\\Microsoft.NET\\Framework\\v4.0.30319',
  command     => 'aspnet_regiis.exe -ga aspnet_banking_app',
  refreshonly => true
}
```

In this example, the user is created in the appropriate groups, and the ASP.NET IIS registration command is run after the user is created to ensure file permissions are correct.



In the user resource, there are a few important details to note:

- `managehome` is set to create the user's home directory on disk.

- `auth_membership` is set to minimum, meaning that Puppet makes sure the `aspnet_banking_app` user is a part of the `IIS_IUSRS` and `Users` group, but doesn't remove the user from any other groups it might be a part of.
- `notify` is set on the user, and `refreshonly` is set on the `exec`. This tells Puppet to run `aspnet_regiis.exe` only when the `aspnet_banking_app` is created or changed.

Manage local groups

Local user accounts are often desirable for isolating applications requiring unique permissions. It can also be useful to manipulate existing local groups.

To add domain users or groups not present in the Domain Administrators group to the local Administrators group, use this code:

```
group { 'Administrators':
  ensure => 'present',
  members => ['DOMAIN\User'],
  auth_membership => false
}
```

In this case, `auth_membership` is set to false to ensure that `DOMAIN\User` is present in the Administrators group, but that other accounts that might be present in Administrators are not removed.

Note that the `groups` attribute of `user` and the `members` attribute of `group` might both accept SID values, like the well-known SID for Administrators, `S-1-5-32-544`.

Executing PowerShell code

Some Windows maintenance tasks require the use of Windows Management Instrumentation (WMI), and PowerShell is the most useful way to access WMI methods. Puppet has a special module that can be used to execute arbitrary PowerShell code.

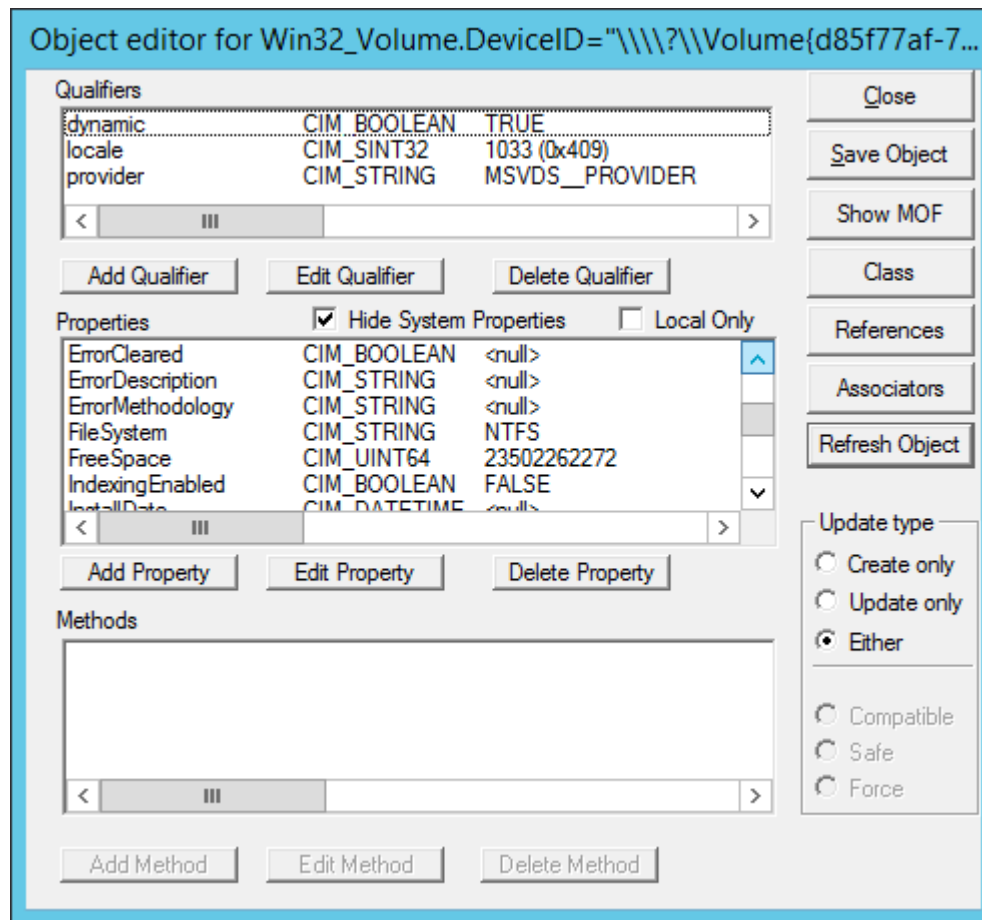
A common Windows maintenance task is to disable Windows drive indexing, because it can negatively impact disk performance on servers.

To disable drive indexing:

```
$drive = 'C:'

exec { 'disable-c-indexing':
  provider => powershell,
  command  => "\$wmi_volume = Get-WmiObject -Class Win32_Volume -Filter
'DriveLetter=\"${drive}\"'; if (\$wmi_volume.IndexingEnabled -ne \$True)
{ return }; \$wmi_volume | Set-WmiInstance -Arguments @{IndexingEnabled = \
$False}",
  unless   => "if ((Get-WmiObject -Class Win32_Volume -Filter
'DriveLetter=\"${drive}\"').IndexingEnabled) { exit 1 }",
}
```

You can see the results in your object editor window:



Using the Windows built-in WBEMTest tool, running this manifest sets `IndexingEnabled` to `FALSE`, which is the desired behavior.

This `exec` sets a few important attributes:

- The provider is configured to use PowerShell (which relies on the module).
- The command contains inline PowerShell, and as such, must be escaped with PowerShell variables preceded with `$` must be escaped as `\$`.
- The `unless` attribute is set to ensure that Puppet behaves idempotently, a key aspect of using Puppet to manage resources. If the resource is already in the desired state, Puppet does not modify the resource state.

Using templates to better manage Puppet code

While inline PowerShell is usable as an `exec` resource in your manifest, such code can be difficult to read and maintain, especially when it comes to handling escaping rules.

For executing multi-line scripts, use Puppet templates instead. The following example shows how you can use a template to organize the code for disabling Windows drive indexing.

```
$drive = 'C:'

exec { 'disable-c-indexing':
  command => template('Disable-Indexing.psl.erb'),
  provider => powershell,
  unless   => "if ((Get-WmiObject -Class Win32_Volume -Filter 'DriveLetter=\\\"$drive\\\"').IndexingEnabled) { exit 1 }",
}
```

The PowerShell code for `Disable-Indexing.ps1.erb` becomes:

```
function Disable-Indexing($Drive)
{
    $drive = Get-WmiObject -Class Win32_Volume -Filter "DriveLetter='$Letter'"
    if ($drive.IndexingEnabled -ne $True) { return }
    $drive | Set-WmiInstance -Arguments @{IndexingEnabled=$False} | Out-Null
}

Disable-Indexing -Drive '<%= @driveLetter %>'
```

Next steps

Now that you have your system up and running with PE, here are some recommendations on where to go next.

- Check out the [Forge](#) to download additional modules and start managing other things like [IIS](#) sites or machines running on [Azure](#).
- Learn how to develop high-quality modules with the [Puppet VSCode extension](#) and the [Puppet Development Kit \(PDK\)](#).
- See the [Configuring Puppet Enterprise](#) on page 189 docs to fine-tune things like the console, orchestration services, java, and proxy settings.
- See the docs on [Managing nodes](#) on page 314 to add more nodes to your inventory and classify them.
- Learn about [Managing and deploying Puppet code](#) on page 550.
- Check out our [Youtube channel](#).

Installing

A Puppet Enterprise deployment typically includes infrastructure components and agents, which are installed on nodes in your environment.

You can install infrastructure components in multiple configurations and scale up with compilers. You can install agents on *nix, Windows, and macOS nodes.

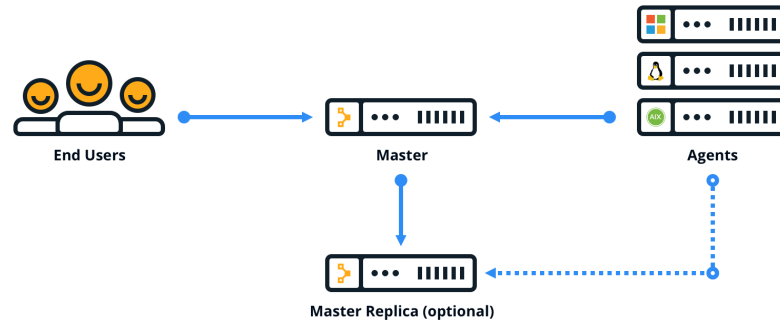
Choosing an architecture

There are several configurations available for Puppet Enterprise. The configuration you use depends on the number of nodes in your environment and the resources required to serve agent catalogs.

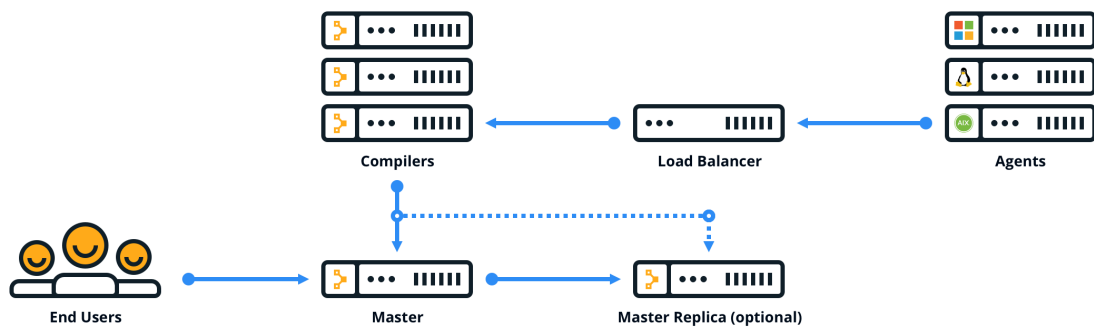
Configuration	Description	Node limit
Standard installation (Recommended)	All infrastructure components are installed on the master. This installation type is the easiest to install, upgrade, and troubleshoot.	Up to 4,000
Large installation	Similar to a standard installation, plus one or more compilers and a load balancer which help distribute the agent catalog compilation workload.	4,000–20,000
Extra-large installation	Similar to a large installation, plus a separate node which hosts the PE-PostgreSQL instance.	More than 20,000

Tip: You can add high availability to a installation with or without compilers by configuring a replica of your master. High availability isn't supported with standalone PE-PostgreSQL.

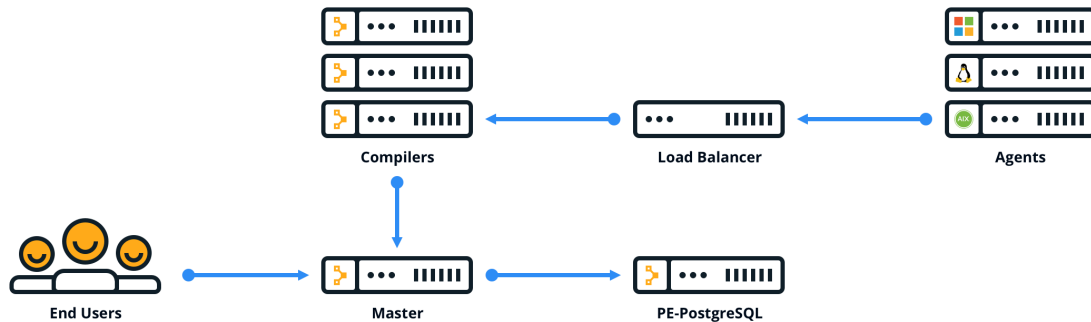
Standard installation



Large installation



Extra-large installation



System requirements

Refer to these system requirements for Puppet Enterprise installations.

Hardware requirements

These hardware requirements are based on internal testing at Puppet and are meant only as guidelines to help you determine your hardware needs.

Your installation's existing code base can significantly affect performance, so adjusting expectations based on differences between your installation's code base and the testing code base as documented in [How we develop hardware requirements](#) is important to make the best use of the requirements on this page.

Related information

[Methods for configuring Puppet Enterprise](#) on page 189

After you've installed Puppet Enterprise (PE), optimize it for your environment by configuring and tuning settings as needed. For example, you might want to add your own certificate to the whitelist, increase the max-threads setting for http and https requests, or configure the number of JRuby instances.

[Tuning standard installations](#) on page 208

Use these guidelines to configure your installation to maximize its use of available system (CPU and RAM) resources.

Hardware requirements for standard installations

In standard installations, hardware requirements differ depending on the number of nodes you're managing. To take advantage of progressively larger hardware, you must configure your environment to use additional resources.

Node volume	Cores	RAM	/opt/	/var/	AWS EC2	Azure
Trial use	2	8 GB	20 GB	24 GB	m5.large	A2 v2
Up to 2,000	4	8 GB	50 GB	24 GB	c5.xlarge	F4s v2
Up to 3,500	8	16 GB	80 GB	24 GB	c5.2xlarge	F8s v2
Up to 4,000	16	32 GB	100 GB	24 GB	c5.4xlarge	F16s v2

Important notes on the data in this chart:

- **Trial mode:** Although the m5.large instance type is sufficient for trial use, it is not supported. A minimum of four cores is required for production workloads.

- **Azure:** Azure requirements are not currently tested by Puppet, but are presented here as our best guidance based on comparable EC2 instance testing.
- **/opt/ storage requirements:** The database should not exceed 50% of /opt/ to allow for future upgrades.
- **/var/ storage requirements:** There are roughly 20 log files stored in /var/ which are limited in size to 1 GB each. Log retention settings make it unlikely that the maximum capacity will be needed, but we recommend allocating 24 GB to avoid issues.

Hardware requirements for large installations

If you are managing more than 4,000 nodes, you can add load-balanced compilers to your installation to increase the number of agents you can manage.

Each compiler increases capacity by approximately 1,500–3,000 nodes, until you exhaust the capacity of PuppetDB or the console, which run on the master. If you start to see performance issues around 8,000 nodes, you can adjust your hardware or move to a larger base infrastructure.

Note: When you expand your deployment to use compilers, you must also start using load balancers. It is simpler to upgrade your hardware in your installation, if you can, than to add compilers and load balancers.

To manage more than 4,000 nodes, we recommend the following minimum hardware:

Node volume	Node	Cores	RAM	/opt/	/var/	EC2
4,000–20,000	Primary node	16	32	150	10	c5.4xlarge
	Each compiler (1,500 - 3,000 nodes)	4	8	30	2	m5.xlarge

Hardware requirements for extra-large installations

You can continue to expand your installation beyond 20,000 nodes by adding compilers and increasing the size of your master and PE-PostgreSQL nodes.

You can also increase the number of nodes you support by tuning certain settings such as the run interval (changing how often agents check in) and `report-ttl` (changing how long PuppetDB stores reports).

To manage more than 4,000 nodes, we recommend the following minimum hardware:

Node volume	Node	Cores	RAM	/opt/	/var/	EC2
20,000+	Primary node	16	32	150	10	c5.4xlarge
	Each compiler (1,500 - 3,000 nodes)	4	8	30	2	m5.xlarge
	PE- PostgreSQL node	16	128	300	4	r5.4xlarge

If you manage more than 20,000 nodes, contact Puppet professional services to talk about optimizing your setup for your specific requirements

How we develop hardware requirements: Performance test methods

Puppet tests the performance of Puppet Enterprise and develops hardware requirements based on our testing. We continue to invest in improvements in our performance testing methods.

The performance of Puppet Enterprise is highly dependent on the code base used by a specific installation. The details included here are intended to help you evaluate how your specific installation compares to what we used to generate our scale and hardware estimates so that you can better estimate what you need to manage your Puppet infrastructure.

Tools: Gatling and puppet infrastructure tune

We use [gatling-puppet-load-test](#), a framework we've developed to perform load and performance testing for PE and open source Puppet. It uses [Gatling](#), an open source load and performance testing tool that records and replays HTTP traffic, then generates reports about the performance of the simulated requests. Using Gatling allows us to simulate agent requests in a full PE installation where Puppet Server is driving communication with PuppetDB, the node classifier, and other tools.

We also use `puppet infrastructure tune`, which provides configuration settings to apply to the installation.

Platform: Amazon EC2

We use Amazon EC2 instances for our performance tests with a master node running PE and a metrics node running the simulated agents and capturing the performance data. We know that the instances' performance varies, but this has less of an impact on overall performance than the variations in the size and complexity of users' code bases.

Table 1: EC2 instance types

AWS EC2	Cores	RAM	Testing use
m5.large	2	8 GB	Confirming trial size works
c5.xlarge	4	8 GB	Scale
c5.2xlarge	8	16 GB	Scale, Apples-to-apples, Soak
c5.4xlarge	16	32 GB	Scale

Test environment

Control repository

We use [puppetlabs-pe_perf_control_repo](#) with r10k to classify the simulated agents using roles and profiles, with three sizes defined.

Facts and reports

Although the simulated agents all share the same classification, we use dynamic facts and varying report responses (no change, failure, intentional change, corrective change) to better simulate a typical customer environment.

	role::by_size::small	role::by_size::medium	role::by_size::large
Resources	129	750	1267
Classes	24	71	120
Facts	151	151	151

Classification

```
class role::by_size::small {
  include ::profile::tomcat::basic
  include ::profile::postgresql::basic
}

class role::by_size::medium {
  include ::role::by_size::small

  include ::profile::users
  include ::profile::sysop::packages
  include ::profile::motd
  include ::profile::hiera_check
}
```

```

class role::by_size::large {
  include ::role::by_size::medium

  include ::profile::apache::basic
  include ::profile::influxdb::basic
  include ::profile::loop_through_file_resources
}

```

Environment cache

Although we recommend [enabling environment caching](#) to improve performance in production environments, we do not enable it in our performance testing environment to avoid unrealistic results due to the identical classification of the simulated agents.

Fact, resource, and class counts

You can query PuppetDB using Puppet Query Language (PQL) to find the number of facts, resources, and classes for a given node.

First, you will need a token with permissions to query Puppet DB. If you don't already have one, you can generate one using `puppet-access` on the master. For example:

```
puppet access login -l ld
```

See our [Token-based authentication](#) on page 242 docs for more information on generating tokens.

Once you have a valid token, determine the number of facts, resources, and classes for the specified node using the example code below. In the examples, we used `perf-agent1` as the node name.

```
puppet query "facts[count()] {certname = 'perf-agent1'}"
```

```
puppet query "resources[count()] {certname = 'perf-agent1'}"
```

```
puppet query "resources[count()] {type = 'Class' and certname = 'perf-agent1'}"
```

For background and additional examples, check out this [blog post](#) or view the full [PQL documentation](#).

Testing methodology

Warming up Puppet Server

JRuby 9000 provides better overall performance than the previous version, but it also comes with a warm-up burden. For some tests we want the Puppet Server process to be warmed up before we begin. Our warm-up process involves running a portion of the agents to generate about 300 connections to the endpoints served by JRubies, multiplied by the number of configured JRuby instances. In general, the closer you are to the node capacity of your install, the faster it will warm up when running the normal load. Our process takes about five minutes.

Apples-to-apples testing

To do an apples-to-apples test, we run a simulation for two different PE versions of 600 agents each (classified as large) checking in at the default 30-minute interval for eight iterations. We run the warm-up procedure before the simulation. Then we compare response times, system resource usage, and process resource usage. We do this on a weekly basis for all active development branches to ensure that general performance doesn't degrade during development.

Scale testing

In order to determine the node capacity of each targeted EC2 instance type, we perform scale testing with a small load that incrementally increases the number of simulated agents until requests begin to time out or respond with errors.

The scale test is structured to simulate the default 30-minute agent check-in interval. We start the test with an agent volume below the expected performance threshold and add 100 agents at a time until more than 10 timeouts are encountered. For example, when testing the EC2 c5.2xlarge (8 cores and 16 GB), we start with 3,000 agents.

Cold-start versus warm-start scale testing

To simulate recovering from a Puppet Server restart, we use a cold-start scenario. We restart the `pe-puppetserver` service between each iteration so that the warm-up burden will reduce the maximum node count. This is the most conservative estimate of node capacity we test for.

To determine burst capacity, we use a warm-start scenario. When doing a warm-start scale test, we don't restart the `pe-puppetserver` service, and we run the test on the same host that ran the cold-start tests to ensure it is already warmed up.

The extra capacity provided by a warmed up system is great for serving Puppet agent runs triggered by orchestration from the console, Continuous Delivery for Puppet Enterprise, or Bolt. Therefore we recommend keeping your node count near the cold-start numbers.

Soak testing

To ensure PE doesn't suffer a performance degradation over time we run a soak test on each release. We use 600 agents, classified as large, checking in at the default 30-minute interval for 14 days. We run the warm-up before we start the test. This two-week test lets us verify that all garbage collection features are being triggered and keeping things under control. We check the Gatling response time graphs to ensure that performance is stable throughout the test.

Supported operating systems

Puppet Enterprise supports various operating systems depending on the role a machine assumes in your infrastructure.

Supported operating systems and devices

When choosing an operating system, first consider the machine's role. Different roles support different operating systems and architectures.

Tip: For details about platform support lifecycles and planned end-of-life support, see [Platform support lifecycle](#) on the Puppet website.

Master platforms

The master role can be installed on these operating systems and architectures.

Operating system	Versions	Architecture
Enterprise Linux <ul style="list-style-type: none"> CentOS Oracle Linux Red Hat Enterprise Linux Scientific Linux 	6, 7, 8	<ul style="list-style-type: none"> x86_64 FIPS 140-2 compliant Enterprise Linux 7
SUSE Linux Enterprise Server	12	x86_64
Ubuntu (General Availability kernels)	16.04, 18.04	amd64

Agent platforms

The agent role can be installed on these operating systems and architectures.

Operating system	Versions	Architecture
AIX	6.1, 7.1, 7.2	

Operating system	Versions	Architecture
Amazon Linux	2	
Debian	Jessie (8), Stretch (9), Buster (10)	<ul style="list-style-type: none"> • i386 • amd64
Enterprise Linux	5, 6, 7, 8	<ul style="list-style-type: none"> • x86_64 • i386 for versions 5, 6 • ppc64le for version 7 • aarch64 for version 7 • FIPS 140-2 compliant Enterprise Linux 7
<ul style="list-style-type: none"> • CentOS • Oracle Linux • Red Hat Enterprise Linux • Scientific Linux 	Note: Scientific Linux 5 is not supported.	
Fedora	28, 29	<ul style="list-style-type: none"> • x86_64 • i386 for version 25
macOS	10.12, 10.13, 10.14	
Microsoft Windows	7, 8.1, 10	<ul style="list-style-type: none"> • x64 • x86
Microsoft Windows Server	2008, 2008R2, 2012, 2012R2, 2012R2 core, 2016, 2016 core, and Desktop Experience, 2019	<ul style="list-style-type: none"> • x64 for all 2008 and 2012 series • x86 for 2008
Solaris	10, 11	<ul style="list-style-type: none"> • SPARC • i386
SUSE Linux Enterprise Server	11, 12, 15	<ul style="list-style-type: none"> • x86_64 • i386 for version 11 • ppc64le for version 12
Ubuntu (General Availability kernels)	16.04, 18.04	<ul style="list-style-type: none"> • amd64 • i386 for versions 16.04 • ppc64el for version 16.04

Note: Some operating systems require an active subscription with the vendor's package management system (for example, the Red Hat Network) to install dependencies.

CentOS dependencies

When you install PE or an agent, these packages are also installed from the various operating system repositories.

If the machine you're installing on has internet access, dependencies are set up during installation. If your machine doesn't have internet access, you must manually install dependencies.

	All Nodes	Master Nodes	Console Nodes	Console/Console DB Nodes
pciutils	x			
which	x			
libxml2	x			
dmidecode	x			
net-tools	x			

	All Nodes	Master Nodes	Console Nodes	Console/Console DB Nodes
curl		x	x	
mailcap		x	x	
libjpeg		x		x
libtool-ltdl		x	x	
unixODBC		x	x	
libxslt				x
zlib	x			

RHEL dependencies

When you install PE or an agent, these packages are also installed from the various operating system repositories.

If the machine you're installing on has internet access, dependencies are set up during installation. If your machine doesn't have internet access, you must manually install dependencies.

	All Nodes	Master Nodes	Console Nodes	Console/Console DB Nodes
pciutils	x			
which	x			
libxml2	x			
dmidecode	x			
net-tools	x			
cronie (RHEL 6, 7)	x			
vixie-cron (RHEL 4, 5)	x			
curl		x	x	
mailcap		x	x	
libjpeg		x		x
libtool-ltdl (RHEL 7)		x	x	
unixODBC (RHEL 7)		x	x	
libxslt				x
zlib	x			
gtk2		x		

SUSE Linux Enterprise Server dependencies

When you install PE or an agent, these packages are also installed from the various operating system repositories.

If the machine you're installing on has internet access, most dependencies are installed during installation. If you encounter problems, inspect the error messages for packages that require other SUSE Linux Enterprise Server packaging modules to be enabled, and use `zypper package-search <PACKAGE NAME>` to locate them for manual installation.

	All Nodes	Master Nodes	Console Nodes	Console/Console DB Nodes
pciutils	x			
pmtools	x			
cron	x			
libxml2	x			
net-tools	x			
libxslt	x	x		x
curl		x	x	
libjpeg		x		x
db43		x	x	
unixODBC		x	x	
zlib	x			
libboost_atomic	x			
libboost_chrono	x			
libboost_date_time	x			
libboost_filesystem	x			
libboost_locale	x			
libboost_log	x			
libboost_program_options	x			
libboost_random	x			
libboost_regex	x			
libyaml-cpp	x			

Ubuntu dependencies

When you install PE or an agent, these packages are also installed from the various operating system repositories.

If the machine you're installing on has internet access, dependencies are set up during installation. If your machine doesn't have internet access, you must manually install dependencies.

	All Nodes	Master Nodes	Console Nodes	Console/Console DB Nodes
pciutils	x			
dmidecode	x			
cron	x			
libxml2	x			
hostname	x			
libldap-2.4-2	x			
libreadline5	x			
file		x	x	
libmagic1		x	x	

	All Nodes	Master Nodes	Console Nodes	Console/Console DB Nodes
libpcre3		x	x	
curl		x	x	
perl		x	x	
mime-support		x	x	
libcap2		x	x	
libjpeg62		x		x
libxslt1.1				x
libgtk2.0-0		x	x	x
ca-certificates-java		x	x	x
openjdk-7-jre-headless*		x	x	x
libssp-uuid16		x	x	x
zlib	x			

*For Ubuntu 18.04, use openjdk-8-jre-headless. This package requires the [universe repository](#) to install.

AIX dependencies and limitations

Before installing the agent on AIX systems, install these packages.

- bash
- zlib
- readline
- curl
- OpenSSL



CAUTION: For cURL and OpenSSL, you must use the versions provided by the "AIX Toolbox Cryptographic Content" repository, which is available via IBM support. Note that the cURL version must be 7.9.3. Do not use the cURL version in the AIX toolbox package for Linux applications, as that version does not include support for OpenSSL.

To install the bash, zlib, and readline packages on a node directly, run `rpm -Uvh` with the following URLs. The RPM package provider must be run as root.

- <ftp://ftp.software.ibm.com/aix/freeSoftware/aixtoolbox/RPMS/ppc/bash/bash-3.2-1.aix5.2.ppc.rpm>
- <ftp://ftp.software.ibm.com/aix/freeSoftware/aixtoolbox/RPMS/ppc/zlib/zlib-1.2.3-4.aix5.2.ppc.rpm>
- <ftp://ftp.software.ibm.com/aix/freeSoftware/aixtoolbox/RPMS/ppc/readline/readline-6.1-1.aix6.1.ppc.rpm> (AIX 6.1 and 7.1 only)
- <ftp://ftp.software.ibm.com/aix/freeSoftware/aixtoolbox/RPMS/ppc/readline/readline-4.3-2.aix5.1.ppc.rpm> (AIX 5.3 only)

If you are behind a firewall or running an http proxy, the above commands might not work. Instead, use the AIX toolbox packages download available from IBM.

GPG verification does not work on AIX, because the RPM version it uses is too old. The AIX package provider doesn't support package downgrades (installing an older package over a newer package). Avoid using leading zeros when specifying a version number for the AIX provide, for example, use 2.3.4 not 02.03.04.

The PE AIX implementation supports the NIM, BFF, and RPM package providers. Check the type reference for technical details on these providers.

Solaris dependencies and limitations

Solaris support is agent only.

For Solaris 10, these packages are required:

- SUNWgccruntime
- SUNWzlib
- In some instances, bash might not be present on Solaris systems. It needs to be installed before running the installer. Install bash via the media used to install the operating system, or via CSW if that is present on your system. (CSWbash or SUNWbash are both suitable.)

For Solaris 11 these packages are required:

- system/readline
- system/library/gcc-45-runtime
- library/security/openssl

These packages are available in the Solaris release repository, which is enabled by default in version 11. The installer automatically installs these packages; however, if the release repository is not enabled, the packages must be installed manually.

Upgrade your operating system with PE installed

If you have PE installed, take extra precautions before performing a major upgrade of your machine's operating system.

Performing major upgrades of your operating system with PE installed can cause errors and issues with PE. A major operating system upgrade is an upgrade to a new whole version, such as an upgrade from CentOS 6.0 to 7.0; it does not refer to a minor version upgrade, like CentOS 6.5 to 6.6. Major upgrades typically require a new version of PE.

1. Back up your databases and other PE files.
2. Perform a complete uninstall (using the `-p` and `-d` uninstaller options).
3. Upgrade your operating system.
4. Install PE.
5. Restore your backup.

Related information

[Back up your infrastructure](#) on page 698

PE backup creates a copy of your Puppet infrastructure, including configuration, certificates, code, and PuppetDB.

[Restore your infrastructure](#) on page 699

Use the restore commands to migrate your PE master to a new host or to recover from system failure.

[Uninstalling](#) on page 177

Puppet Enterprise includes a script for uninstalling. You can uninstall component infrastructure nodes or from agent nodes.

[Installing Puppet Enterprise](#) on page 135

You can install PE using *express install*, which relies on defaults, or *text install*, where you provide a `pe.conf` file with installation parameters. Either of these methods is appropriate for installing infrastructure components on your master.

Supported browsers

The following browsers are supported for use with the console.

Browser	Supported versions
Google Chrome	Current version as of release
Mozilla Firefox	Current version as of release
Microsoft Edge	Current version as of release

Browser	Supported versions
Apple Safari	10 or later

System configuration

Before installing Puppet Enterprise, make sure that your nodes and network are properly configured.

Note: Port numbers are Transmission Control Protocols (TCP), unless noted otherwise.

Timekeeping and name resolution

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

Timekeeping

Use NTP or an equivalent service to ensure that time is in sync between your master, which acts as the certificate authority, and any agent nodes. If time drifts out of sync in your infrastructure, you might encounter issues such as agents 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 master. Ensure that the master can be reached by domain name lookup by all future agent nodes.

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

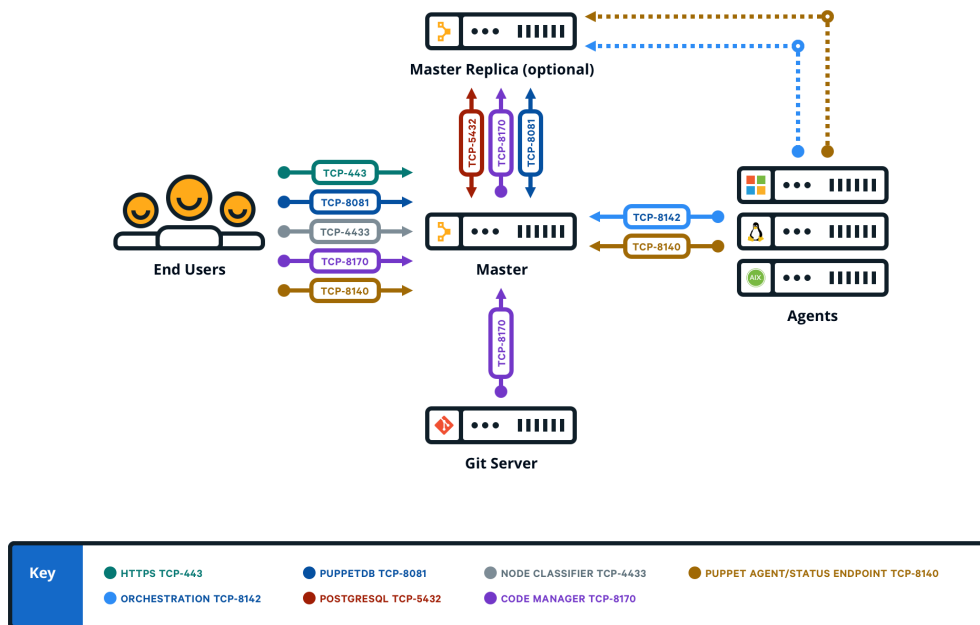
Web URLs used for deployment and management

PE uses some external web URLs for certain deployment and management tasks. You might want to ensure these URLs are reachable from your network prior to installation, and be aware that they might be called at various stages of configuration.

URL	Enables
forgeapi.puppet.com	Puppet module downloads.
pm.puppetlabs.com	Agent module package downloads.
s3.amazonaws.com	Agent module package downloads (redirected from pm.puppetlabs.com).
rubygems.org	Puppet and Puppet Server gem downloads.
github.com	Third-party module downloads not served by the Forge and access to control repositories.

Firewall configuration for standard installations

These are the port requirements for standard installations.

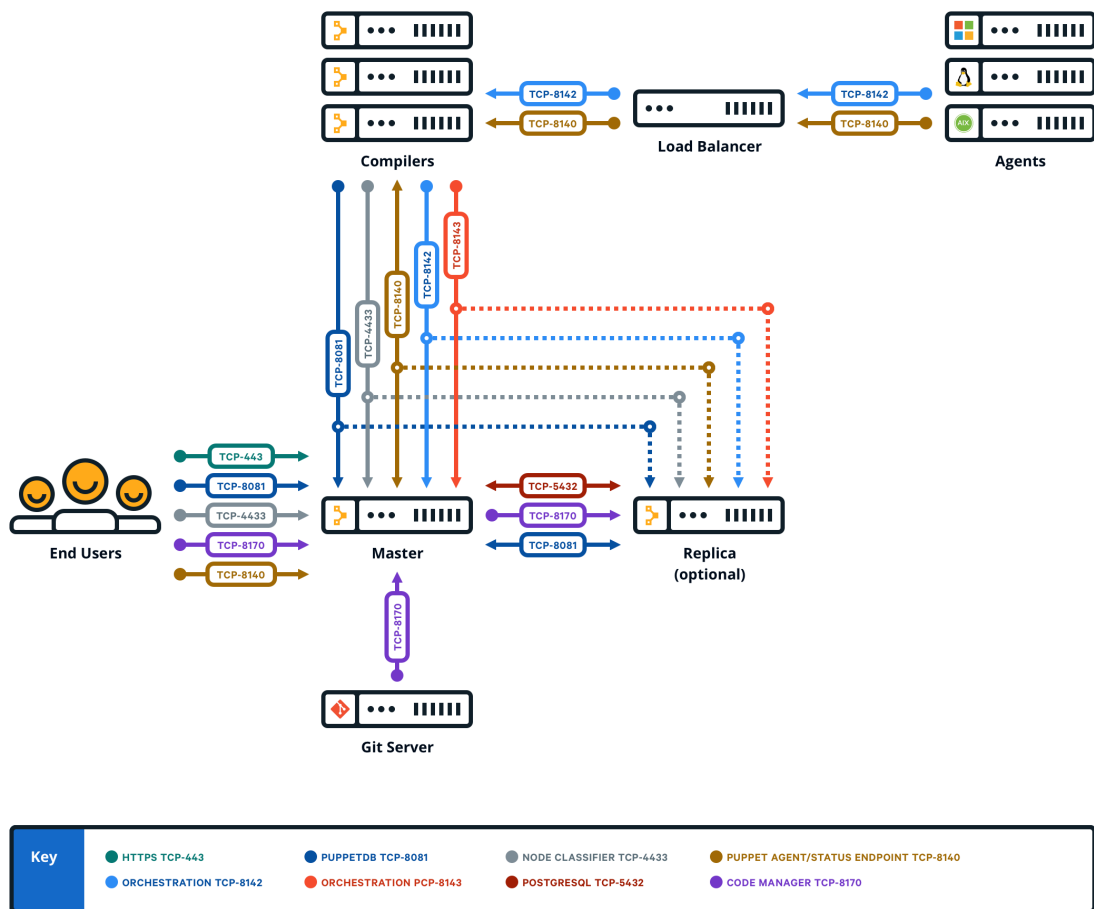


Port	Use
8140	<ul style="list-style-type: none"> The master uses this port to accept inbound traffic/ requests from agents. The console sends requests to the master on this port. Certificate requests are passed over this port unless <code>ca_port</code> is set differently. status checks are sent over this port. Classifier group: PE Master
443	<ul style="list-style-type: none"> This port provides host access to the console The console accepts HTTPS traffic from end users on this port. Classifier group: PE Console
4433	<ul style="list-style-type: none"> This port is used as a classifier / console services API endpoint. The master communicates with the console over this port. Classifier group: PE Console
8081	<ul style="list-style-type: none"> accepts traffic/requests on this port. The master and console send traffic to on this port. status checks are sent over this port. Classifier group: PE PuppetDB

Port	Use
8142	<ul style="list-style-type: none"> Orchestrator and the Run Puppet button use this port on the master to accept inbound traffic/responses from agents via the agent. Classifier group: PE Orchestrator
8170	<ul style="list-style-type: none"> uses this port to deploy environments, run webhooks, and make API calls.
5432	<ul style="list-style-type: none"> This port is used to replicate data between the master and replica.
8150 and 8151	<ul style="list-style-type: none"> uses port 8150 for HTTP and 8151 for HTTPS. Any node classified as a server must be able to use these ports.

Firewall configuration for large installations with compilers

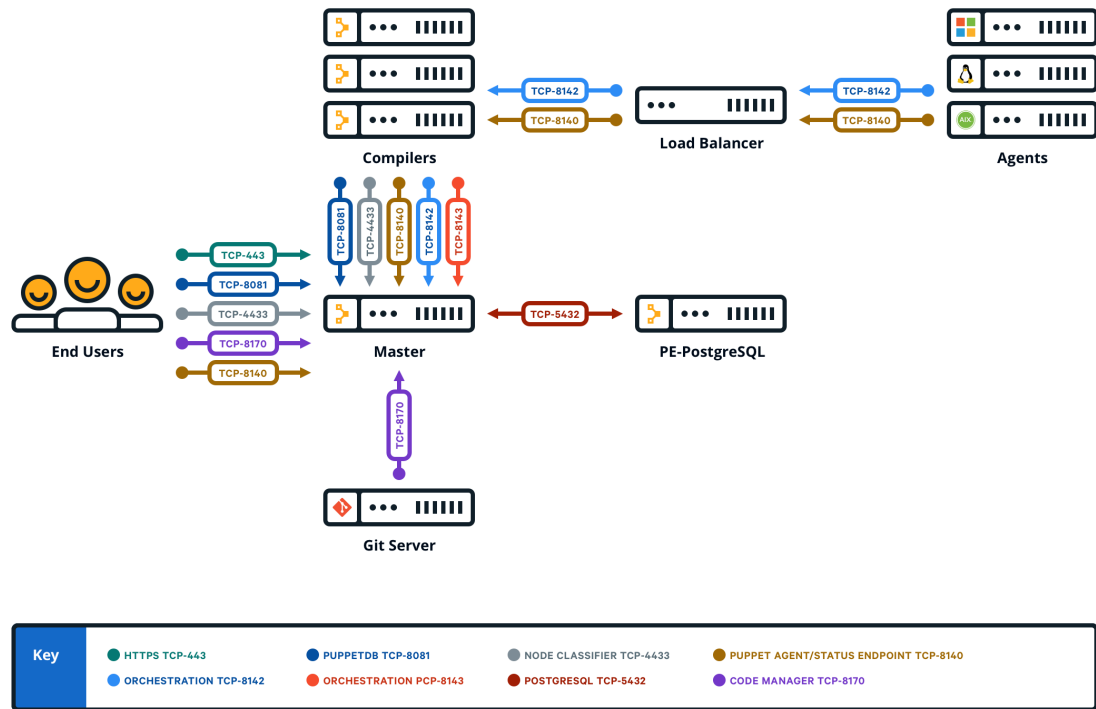
These are the port requirements for large installations with compilers.



Port	Use
8140	<ul style="list-style-type: none"> • The master uses this port to accept inbound traffic/ requests from agents. • The console sends requests to the master on this port. • Certificate requests are passed over this port unless <code>ca_port</code> is set differently. • status checks are sent over this port. • The master uses this port to send status checks to compilers. (Not required to run PE.) • Classifier group: PE Master
443	<ul style="list-style-type: none"> • This port provides host access to the console • The console accepts HTTPS traffic from end users on this port. • Classifier group: PE Console
4433	<ul style="list-style-type: none"> • This port is used as a classifier / console services API endpoint. • The master communicates with the console over this port. • Classifier group: PE Console
8081	<ul style="list-style-type: none"> • accepts traffic/requests on this port. • The master and console send traffic to on this port. • status checks are sent over this port. • Classifier group: PE PuppetDB
8142	<ul style="list-style-type: none"> • Orchestrator and the Run Puppet button use this port on the master to accept inbound traffic/responses from agents via the agent. • Classifier group: PE Orchestrator
8143	<ul style="list-style-type: none"> • Orchestrator uses this port to accept connections from brokers to relay communications. The orchestrator client also uses this port to communicate with the orchestration services running on the master. If you install the client on a workstation, this port must be available on the workstation. • Classifier group: PE Orchestrator
8170	<ul style="list-style-type: none"> • uses this port to deploy environments, run webhooks, and make API calls.
5432	<ul style="list-style-type: none"> • This port is used to replicate data between the master and replica.
8150 and 8151	<ul style="list-style-type: none"> • uses port 8150 for HTTP and 8151 for HTTPS. Any node classified as a server must be able to use these ports.

Firewall configuration for extra-large installations with compilers and standalone PE-PostgreSQL

These are the port requirements for extra-large installations with compilers and standalone PE-PostgreSQL



Port	Use
8140	<ul style="list-style-type: none"> The master uses this port to accept inbound traffic/ requests from agents. The console sends requests to the master on this port. Certificate requests are passed over this port unless <code>ca_port</code> is set differently. status checks are sent over this port. The master uses this port to send status checks to compilers. (Not required to run PE.) Classifier group: PE Master
443	<ul style="list-style-type: none"> This port provides host access to the console The console accepts HTTPS traffic from end users on this port. Classifier group: PE Console
4433	<ul style="list-style-type: none"> This port is used as a classifier / console services API endpoint. The master communicates with the console over this port. Classifier group: PE Console

Port	Use
8081	<ul style="list-style-type: none"> accepts traffic/requests on this port. The master and console send traffic to on this port. status checks are sent over this port. Classifier group: PE PuppetDB
8142	<ul style="list-style-type: none"> Orchestrator and the Run Puppet button use this port on the master to accept inbound traffic/responses from agents via the agent. Classifier group: PE Orchestrator
8143	<ul style="list-style-type: none"> Orchestrator uses this port to accept connections from brokers to relay communications. The orchestrator client also uses this port to communicate with the orchestration services running on the master. If you install the client on a workstation, this port must be available on the workstation. Classifier group: PE Orchestrator
8170	<ul style="list-style-type: none"> uses this port to deploy environments, run webhooks, and make API calls.
5432	<ul style="list-style-type: none"> The standalone PE-PostgreSQL node uses this port to accept inbound traffic/requests from the master.
8150 and 8151	<ul style="list-style-type: none"> uses port 8150 for HTTP and 8151 for HTTPS. Any node classified as a server must be able to use these ports.

What gets installed and where?

Puppet Enterprise installs several software components, configuration files, databases, services and users, and log files. It's useful to know the locations of these should you ever need to troubleshoot or manage your infrastructure.

Software components installed

PE installs several software components and dependencies. These tables show which version of each component is installed for releases dating back to the previous long term supported (LTS) release.

The functional components of the software are separated between those packaged with the agent and those packaged on the server side (which also includes the agent).

Note: PE also installs other dependencies, as documented in the system requirements.

This table shows the components installed on all agent nodes.

Note: 5 is a backwards-compatible evolution of , which is built into 4.9.0 and higher. To provide some backwards-compatible features, it uses the classic 3.x.x codebase version listed in this table.

Version	Agent				Resource API		Ruby	OpenSSL
2019.2.2	6.10.1	6.10.1	3.14.5	3.6.0	1.8.9	N/A	2.5.7	1.1.1d
2019.2.1	6.10.1	6.10.1	3.14.5	3.6.0	1.8.9	N/A	2.5.7	1.1.1d
2019.2.0	6.10.1	6.10.1	3.14.5	3.6.0	1.8.9	N/A	2.5.7	1.1.1d

Version	Agent				Resource API		Ruby	OpenSSL
2019.1.3	6.4.4	6.4.4	3.13.4	3.5.0	1.8.9	N/A	2.5.7	1.1.1d
2019.1.1	6.4.3	6.4.3	3.13.3	3.5.0	1.8.6	N/A	2.5.3	1.1.1a
2019.1.0	6.4.2	6.4.2	3.13.2	3.5.0	1.8.2	N/A	2.5.3	1.1.1a
2019.0.4	6.1.10	6.0.10	3.12.5	3.4.6	1.6.5	N/A	2.5.3	1.1.1a
2019.0.3	6.0.9	6.0.9	3.12.4	3.4.6	1.6.4	N/A	2.5.3	1.1.1a
2019.0.2	6.0.5	6.0.5	3.12.3	3.4.6	1.6.3	N/A	2.5.1	1.0.2n
2019.0.1	6.0.4	6.0.4	3.12.1	3.4.5	1.6.2	N/A	2.5.1	1.0.2n
2019.0.0	6.0.2	6.0.2	3.12.0	3.4.5	1.6.0	N/A	2.5.1	1.1.0h
2018.1.11 (LTS)	5.5.17	5.5.17	3.11.10	3.4.6	N/A	2.12.5	2.4.9	1.0.2t
2018.1.9	5.5.16	5.5.16	3.11.9	3.4.6	N/A	2.12.4	2.4.5	1.0.2r
2018.1.8	5.5.14	5.5.14	3.11.8	3.4.6	N/A	2.12.4	2.4.5	1.0.2r
2018.1.7	5.5.10	5.5.10	3.11.7	3.4.6	N/A	2.12.4	2.4.5	1.0.2n
2018.1.5	5.5.8	5.5.8	3.11.6	3.4.5	N/A	2.12.4	2.4.4	1.0.2n
2018.1.4	5.5.6	5.5.6	3.11.4	3.4.4	N/A	2.12.3	2.4.4	1.0.2n
2018.1.3	5.5.4	5.5.3	3.11.3	3.4.3	N/A	2.12.2	2.4.4	1.0.2n
2018.1.2	5.5.3	5.5.2	3.11.2	3.4.3	N/A	2.12.2	2.4.4	1.0.2n
2018.1.0	5.5.1	5.5.1	3.11.1	3.4.3	N/A	2.12.1	2.4.4	1.0.2n

This table shows components installed on server nodes.

Note: FIPS-compliant version 2019.2 includes version 9.6.

Version	Server				Services	Agentless Catalog Executor (ACE) Services	Java		Nginx	
2019.2.2	6.7.1	6.7.3	3.3.3	1.9.6	1.33.0	1.0.0	11.5	1.8.0	N/A	1.16.1
2019.2.1	6.7.1	6.7.3	3.3.3	1.9.6	1.33.0	1.0.0	11.5	1.8.0	N/A	1.16.1
2019.2.0	6.7.1	6.7.2	3.3.3	1.9.6	1.33.0	1.0.0	11.5	1.8.0	N/A	1.16.1
2019.1.3	6.3.2	6.3.6	3.2.3	1.9.5	1.34.0	1.0.0	9.6.15	1.8.0	N/A	1.16.1
2019.1.1	6.3.1	6.3.4	3.2.0	1.9.5	1.26.0	0.9.1	9.6.13	1.8.0	N/A	1.14.2
2019.1.0	6.3.0	6.3.2	3.2.0	1.9.5	1.17.0	0.9.1	9.6.12	1.8.0	N/A	1.14.2
2019.0.4	6.0.5	6.0.4	3.0.4	1.9.4	1.26.0	N/A	9.6.13	1.8.0	N/A	1.14.2
2019.0.3	6.0.4	6.0.3	3.0.3	1.9.4	1.15.0	N/A	9.6.12	1.8.0	N/A	1.14.2
2019.0.2	6.0.3	6.0.2	3.0.3	1.9.3	1.10.0	N/A	9.6.10	1.8.0	N/A	1.14.0
2019.0.1	6.0.2	6.0.1	3.0.3	1.9.3	1.1.0	N/A	9.6.10	1.8.0	N/A	1.14.0
2019.0.0	6.0.1	6.0.0	3.0.2	1.9.3	0.24.0	N/A	9.6.10	1.8.0	N/A	1.14.0

Version			Server	Services	Agentless Catalog Executor (ACE) Services			Java			Nginx
2018.1.11 (LTS)	5.3.10	5.2.11	2.6.7	1.9.2	N/A	N/A	9.6.15	1.8.0	5.15.5	1.16.1	
2018.1.9	5.3.9	5.2.9	2.6.6	1.9.2	N/A	N/A	9.6.13	1.8.0	5.15.5	1.14.2	
2018.1.8	5.3.8	5.2.8	2.6.5	1.9.2	N/A	N/A	9.6.12	1.8.0	5.15.5	1.14.2	
2018.1.7	5.3.7	5.2.7	2.6.5	1.9.2	N/A	N/A	9.6.10	1.8.0	5.15.5	1.14.0	
2018.1.5	5.3.6	5.2.6	2.6.5	1.9.2	N/A	N/A	9.6.10	1.8.0	5.15.5	1.14.0	
2018.1.4	5.3.5	5.2.4	2.6.2	1.9.2	N/A	N/A	9.6.10	1.8.0	5.15.3	1.14.0	
2018.1.3	5.3.4	5.2.4	2.6.2	1.9.2	N/A	N/A	9.6.8	1.8.0	5.15.3	1.14.0	
2018.1.2	5.3.3	5.2.2	2.6.2	1.9.2	N/A	N/A	9.6.8	1.8.0	5.15.3	1.12.1	
2018.1.0	5.3.2	5.2.2	2.6.2	1.8.1	N/A	N/A	9.6.8	1.8.0	5.15.3	1.12.1	

Executable binaries and symlinks installed

PE installs executable binaries and symlinks for interacting with tools and services.

On *nix nodes, all software is installed under `/opt/puppetlabs`.

On Windows nodes, all software is installed in Program Files at Puppet Labs\Puppet.

Executable binaries on *nix are in `/opt/puppetlabs/bin` and `/opt/puppetlabs/sbin`.

Tip: To include binaries in your default `$PATH`, manually add them to your profile or export the path:

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

To make essential Puppet tools available to all users, the installer automatically creates symlinks in `/usr/local/bin` for the `facter`, `puppet`, `pe-man`, `r10k`, and `hiera` binaries. Symlinks are created only if `/usr/local/bin` is writeable. Users of AIX and Solaris versions 10 and 11 must add `/usr/local/bin` to their default path.

For macOS agents, symlinks aren't created until the first successful run that applies the agents' catalogs.

Tip: You can disable symlinks by changing the `manage_symlinks` setting in your default Hiera file:

```
puppet_enterprise::manage_symlinks: false
```

Binaries provided by other software components, such as those for interacting with the PostgreSQL server, PuppetDB, or Ruby packages, do not have symlinks created.

Modules and plugins installed

PE installs modules and plugins for normal operations.

Modules included with the software are installed on the master in `/opt/puppetlabs/puppet/modules`.

Don't modify anything in this directory or add modules of your own. Instead, install non-default modules in `/etc/puppetlabs/code/environments/<environment>/modules`.

Configuration files installed

PE installs configuration files that you might need to interact with from time to time.

On *nix nodes, configuration files live at `/etc/puppetlabs/puppet`.

On Windows nodes, configuration files live at <COMMON_APPDATA>\PuppetLabs. The location of this folder varies by Windows version; in 2008 and 2012, its default location is C:\ProgramData\PuppetLabs\puppet\etc.

The software's `confdir` is in the `puppet` subdirectory. This directory contains the `puppet.conf` file, `auth.conf`, and the SSL directory.

Tools installed

PE installs several suites of tools to help you work with the major components of the software.

- **Puppet tools** — Tools that control basic functions of the software such as `puppet master` and `puppet cert`.
- **Client tools** — The `pe-client-tools` package collects a set of CLI tools that extend the ability for you to access services from the master or a workstation. This package includes:
 - **Orchestrator** — The orchestrator is a set of interactive command line tools that provide the interface to the orchestration service. Orchestrator also enables you to enforce change on the environment level. Tools include `puppet job` and `puppet app`.
 - **Puppet Access** — Users can generate tokens to authenticate their access to certain command line tools and API endpoints.
 - **Code Manager CLI** — The `puppet-code` command allows you to trigger Code Manager from the command line to deploy your environments.
 - **PuppetDB CLI** — This is a tool for working with PuppetDB, including building queries and handling exports.
- **Module tool** — The module tool is used to access and create modules, which are reusable chunks of Puppet code users have written to automate configuration and deployment tasks. For more information, and to access modules, visit the Forge.
- **Console** — The console is the web user interface for PE. The console provides tools to view and edit resources on your nodes, view reports and activity graphs, and more.

Databases installed

PE installs several default databases, all of which use PostgreSQL as a database backend.

The PE PostgreSQL database includes these following databases.

Database	Description
pe-activity	Activity data from the classifier, including who, what, and when.
pe-classifier	Classification data, all node group information.
pe-puppetdb	PuppetDB data, including exported resources, catalogs, facts, and reports.
pe-rbac	RBAC data, including users, permissions, and AD/LDAP info.
pe-orchestrator	Orchestrator data, including details about job runs.

Use the native PostgreSQL tools to perform database exports and imports. At a minimum, perform backups to a remote system nightly, or as dictated by your company policy.

Services installed

PE installs several services used to interact with the software during normal operations.

Service	Definition
pe-console-services	Manages and serves the console.

Service	Definition
pe-puppetserver	Runs the master server, which manages the master component.
pe-nginx	Nginx, serves as a reverse-proxy to the console.
puppet	(on Enterprise Linux and Debian-based platforms) Runs the agent daemon on every agent node.
pe-puppetdb, pe-postgresql	Daemons that manage and serve the database components. The pe-postgresql service is created only if the software installs and manages PostgreSQL.
pxp-agent	Runs the Puppet Execution Protocol agent process.
pe-orchestration-services	Runs the orchestration process.

User and group accounts installed

These are the user and group accounts installed.

User	Definition
pe-puppet	Runs the master processes spawned by pe-puppetserver.
pe-webserver	Runs Nginx.
pe-puppetdb	Has root access to the database.
pe-postgres	Has access to the pe-postgreSQL instance. Created only if the software installs and manages PostgreSQL.
pe-console-services	Runs the console process.
pe-orchestration-services	Runs the orchestration process.

Log files installed

The software distributed with PE generates log files that you can collect for compliance or use for troubleshooting.

Master logs

The master has these logs.

- `/var/log/puppetlabs/puppetserver/code-manager-access.log`
- `/var/log/puppetlabs/puppetserver/file-sync-access.log`
- `/var/log/puppetlabs/puppetserver/masterhttp.log`
- `/var/log/puppetlabs/puppetserver/pcp-broker.log` — The log file for Puppet Communications Protocol brokers on compilers.
- `/var/log/puppetlabs/puppetserver/puppetserver.log`
- `/var/log/puppetlabs/puppetserver/puppetserver.log` — The master application logs its activity, including compilation errors and deprecation warnings, here.
- `/var/log/puppetlabs/puppetserver/puppetserver-access.log`
- `/var/log/puppetlabs/puppetserver/puppetserver-daemon.log` — This is where fatal errors or crash reports can be found.
- `/var/log/puppetlabs/puppetserver/puppetserver-status.log`

Agent logs

The locations of agent logs depend on the agent operating system.

On *nix nodes, the agent service logs its activity to the syslog service. Your syslog configuration dictates where these messages are saved, but the default location is `/var/log/messages` on Linux, `/var/log/system.log` on macOS, and `/var/adm/messages` on Solaris.

On Windows nodes, the agent service logs its activity to the event log. You can view its logs by browsing the event viewer.

Console and console services logs

The console and pe-console-services has these logs.

- `/var/log/puppetlabs/console-services/console-services.log`
- `/var/log/puppetlabs/console-services/console-services-api-access.log`
- `/var/log/puppetlabs/console-services-access.log`
- `/var/log/puppetlabs/console-services-daemon.log` — This is where fatal errors or crash reports can be found.
- `/var/log/puppetlabs/nginx/access.log`
- `/var/log/puppetlabs/nginx/error.log` — Contains errors related to nginx. Console errors that aren't logged elsewhere can be found in this log.

Installer logs

The installer has these logs.

- `/var/log/puppetlabs/installer/http.log` — Contains web requests sent to the installer. This log is present only on the machine from which a web-based installation was performed.
- `/var/log/puppetlabs/installer/install_log.lastrun.<hostname>.log` — Contains the contents of the last installer run.
- `/var/log/puppetlabs/installer/installer-<timestamp>.log` — Contains the operations performed and any errors that occurred during installation.

Database logs

The database has these logs.

- `/var/log/puppetlabs/postgresql/9.6/pgstartup.log`
- `/var/log/puppetlabs/postgresql/9.6/postgresql-Mon.log`
- `/var/log/puppetlabs/postgresql/9.6/postgresql-Tue.log`
- `/var/log/puppetlabs/postgresql/9.6/postgresql-Wed.log`
- `/var/log/puppetlabs/postgresql/9.6/postgresql-Thu.log`
- `/var/log/puppetlabs/postgresql/9.6/postgresql-Fri.log`
- `/var/log/puppetlabs/postgresql/9.6/postgresql-Sat.log`
- `/var/log/puppetlabs/postgresql/9.6/postgresql-Sun.log`
- `/var/log/puppetlabs/puppetdb/puppetdb.log`
- `/var/log/puppetlabs/puppetdb/puppetdb-access.log`
- `/var/log/puppetlabs/puppetdb/puppetdb-status.log`

Orchestration logs

The orchestration service and related components have these logs.

- `/var/log/puppetlabs/orchestration-services/aggregate-node-count.log`
- `/var/log/puppetlabs/orchestration-services/pcp-broker.log` — The log file for Puppet Communications Protocol brokers on the master.
- `/var/log/puppetlabs/orchestration-services/pcp-broker-access.log`
- `/var/log/puppetlabs/orchestration-services/orchestration-services.log`
- `/var/log/puppetlabs/orchestration-services/orchestration-services-access.log`

- `/var/log/puppetlabs/orchestration-services/orchestration-services-daemon.log`
— This is where fatal errors or crash reports can be found.
- `/var/log/puppetlabs/orchestration-services/orchestration-services-status.log`
- `/var/log/puppetlabs/pxp-agent/pxp-agent.log` (on *nix) or `C:\ProgramData\PuppetLabs/pxp-agent/var/log/pxp-agent.log` (on Windows) — Contains the Puppet Execution Protocol agent log file.
- `/var/log/puppetlabs/bolt-server/bolt-server.log` - Log file for PE Bolt server.
- `/var/log/puppetlabs/orchestration-services/orchestration-services.log` — Log file for the node inventory service.

Certificates installed

During installation, the software generates and installs a number of SSL certificates so that agents and services can authenticate themselves.

These certs can be found at `/etc/puppetlabs/puppet/ssl/certs`.

A certificate with the same name as the agent that runs on the master is generated during installation. This certificate is used by PuppetDB and the console.

Services that run on the master — for example, `pe-orchestration-services` and `pe-console-services` — use the agent certificate to authenticate.

Secret key file installed

During installation, the software generates a secret key file that is used to encrypt and decrypt sensitive data stored in the inventory service.

The secret key is stored at: `/etc/puppetlabs/orchestration-services/conf.d/secrets/keys.json`

Installing Puppet Enterprise

You can install PE using *express install*, which relies on defaults, or *text install*, where you provide a `pe.conf` file with installation parameters. Either of these methods is appropriate for installing infrastructure components on your master.

Related information

[Configuring high availability](#) on page 218

Enabling high availability for Puppet Enterprise ensures that your system remains operational even if certain infrastructure components become unreachable.

[Installing compilers](#) on page 165

As your Puppet Enterprise infrastructure scales up to 4,000 nodes and beyond, add load-balanced compilers to your monolithic installation to increase the number of agents you can manage.

[Installing external PostgreSQL](#) on page 173

By default, Puppet Enterprise includes its own database backend, PE-PostgreSQL, which is installed alongside PuppetDB. If the load on your PuppetDB node is larger than it can effectively scale to (greater than 20,000 nodes), you can install a standalone instance of PE-PostgreSQL.

Download and verify the installation package

PE is distributed in downloadable packages specific to supported operating system versions and architectures.

Installation packages include the full installation tarball and a GPG signature (`.asc`) file used to verify authenticity.

Before you begin

GnuPG is an open source program that allows you to safely encrypt and sign digital communications. You must have GnuPG installed to sign for the release key. Visit the [GnuPG website](#) for more information or to download GnuPG.

1. [Download](#) the tarball appropriate to your operating system and architecture.
2. Import the Puppet public key.

```
wget -O - https://downloads.puppetlabs.com/puppet-gpg-signing-key.pub |
gpg --import
```

3. Print the fingerprint of the key.

```
gpg --fingerprint 0x7F438280EF8D349F
```

The primary key fingerprint displays: 6F6B 1550 9CF8 E59E 6E46 9F32 7F43 8280 EF8D 349F.

4. Verify the release signature of the installation package.

```
$ gpg --verify puppet-enterprise-<version>-<platform>.tar.gz.asc
```

The result is similar to:

```
gpg: Signature made Tue 18 Sep 2016 10:05:25 AM PDT using RSA key ID
EF8D349F
gpg: Good signature from "Puppet, Inc. Release Key (Puppet, Inc. Release
Key)"
```

Note: If you don't have a trusted path to one of the signatures on the release key, you receive a warning that a valid path to the key couldn't be found.

Install using express install

Express installation uses default settings to install PE, so you don't have to edit a `pe.conf` file before or during installation. At the end of the installation process, you're prompted to provide a console administrator password, which is the only user-required value.

Log in as a root user to perform these steps.

1. Unpack the installation tarball:

```
tar -xf <TARBALL_FILENAME>
```

2. From the installer directory, run the installer:

```
sudo ./puppet-enterprise-installer
```

3. When prompted, select express install mode.
4. After installation completes, follow the prompts to specify a password, or run: `puppet infrastructure console_password --password=<MY_PASSWORD>`
5. Run Puppet twice: `puppet agent -t`.

You must restart the shell before you can use PE client tool subcommands.

Install using text install

When you run the installer in text mode, you provide a configuration file (`pe.conf`) to the installer. The `pe.conf` file contains values for the parameters needed for installation.

Perform these steps logged in as root.

1. Unpack the installation tarball:

```
tar -xf <TARBALL_FILENAME>
```


2. From the installer directory, run the installer. The installation steps vary depending on the method you choose.

- To use a `pe.conf` file that you've previously populated, run the installer **with the `-c` flag** pointed at the `pe.conf` file.:

```
sudo ./puppet-enterprise-installer -c <FULL PATH TO pe.conf>
```

- To have the installer open a copy of `pe.conf` for you to edit and install with, run the installer **without the `-c` flag**:

```
sudo ./puppet-enterprise-installer
```

- Select **text-mode** when prompted.
 - Specify required installation parameters. If you're installing with specialized configuration, such as external PostgreSQL or an independent intermediate certificate authority, add parameters required for those configurations.
 - Save and close the file. Installation begins.
3. After installation completes, if you didn't specify a console admin password in your `pe.conf` file, follow the prompts to specify a password, or run: `puppet infrastructure console_password --password=<MY_PASSWORD>`
4. Run Puppet twice: `puppet agent -t`.

You must restart the shell before you can use PE client tool subcommands.

Related information

[Use an independent intermediate certificate authority](#) on page 693

The built-in Puppet certificate authority automatically generates a root and intermediate certificate, but you can set up an independent intermediate certificate authority during installation if you need additional intermediate certificates, or if you prefer to use a public authority CA.

[Text install options](#) on page 137

When you run the installer in text mode, you can use the `-c` option to specify the full path to an existing `pe.conf` file. You can pair these additional options with the `-c` option.

[Configuration parameters and the `pe.conf` file](#) on page 138

A `pe.conf` file is a HOCON formatted file that declares parameters and values needed to install, upgrade, and configure PE.

[External PostgreSQL parameters](#) on page 141

These parameters are required to install an external PostgreSQL instance. Password parameters can be added to standard installations if needed.

[Executable binaries and symlinks installed](#) on page 131

PE installs executable binaries and symlinks for interacting with tools and services.

Text install options

When you run the installer in text mode, you can use the `-c` option to specify the full path to an existing `pe.conf` file. You can pair these additional options with the `-c` option.

Option	Definition
<code>-D</code>	Display debugging information
<code>-q</code>	Run in quiet mode. The installation process isn't displayed. If errors occur during the installation, the command quits with an error message.
<code>-Y</code>	Run automatically using the <code>pe.conf</code> file at <code>/etc/puppetlabs/enterprise/conf.d/</code> . If the file is not present or is invalid, installation or upgrade fails.
<code>-V</code>	Display verbose debugging information.

Option	Definition
-h	Display help information.
force	For upgrades only, bypass PostgreSQL migration validation. This option must appear last, after the end-of-options signifier (--), for example <code>sudo ./puppet-enterprise-installer -c pe.conf -- --force</code>

Configuration parameters and the `pe.conf` file

A `pe.conf` file is a HOCON formatted file that declares parameters and values needed to install, upgrade, and configure PE.

You can create or obtain a `pe.conf` file by:

- Using the example `pe.conf` file provided in the `conf.d` directory in the installer tarball.
- Tip:** In most cases, you can use the example `pe.conf` file without making any changes.
- Selecting the text-mode installation option when prompted by the installer. This option opens your default text editor with the example `pe.conf` file, which you can modify as needed. Installation proceeds using that `pe.conf` after you quit the editor.
- Using the web-based installer to create a `pe.conf` file. After you run the web-based installer, you can find the file at `/etc/puppetlabs/enterprise/conf.d`. You can also download the file by following the link provided on the confirmation page of the web-based installer.

The following are examples of valid parameter and value expressions:

Type	Value
FQDNs	<code>"puppet_enterprise::puppet_master_host": "master.example.com"</code>
Strings	<code>"console_admin_password": "mypassword"</code>
Arrays	<code>["puppet", "puppetlb-01.example.com"]</code>
Booleans	<code>"puppet_enterprise::profile::orchestrator::run_sensu": true</code> Valid Boolean values are <code>true</code> or <code>false</code> (case sensitive, no quotation marks). Note: Don't use <code>Yes</code> (<code>y</code>), <code>No</code> (<code>n</code>), <code>1</code> , or <code>0</code> .
JSON hashes	<code>"puppet_enterprise::profile::orchestrator::java_opts": { "Xmx": "256m", "Xms": "256m" }</code>
Integer	<code>"puppet_enterprise::profile::console::rbac_session_timeout": 60</code>

Important: Don't use single quotes on parameter values. Use double quotes as shown in the examples.

Installation parameters

These parameters are required for installation.

Tip: To simplify installation, you can keep the default value of `%{::trusted.certname}` for your master and provide a console administrator password after running the installer.

`puppet_enterprise::puppet_master_host`

The FQDN of the node hosting the master, for example `master.example.com`.

Default: `%{::trusted.certname}`

Database configuration parameters

These are the default parameters and values supplied for the PE databases.

This list is intended for reference only; don't change or customize these parameters.

puppet_enterprise::activity_database_name

Name for the activity database.

Default: `pe-activity`

puppet_enterprise::activity_database_read_user

Activity database user that can perform only read functions.

Default: `pe-activity-read`

puppet_enterprise::activity_database_write_user

Activity database user that can perform only read and write functions.

Default: `pe-activity-write`

puppet_enterprise::activity_database_super_user

Activity database superuser.

Default: `pe-activity`

puppet_enterprise::activity_service_migration_db_user

Activity service database user used for migrations.

Default: `pe-activity`

puppet_enterprise::activity_service_regular_db_user

Activity service database user used for normal operations.

Default: `pe-activity-write`

puppet_enterprise::classifier_database_name

Name for the classifier database.

Default: `pe-classifier`

puppet_enterprise::classifier_database_read_user

Classifier database user that can perform only read functions.

Default: `pe-classifier-read`

puppet_enterprise::classifier_database_write_user

Classifier database user that can perform only read and write functions.

`pe-classifier-write`

puppet_enterprise::classifier_database_super_user

Classifier database superuser.

`pe-classifier`

puppet_enterprise::classifier_service_migration_db_user

Classifier service user used for migrations.

Default: `pe-classifier`

puppet_enterprise::classifier_service_regular_db_user

Classifier service user used for normal operations.

Default: `pe-classifier-write`

puppet_enterprise::orchestrator_database_name

Name for the orchestrator database.

Default: pe-orchestrator

puppet_enterprise::orchestrator_database_read_user

Orchestrator database user that can perform only read functions.

Default: pe-orchestrator-read

puppet_enterprise::orchestrator_database_write_user

Orchestrator database user that can perform only read and write functions.

Default: pe-orchestrator-write

puppet_enterprise::orchestrator_database_super_user

Orchestrator database superuser.

Default: pe-orchestrator

puppet_enterprise::orchestrator_service_migration_db_user

Orchestrator service user used for migrations.

Default: pe-orchestrator

puppet_enterprise::orchestrator_service_regular_db_user

Orchestrator service user used for normal operations.

Default: pe-orchestrator-write

puppet_enterprise::puppetdb_database_name

Name for the PuppetDB database.

Default: pe-puppetdb

puppet_enterprise::rbac_database_name

Name for the RBAC database.

Default: pe-rbac

puppet_enterprise::rbac_database_read_user

RBAC database user that can perform only read functions.

Default: pe-rbac-read

puppet_enterprise::rbac_database_write_user

RBAC database user that can perform only read and write functions.

Default: pe-rbac-write

puppet_enterprise::rbac_database_super_user

RBAC database superuser.

Default: pe-rbac

puppet_enterprise::rbac_service_migration_db_user

RBAC service user used for migrations.

pe-rbac

puppet_enterprise::rbac_service_regular_db_user

RBAC service user used for normal operations.

Default: pe-rbac-write

External PostgreSQL parameters

These parameters are required to install an external PostgreSQL instance. Password parameters can be added to standard installations if needed.

puppet_enterprise::database_host

Agent certname of the node hosting the database component. Don't use an alt name for this value.

puppet_enterprise::database_port

The port that the database is running on.

Default: 5432

puppet_enterprise::database_ssl

true or false. For unmanaged PostgreSQL installations don't use SSL security, set this parameter to false.

Default: true

puppet_enterprise::database_cert_auth

true or false.

Important: For unmanaged PostgreSQL installations don't use SSL security, set this parameter to false.

Default: true

puppet_enterprise::puppetdb_database_password

Password for the PuppetDB database user. Must be a string, such as "mypassword".

puppet_enterprise::classifier_database_password

Password for the classifier database user. Must be a string, such as "mypassword".

puppet_enterprise::classifier_service_regular_db_user

Database user the classifier service uses for normal operations.

Default: pe-classifier

puppet_enterprise::classifier_service_migration_db_user

Database user the classifier service uses for migrations.

Default: pe-classifier

puppet_enterprise::activity_database_password

Password for the activity database user. Must be a string, such as "mypassword".

puppet_enterprise::activity_service_regular_db_user

Database user the activity service uses for normal operations.

Default: "pe-activity"

puppet_enterprise::activity_service_migration_db_user

Database user the activity service uses for migrations.

Default: pe-activity"

puppet_enterprise::rbac_database_password

Password for the RBAC database user. Must be a string, such as "mypassword".

puppet_enterprise::rbac_service_regular_db_user

Database user the RBAC service uses for normal operations.

Default: "pe-rbac"

puppet_enterprise::rbac_service_migration_db_user

Database user the RBAC service uses for migrations.

Default: "pe-rbac"

puppet_enterprise::orchestrator_database_password

Password for the orchestrator database user. Must be a string, such as "mypassword".

puppet_enterprise::orchestrator_service_regular_db_user

Database user the orchestrator service uses for normal operations.

Default: pe-orchestrator

puppet_enterprise::orchestrator_service_migration_db_user

Database user the orchestrator service uses for migrations.

Default: "pe-orchestrator"

Master parameters

Use these parameters to configure and tune the master.

pe_install::puppet_master_dnsaltnames

An array of strings that represent the DNS altnames to be added to the SSL certificate generated for the master.

Default: ["puppet"]

puppet_enterprise::profile::certificate_authority

Array of additional certificates to be allowed access to the /certificate_statusAPI endpoint. This list is added to the base certificate list.

puppet_enterprise::profile::master::code_manager_auto_configure

true to automatically configure the Code Manager service, or false.

puppet_enterprise::profile::master::r10k_remote

String that represents the Git URL to be passed to the r10k.yaml file, for example "git@your.git.server.com:puppet/control.git". The URL can be any URL that's supported by r10k and Git. This parameter is required only if you want r10k configured when PE is installed; it must be specified in conjunction with puppet_enterprise::profile::master::r10k_private_key.

puppet_enterprise::profile::master::r10k_private_key

String that represents the local file system path on the master where the SSH private key can be found and used by r10k, for example "/etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa". This parameter is required only if you want r10k configured when PE is installed; it must be specified in conjunction with puppet_enterprise::profile::master::r10k_remote.

puppet_enterprise::profile::master::check_for_updates

true to check for updates whenever the pe-puppetserver service restarts, or false.

Default: true

Console and console-services parameters

Use these parameters to customize the behavior of the console and console-services. Parameters that begin with puppet_enterprise::profile can be modified from the console itself. See the configuration methods documents for more information on how to change parameters in the console or Hiera.

puppet_enterprise::profile::console::classifier_synchronization_period

Integer representing, in seconds, the classifier synchronization period, which controls how long it takes the node classifier to retrieve classes from the master.

Default: "600" (seconds).

puppet_enterprise::profile::console::rbac_failed_attempts_lockout

Integer specifying how many failed login attempts are allowed on an account before that account is revoked.

Default: "10" (attempts).

puppet_enterprise::profile::console::rbac_password_reset_expiration

Integer representing, in hours, how long a user's generated token is valid for. An administrator generates this token for a user so that they can reset their password.

Default: "24" (hours).

puppet_enterprise::profile::console::rbac_session_timeout

Integer representing, in minutes, how long a user's session can last. The session length is the same for node classification, RBAC, and the console.

Default: "60" (minutes).

puppet_enterprise::profile::console::session_maximum_lifetime

Integer representing the maximum allowable period that a console session can be valid. To not expire before the maximum token lifetime, set to '0'.

Supported units are "s" (seconds), "m" (minutes), "h" (hours), "d" (days), "y" (years). Units are specified as a single letter following an integer, for example "1d"(1 day). If no units are specified, the integer is treated as seconds.

puppet_enterprise::profile::console::console_ssl_listen_port

Integer representing the port that the console is available on.

Default: [443]

puppet_enterprise::profile::console::ssl_listen_address

Nginx listen address for the console.

Default: "0.0.0.0"

puppet_enterprise::profile::console::classifier_prune_threshold

Integer representing the number of days to wait before pruning the size of the classifier database. If you set the value to "0", the node classifier service is never pruned.

puppet_enterprise::profile::console::classifier_node_check_in_storage

"true" to store an explanation of how nodes match each group they're classified into, or "false".

Default: "false"

puppet_enterprise::profile::console::display_local_time

"true" to display timestamps in local time, with hover text showing UTC time, or "false" to show timestamps in UTC time.

Default: "false"

Modify these configuration parameters in `Hiera` or `pe.conf`, not the console:

puppet_enterprise::api_port

SSL port that the node classifier is served on.

Default: [4433]

puppet_enterprise::console_services::no_longer_reporting_cutoff

Length of time, in seconds, before a node is considered unresponsive.

Default: 3600 (seconds)

console_admin_password

The password to log into the console, for example "myconsolepassword".

Default: Specified during installation.

Orchestrator and orchestration services parameters

Use these parameters to configure and tune the orchestrator and orchestration services.

puppet_enterprise::profile::agent::pxp_enabled

true to enable the Puppet Execution Protocol service, which is required to use the orchestrator and run Puppet from the console, or false.

Default: true

puppet_enterprise::profile::bolt_server::concurrency

An integer that determines the maximum number of concurrent requests orchestrator can make to bolt-server.



CAUTION: Do not set a concurrency limit that is higher than the bolt-server limit. This can cause timeouts that lead to failed task runs.

Default: The default value is set to the current value stored for bolt-server.

puppet_enterprise::profile::orchestrator::global_concurrent_compiles

Integer representing how many concurrent compile requests can be outstanding to the master, across all orchestrator jobs.

Default: "8" requests

puppet_enterprise::profile::orchestrator::job_prune_threshold

Integer representing the days after which job reports should be removed.

Default: "30" days

puppet_enterprise::profile::orchestrator::pcp_timeout

Integer representing the length of time, in seconds, before timeout when agents attempt to connect to the Puppet Communications Protocol broker in a Puppet run triggered by the orchestrator.

Default: "30" seconds

puppet_enterprise::profile::orchestrator::run_service

true to enable orchestration services, or false.

Default: true

puppet_enterprise::profile::orchestrator::task_concurrency

Integer representing the number of tasks that can run at the same time.

Default: "250" tasks

puppet_enterprise::profile::orchestrator::use_application_services

true to enable application management, or false.

Default: false

puppet_enterprise::pxp_agent::ping_interval

Integer representing the interval, in seconds, between agents' attempts to ping Puppet Communications Protocol brokers.

Default: "120" seconds

puppet_enterprise::pxp_agent::pxp_logfile

String representing the path to the Puppet Execution Protocol agent log file. Change as needed.

Default: /var/log/puppetlabs/pxp-agent/pxp-agent.log (*nix) or C:/ProgramData/PuppetLabs/pxp-agent/var/log/pxp-agent.log (Windows)

Related information

[Configuring Puppet orchestrator](#) on page 421

Once you've installed PE or the client tools package, there are a few tasks you need to do to prepare your PE infrastructure for orchestration services.

PuppetDB parameters

Use these parameters to configure and tune PuppetDB.

puppet_enterprise::puppetdb::command_processing_threads

Integer representing how many command processing threads PuppetDB uses to sort incoming data. Each thread can process a single command at a time.

Default: Half the number of cores in your system, for example "8".

puppet_enterprise::profile::master::puppetdb_report_processor_ensure

present to generate agent run reports and submit them to PuppetDB, or absent

Default: present

puppet_enterprise::puppetdb_port

Integer in brackets representing the SSL port that PuppetDB listens on.

Default: "[8081]"

puppet_enterprise::profile::puppetdb::node_purge_ttl

"Time-to-live" value before deactivated or expired nodes are deleted, along with all facts, catalogs, and reports for the node. For example, a value of "14d" sets the time-to-live to 14 days.

Default: "14d"

Java parameters

Use these parameters to configure and tune Java.

puppet_enterprise::profile::master::java_args

JVM (Java Virtual Machine) memory, specified as a JSON hash, that is allocated to the Puppet Server service, for example {"Xmx": "4096m", "Xms": "4096m"}.

puppet_enterprise::profile::puppetdb::java_args

JVM memory, specified as a JSON hash, that is allocated to the PuppetDB service, for example {"Xmx": "512m", "Xms": "512m"}.

puppet_enterprise::profile::console::java_args

JVM memory, specified as a JSON hash, that is allocated to console services, for example {"Xmx": "512m", "Xms": "512m"}.

puppet_enterprise::profile::orchestrator::java_args

JVM memory, set as a JSON hash, that is allocated to orchestration services, for example, {"Xmx": "256m", "Xms": "256m"}.

Purchasing and installing a license key

Your license must support the number of nodes that you want to manage with Puppet Enterprise.

Complimentary license

You can manage up to 10 nodes at no charge, and no license key is needed. When you have 11 or more active nodes and no license key, license warnings appear in the console until you install an appropriate license key.

Purchased license

To manage 11 or more active nodes, you must purchase a license. After you purchase a license and install a license key file, your licensed node count and subscription expiration date appear on the **License** page.

Note: To support spikes in node usage, four days per calendar month you can exceed your licensed node count up to double the number of nodes you purchased. This increased number of nodes is called your *bursting limit*. You must buy more nodes for your license if you exceed the licensed node count within the bursting limit on more than four days per calendar month, or if you exceed your bursting limit at all. In these cases, license warnings appear in the console until you contact your Puppet representative.

Related information

[How nodes are counted](#) on page 319

Your *node count* is the number of nodes in your inventory. Your license limits you to a certain number of active nodes before you hit your *bursting limit*. If you hit your bursting limit on four days during a month, you must purchase a license for more nodes or remove some nodes from your inventory.

Getting a license

Contact Puppet to purchase a license subscription.

- New PE customers — Purchase a subscription from the Puppet website, or contact our sales team. Find more information at [Ready to buy Puppet Enterprise?](#)
- Existing PE customers — To add nodes, upgrade, or renew your annual license, contact your sales representative or email sales@puppet.com.

Tip: To reduce your active node count and free up licenses, remove inactive nodes from your deployment. By default, nodes with Puppet agents are automatically deactivated after seven days with no activity (no new facts, catalog, or reports).

Related information

[Remove agent nodes](#) on page 315

If you no longer wish to manage an agent node, you can remove it and make its license available for another node.

Install a license key

Install the `license.key` file to upgrade from a test installation to an active installation.

1. Install the license key by copying the file to `/etc/puppetlabs/license.key` on the master node.
2. Verify that Puppet has permission to read the license key by checking its ownership and permissions: `ls -la /etc/puppetlabs/license.key`
3. If the ownership is not `root` and permissions are not `-rw-r--r--` (octal 644), set them:

```
sudo chown root:root /etc/puppetlabs/license.key
sudo chmod 644 /etc/puppetlabs/license.key
```

View license details for your environment

Check the number of active nodes in your deployment, the number of licensed nodes you purchased, and the expiration date for your license.

Procedure

- In the console, click **License**.

The **License** page opens with information on your licensed nodes, bursting limit, and subscription expiration date. Any license warnings that appear in the console navigation are explained here. For example, if your license is expired or out of compliance.

Related information

[Remove agent nodes](#) on page 315

If you no longer wish to manage an agent node, you can remove it and make its license available for another node.

[Puppet orchestrator API: usage endpoint](#) on page 548

Use the `/usage` endpoint to view details about the active nodes in your deployment.

Installing agents

You can install Puppet Enterprise agents on `*nix`, Windows, and macOS.

The master hosts a package repo used to install agents in your infrastructure.

The PE package management repo is created during installation of the master and serves packages over HTTPS using the same port as the master (8140). This means agents don't require any new ports to be open other than the one they already need to communicate with the master.

After installing agents, you must sign their certificates. For details, see [Managing certificate signing requests](#) on page 54.

Using the install script

The install script installs and configures the agent on target nodes using installation packages from the PE package management repo.

The agent install script performs these actions:

- Detects the operating system on which it's running, sets up an apt, yum, or zipper repo that refers back to the master, and then pulls down and installs the `puppet-agent` packages. If the install script can't find agent packages corresponding to the agent's platform, it fails with an error telling you which `pe_repo` class you need to add to the master.
- For `*nix` agents, downloads a tarball of plugins from the master. This feature is controlled by the setting `pe_repo::enable_bulk_pluginsync`, which you can configure in `Hiera` or in the console. Bulk plugin sync is set to `true` (enabled) by default.
- Creates a basic `puppet.conf` file.
- Kicks off a Puppet run.

Automatic downloading of agent installer packages and plugins using the `pe_repo` class requires an internet connection.

Tip: If your master uses a proxy server to access the internet, prior to installation, specify `pe_repo::http_proxy_host` and `pe_repo::http_proxy_port` in `pe.conf`, `Hiera`, or in the console, in the `pe_repo` class of the **PE Master** node group.

You can customize agent installation by providing as flags to the script any number of these options, in any order:

Option	Example	Result
puppet.conf settings	<pre>agent:splay=true agent:certname=node1.corp.net agent:environment=development</pre>	<p>The puppet.conf file looks like this:</p> <pre>[agent] certname = node1.corp.net splay = true environment = development</pre>
CSR attribute settings	<pre>extension_requests:pp_role=webserver custom_attributes:challengePassword=abc123</pre>	<p>The installer creates a <code>csr_attributes.yaml</code> file before installing with this content:</p> <pre>--- custom_attributes: challengePassword: abc123 extension_requests: pp_role: webserver</pre>
MSI properties (Windows only)	<pre>-PuppetAgentAccountUser 'pup_adm' - PuppetAgentAccountPassword 'secret'</pre>	<p>The Puppet service runs as <code>pup_adm</code> with a password of <code>secret</code>.</p>
Puppet service status	<pre>*nix: --puppet-service-ensure stopped --puppet-service-enable false Windows -PuppetServiceEnsure stopped -PuppetServiceEnable false</pre>	<p>The Puppet service is stopped and doesn't boot after installation. An initial Puppet run doesn't occur after installation.</p>

puppet.conf settings

You can specify any agent configuration option using the install script. Configuration settings are added to `puppet.conf`.

These are the most commonly specified agent config options:

- server
- certname
- environment
- splay
- splaylimit

- noop

Tip: On Enterprise Linux systems, if you have a proxy between the agent and the master, you can specify `http_proxy_host`, for example `-s agent:http_proxy_host=<PROXY_FQDN>`.

See the [Configuration Reference](#) for details.

CSR attribute settings

These settings are added to `puppet.conf` and included in the `custom_attributes` and `extension_requests` sections of `csr_attributes.yaml`.

You can pass as many parameters as needed. Follow the `section:key=value` pattern and leave one space between parameters.

See the [csr_attributes.yaml](#) reference for details.

*nix install script with example agent setup and certificate signing parameters:

```
curl -k https://master.example.com:8140/packages/current/
install.bash | sudo bash -s agent:certname=<certnameOtherThanFQDN>
custom_attributes:challengePassword=<passwordForAutosignerScript>
extension_requests:pp_role=<puppetNodeRole>
```

Windows install script with example agent setup and certificate signing parameters:

```
[Net.ServicePointManager]::ServerCertificateValidationCallback
= {$true}; $webClient = New-Object System.Net.WebClient;
$webClient.DownloadFile('https://<PUPPET MASTER
FQDN>:8140/packages/current/install.ps1', 'install.ps1'); .
\install.ps1 agent:certname=<certnameOtherThanFQDN>
custom_attributes:challengePassword=<passwordForAutosignerScript>
extension_requests:pp_role=<puppetNodeRole>
```

MSI properties (Windows)

For Windows, you can set these MSI properties, with or without additional agent configuration settings.

MSI Property	PowerShell flag
INSTALLDIR	-InstallDir
PUPPET_AGENT_ACCOUNT_USER	-PuppetAgentAccountUser
PUPPET_AGENT_ACCOUNT_PASSWORD	-PuppetAgentAccountPassword
PUPPET_AGENT_ACCOUNT_DOMAIN	-PuppetAgentAccountDomain

Windows install script with MSI properties and agent configuration settings:

```
[Net.ServicePointManager]::ServerCertificateValidationCallback
= {$true}; $webClient = New-Object System.Net.WebClient;
$webClient.DownloadFile('https://<MASTER HOSTNAME>:8140/packages/current/
install.ps1', 'install.ps1'); .\install.ps1 -PuppetAgentAccountUser
"svcPuppet" -PuppetAgentAccountPassword "s3kr3t_P@ssword" agent:splay=true
agent:environment=development
```

Puppet service status

By default, the install script starts the Puppet agent service and kicks off a Puppet run. If you want to manually trigger a Puppet run, or you're using a provisioning system that requires non-default behavior, you can control whether the service is running and enabled.

Option	*nix	Windows	Values
ensure	--puppet-service-ensure <VALUE>	- PuppetServiceEnsure <VALUE>	<ul style="list-style-type: none"> • running • stopped
enable	--puppet-service-enable <VALUE>	- PuppetServiceEnable <VALUE>	<ul style="list-style-type: none"> • true • false • manual (Windows only) • mask

For example:

*nix

```
curl -k https://master.example.com:8140/packages/current/install.bash | sudo
bash -s -- --puppet-service-ensure stopped
```

Windows

```
[Net.ServicePointManager]::ServerCertificateValidationCallback
= {$true}; $webClient = New-Object System.Net.WebClient;
$webClient.DownloadFile('https://<PUPPET MASTER FQDN>:8140/packages/
current/install.ps1', 'install.ps1'); .\install.ps1 -PuppetServiceEnsure
stopped
```

Install *nix agents

PE has package management tools to help you easily install and configure agents. You can also use standard *nix package management tools.

Note:

You must enable TLSv1 to install agents on these platforms:

-
- 5
- 5
- SLES 11
- 10
- Server 2008r2

Related information

[Enable TLSv1](#) on page 697

TLSv1 and TLSv1.1 are disabled by default in PE.

Install agents from the console

You can use the console to leverage tasks that install *nix or Windows agents on target nodes.

Before you begin

Add classes to the **PE Master** node group for each agent platform used in your environment. For example, `pe_repo::platform::el_7_x86_64`.

Make sure you have permission to run the appropriate task to install agents on all nodes:

- *nix targets use the task `pe_bootstrap::linux`
- Windows targets use the task `pe_bootstrap::windows`

For Windows targets, this task requires:

- Windows 2008 SP2 or newer
- PowerShell version 3 or higher
- Microsoft .NET Framework 4.5 or higher

1. In the console, click **Inventory**.
2. Select the node type **Nodes with Puppet agents**.
3. Select a transport method.
 - **SSH** for *nix targets
 - **WinRM** for Windows targets
4. Enter the target host names and the credentials required to access them. If you use an SSH key, include begin and end tags.
5. Click **Add nodes**.

Tip: Click **Installation job started** to view the job details for the task.

Agents are installed on the target nodes and then automatically submit certificate signing requests (CSR) to the master. The list of unsigned certificates is updated with new targets.

Related information

[Managing certificate signing requests in the console](#) on page 54

Open certificate signing requests appear in the console on the **Unsigned certs** page. Accept or reject submitted requests individually or in a batch.

[Managing certificate signing requests on the command line](#) on page 54

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

Install *nix agents with PE package management

PE provides its own package management to help you install agents in your infrastructure.

Note: The <MASTER HOSTNAME> portion of the installer script—as provided in the following example—refers to the FQDN of the master. The FQDN must be fully resolvable by the machine on which you're installing or upgrading the agent.

1. If you're installing an agent with a different OS than the master, add the appropriate class for the repo that contains the agent packages.
 - a) In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Master** group.
 - b) On the **Configuration** tab in the **Class name** field, enter `pe_repo` and select the repo class from the list of classes.

Note: The repo classes are listed as

`pe_repo::platform::<AGENT_OS_VERSION_ARCHITECTURE>`.

- c) Click **Add class**, and commit changes.
- d) Run `puppet agent -t` to configure the master node using the newly assigned class.

The new repo is created in `/opt/puppetlabs/server/data/packages/public/<PE VERSION>/<PLATFORM>/`.

2. SSH into the node where you want to install the agent, and run the installation command appropriate to your environment.

- Curl

```
curl -k https://<MASTER_HOSTNAME>:8140/packages/current/install.bash |
sudo bash
```

Tip: On versions 7.10 and earlier, which don't support the `-k` option, use `--tlsv1` instead. If neither `-k` or `--tlsv1` is supported, you must install using a manually transferred certificate.

- 5 and 5

```
curl -k --tlsv1 https://>MASTER_HOSTNAME>:8140/packages/current/
install.bash | sudo bash
```

- wget

```
wget -O - -q --no-check-certificate https://<MASTER_HOSTNAME>:8140/
packages/current/install.bash | sudo bash
```

- 10 (run as root)

```
export PATH=$PATH:/opt/sfw/bin
wget -O - -q --no-check-certificate --secure-protocol=TLSv1 https://
<MASTER_HOSTNAME>:8140/packages/current/install.bash | bash
```

Install *nix agents with your own package management

If you choose not to use PE package management to install agents, you can use your own package management tools.

Before you begin

[Download](#) the appropriate agent tarball.

Agent packages can be found on the master in `/opt/puppetlabs/server/data/packages/public/<PE_VERSION>/`. This directory contains the platform specific repository file structure for agent packages. For example, if your master is running on CentOS 7, in `/opt/puppetlabs/server/data/packages/public/<PE_VERSION>/`, there's a directory `el-7-x86_64`, which contains the directories with all the packages needed to install an agent.

If your nodes are running an operating system or architecture that is different from the master, download the appropriate agent package, extract the agent packages into the appropriate repo, and then install the agents on your nodes just as you would any other package, for example `yum install puppet-agent`.

1. Add the agent package to your own package management and distribution system.
2. Configure the package manager on your agent node (Yum, Apt) to point to that repo.
3. Install the agent using the command appropriate to your environment.

- Yum

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

- Apt

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

4. Configure the agent as needed: `puppet config set`

Install *nix agents using a manually transferred certificate

If you choose not to or can't use `curl -k` to trust the master during agent installation, you can manually transfer the master CA certificate to any machines you want to install agents on, and then run the installation script against that cert.

1. On the machine that you're installing the agent on, create the directory `/etc/puppetlabs/puppet/ssl/certs/`.
2. On the master, navigate to `/etc/puppetlabs/puppet/ssl/certs/` and transfer `ca.pem` to the certs directory you created on the agent node.
3. On the agent node, verify file permissions: `chmod 444 /etc/puppetlabs/puppet/ssl/certs/ca.pem`
4. Run the installation command, using the `--cacert` flag to point to the cert:

```
curl --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem https://<MASTER
HOSTNAME>:8140/packages/current/install.bash | sudo bash
```

Install *nix agents without internet access

If you don't have access to the internet beyond your infrastructure, you can download the appropriate agent tarball from an internet-connected system and then install using the package management solution of your choice.

Before you begin

[Download](#) the appropriate agent tarball.

Install *nix agents with PE package management without internet access

Use PE package management to install agents when you don't have internet access beyond your infrastructure.

Note: You must repeat this process each time you upgrade your master.

1. On your master, copy the agent tarball to `/opt/puppetlabs/server/data/staging/pe_repo-puppet-agent-<AGENT_VERSION>`, for example `/opt/puppetlabs/server/data/staging/pe_repo-puppet-agent-1.10.4`.
2. If you're installing an agent with a different OS than the master, add the appropriate class for the repo that contains the agent packages.
 - a) In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Master** group.
 - b) On the **Configuration** tab in the **Class name** field, enter `pe_repo` and select the repo class from the list of classes.

Note: The repo classes are listed as `pe_repo::platform::<AGENT_OS_VERSION_ARCHITECTURE>`.
 - c) Click **Add class**, and commit changes.
3. Run Puppet: `puppet agent -t`
4. Follow the steps for [Install *nix agents with PE package management](#) on page 151.

Install *nix agents with your own package management without internet access

Use your own package management to install agents when you don't have internet access beyond your infrastructure.

Note: You must repeat this process each time you upgrade your master.

1. Add the agent package to your own package management and distribution system.
2. Disable the PE-hosted repo.
 - a) In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Master** group.
 - b) On the **Configuration** tab, find `pe_repo` class (as well as any class that begins `pe_repo::`), and click **Remove this class**.
 - c) Commit changes.

Install *nix agents from compilers using your own package management without internet access

If your infrastructure relies on compilers to install agents, you don't have to copy the agent package to each compiler. Instead, use the console to specify a path to the agent package on your package management server.

1. Add the agent package to your own package management and distribution system.
2. Set the `base_path` parameter of the `pe_repo` class to your package management server.
 - a) In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Master** group.
 - b) On the **Configuration** tab, find the `pe_repo` class and specify the parameter.

Parameter	Value
<code>base_path</code>	FQDN of your package management server.

- c) Click **Add parameter** and commit changes.

Install Windows agents

You can install Windows agents with PE package management, with a manually transferred certificate, or with the Windows .msi package.

Note:

You must enable TLSv1 to install agents on these platforms:

-
- 5
- 5
- SLES 11
- 10
- Server 2008r2

Related information

[Enable TLSv1](#) on page 697

TLSv1 and TLSv1.1 are disabled by default in PE.

Install agents from the console

You can use the console to leverage tasks that install *nix or Windows agents on target nodes.

Before you begin

Add classes to the **PE Master** node group for each agent platform used in your environment. For example, `pe_repo::platform::el_7_x86_64`.

Make sure you have permission to run the appropriate task to install agents on all nodes:

- *nix targets use the task `pe_bootstrap::linux`
- Windows targets use the task `pe_bootstrap::windows`

For Windows targets, this task requires:

- Windows 2008 SP2 or newer
- PowerShell version 3 or higher
- Microsoft .NET Framework 4.5 or higher

1. In the console, click **Inventory**.
2. Select the node type **Nodes with Puppet agents**.
3. Select a transport method.
 - **SSH** for *nix targets
 - **WinRM** for Windows targets

4. Enter the target host names and the credentials required to access them. If you use an SSH key, include `begin` and `end` tags.
5. Click **Add nodes**.

Tip: Click **Installation job started** to view the job details for the task.

Agents are installed on the target nodes and then automatically submit certificate signing requests (CSR) to the master. The list of unsigned certificates is updated with new targets.

Related information

[Managing certificate signing requests in the console](#) on page 54

Open certificate signing requests appear in the console on the **Unsigned certs** page. Accept or reject submitted requests individually or in a batch.

[Managing certificate signing requests on the command line](#) on page 54

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

Install Windows agents with PE package management

To install a Windows agent with PE package management, you use the `pe_repo` class to distribute an installation package to agents. You can use this method with or without internet access.

Before you begin

If your master doesn't have internet access, [download](#) the appropriate agent package and save it on your master in the location appropriate for your agent systems:

- 32-bit systems — `/opt/puppetlabs/server/data/packages/public/<PE_VERSION>/windows-i386-<AGENT_VERSION>/`
- 64-bit systems — `/opt/puppetlabs/server/data/packages/public/<PE_VERSION>/windows-x86_64-<AGENT_VERSION>/`

You must use PowerShell 2.0 or later to install Windows agents with PE package management.

Note: The `<MASTER_HOSTNAME>` portion of the installer script—as provided in the following example—refers to the FQDN of the master. The FQDN must be fully resolvable by the machine on which you're installing or upgrading the agent.

1. In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Master** group.
2. On the **Configuration** tab in the **Class name** field, select `pe_repo` and select the appropriate repo class from the list of classes.
 - 64-bit (x86_64) — `pe_repo::platform::windows_x86_64`.
 - 32-bit (i386) — `pe_repo::platform::windows_i386`.
3. Click **Add class** and commit changes.
4. On the master, run Puppet to configure the newly assigned class.

The new repository is created on the master at `/opt/puppetlabs/server/data/packages/public/<PE_VERSION>/<PLATFORM>/`.

5. On the node, open an administrative PowerShell window, and install:

```
[System.Net.ServicePointManager]::SecurityProtocol =
[Net.SecurityProtocolType]::Tls12;
[Net.ServicePointManager]::ServerCertificateValidationCallback
= {$true}; $webClient = New-Object System.Net.WebClient;
$webClient.DownloadFile('https://<MASTER_HOST>:8140/packages/current/
install.ps1', 'install.ps1'); .\install.ps1 -v
```

Microsoft Windows Server 2008r2 only:

```
[Net.ServicePointManager]::ServerCertificateValidationCallback
= {$true}; $webClient = New-Object System.Net.WebClient;
```

```
$webClient.DownloadFile('https://<MASTER_HOST>:8140/packages/current/
install.ps1', 'install.ps1'); .\install.ps1 -v
```

After running the installer, you see the following output, which indicates the agent was successfully installed.

```
Notice: /Service[puppet]/ensure: ensure changed 'stopped' to 'running'
service { 'puppet':
  ensure => 'running',
  enable => 'true',
}
```

Install Windows agents using a manually transferred certificate

If you need to perform a secure installation on Windows nodes, you can manually transfer the master CA certificate to target nodes, and run a specialized installation script against that cert.

1. Transfer the installation script and the CA certificate from your master to the node you're installing.

File	Location on master	Location on target node
Installation script (install.ps1)	/opt/puppetlabs/server/data/packages/public/	Any accessible local directory.
CA certificate (ca.pem)	/etc/puppetlabs/puppet/ssl/certs/	C:\ProgramData\PuppetLabs\puppet\etc\ssl\certs\

2. Run the installation script, using the `-UsePuppetCA` flag: `.\install.ps1 -UsePuppetCA`

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.

Tip: To install on nodes that don't have internet access, save the .msi package to the appropriate location for your system:

- 32-bit systems — /opt/puppetlabs/server/data/packages/public/<PE_VERSION>/windows-i386-<AGENT_VERSION>/
- 64-bit systems — /opt/puppetlabs/server/data/packages/public/<PE_VERSION>/windows-x86_64-<AGENT_VERSION>/

Install Windows agents with the installer

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

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

Install Windows agents using msixec from the command line

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

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

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

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

MSI properties

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

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

Property	Definition	Setting in <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_MASTER_SERVER</code>	Hostname where the master can be reached.	<code>server</code>	<code>puppet</code>
<code>PUPPET_CA_SERVER</code>	Hostname where the CA master can be reached, if you're using multiple masters and only one of them is acting as the CA.	<code>ca_server</code>	Value of <code>PUPPET_MASTER_SERVER</code>
<code>PUPPET_AGENT_CERTNAME</code>	<p>Node's certificate name, and the name it uses when requesting catalogs.</p> <p>For best compatibility, limit the value of <code>certname</code> to lowercase letters, numbers, periods, underscores, and dashes.</p>	<code>certname</code>	Value of <code>facter fqdn</code>
<code>PUPPET_AGENT_ENVIRONMENT</code>	<p>Node's environment.</p> <p>Note: If a value for the <code>environment</code> variable already exists in <code>puppet.conf</code>, specifying it during installation does not override that value.</p>	<code>environment</code>	<code>production</code>

Property	Definition	Setting in <code>pe.conf</code>	Default
PUPPET_AGENT_STARTUP_MODE	<p>When 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
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 <code>LocalService</code> 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 <code>Logon as Service</code> to the user, and if the user isn't already a local administrator, the installer adds it to the <code>Administrators</code> group.</p> <p>This property must be combined with <code>PUPPET_AGENT_ACCOUNT_PASSWORD</code> and <code>PUPPET_AGENT_ACCOUNT_DOMAIN</code>.</p>	n/a	LocalSystem
PUPPET_AGENT_ACCOUNT_PASSWORD	<p>Password for the agent's user account.</p>	n/a	No Value

Property	Definition	Setting in <code>pe.conf</code>	Default
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 <code>REINSTALLMODE=amus</code> 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 master at `puppet.acme.com`:

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

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

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

Windows agent installation details

Windows nodes can fetch configurations from a master and apply manifests locally, and respond to orchestration commands.

After installing a Windows node, the **Start Menu** contains a **Puppet** folder with shortcuts for running the agent manually, running `Factor`, and opening a command prompt for use with Puppet tools.

Note: You must run Puppet with elevated privileges. Select **Run as administrator** when opening the command prompt.

The agent runs as a Windows service. By default, the agent fetches and applies configurations every 30 minutes. The agent service can be started and stopped independently using either the service control manager UI or the command line `sc.exe` utility.

Puppet is automatically added to the machine's `PATH` environment variable, so you can open any command line and run `puppet`, `factor` and the other batch files that are in the `bin` directory of the Puppet installation. Items necessary for the Puppet environment are also added to the shell, but only for the duration of execution of each of the particular commands.

The installer includes Ruby, gems, and `Factor`. If you have existing copies of these applications, such as Ruby, they aren't affected by the re-distributed version included with Puppet.

Program directory

Unless overridden during installation, PE and its dependencies are installed in `Program Files` at `\Puppet Labs\Puppet`.

You can locate the `Program Files` directory using the `PROGRAMFILES` variable or the `PROGRAMFILES(X86)` variable.

The program directory contains these subdirectories.

Subdirectory	Contents
bin	scripts for running Puppet and Facter
facter	Facter source
hiera	Hiera source
misc	resources
puppet	Puppet source
service	code to run the agent as a service
sys	Ruby and other tools

Data directory

PE stores settings, manifests, and generated data — such as logs and catalogs — in the data directory. The data directory contains two subdirectories for the various components:

- `etc` (the `$confdir`): Contains configuration files, manifests, certificates, and other important files.
- `var` (the `$vardir`): Contains generated data and logs.

When you run Puppet with elevated privileges as intended, the data directory is located in the `COMMON_APPDATA.aspx` folder. This folder is typically located at `C:\ProgramData\PuppetLabs\`. Because the common app data directory is a system folder, it is hidden by default.

If you run Puppet without elevated privileges, it uses a `.puppet` directory in the current user's home folder as its data directory, which can result in unexpected settings.

Install macOS agents

You can install macOS agents with PE package management, from Finder, or from the command line.

Before you begin

[Download](#) the appropriate agent tarball.

To install macOS agents with PE package management, follow the steps to [Install *nix agents with PE package management](#) on page 151.

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.

Install macOS agents from Finder

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

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

You must include the master hostname and the agent certname.

Install macOS agents from the command line

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

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

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

3. Change to the directory indicated as the mount point in the previous step, for example: `cd /Volumes/puppet-agent-VERSION`

4. Install the agent package: `sudo installer -pkg puppet-agent-installer.pkg -target /`
5. Verify the installation: `/opt/puppetlabs/bin/puppet --version`
6. Configure the agent to connect to the master: `/opt/puppetlabs/bin/puppet config set server <MASTER_HOSTNAME>`
7. Configure the agent certname: `/opt/puppetlabs/bin/puppet config set certname <AGENT_CERTNAME>`

macOS agent installation details

macOS agents include core Puppet functionality, plus platform-specific capabilities like package installation, service management with LaunchD, facts inventory with the System Profiler, and directory services integration.

Install non-root agents

Running agents without root privileges can assist teams using PE to work autonomously.

For example, your infrastructure's platform might be maintained by one team with root privileges while your infrastructure's applications are managed by a separate team (or teams) with diminished privileges. If the application team wants to be able to manage its part of the infrastructure independently, they can run Puppet without root privileges.

PE is installed with root privileges, so you need a root user to install and configure non-root access to a master. The root user who performs this installation can then set up non-root users on the master and any nodes running an agent.

Non-root users can perform a reduced set of management tasks, including configuring settings, configuring Facter external facts, running `puppet agent --test`, and running Puppet with non-privileged cron jobs or a similar scheduling service. Non-root users can also classify nodes by writing or editing manifests in the directories where they have write privileges.

Install non-root *nix agents

Note: Unless specified otherwise, perform these steps as a root user.

1. Install the agent on each node that you want to operate as a non-root user.
2. Log in to the agent node and add the non-root user:

```
puppet resource user <UNIQUE NON-ROOT USERNAME> ensure=present
managehome=true
```

Note: Each non-root user must have a unique name.

3. Set the non-root user password.

For example, on most *nix systems: `passwd <USERNAME>`

4. Stop the puppet service:

```
puppet resource service puppet ensure=stopped enable=false
```

By default, the puppet service runs automatically as a root user, so it must be disabled.

5. Disable the Puppet Execution Protocol agent.

- a) In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Agent** group.
- b) On the **Configuration** tab, select the **puppet_enterprise::profile::agent** class, specify parameters, click **Add parameter**, and then commit changes.

Parameter	Value
pxp_enabled	false

6. Change to the non-root user and generate a certificate signing request:

```
puppet agent -t --certname "<UNIQUE NON-ROOT USERNAME.HOSTNAME>" --server
"<MASTER HOSTNAME>"
```

Tip: If you wish to use `su - <NON-ROOT USERNAME>` to switch between accounts, make sure to use the `-` (`-l` in some unix variants) argument so that full login privileges are correctly granted. Otherwise you might see permission denied errors when trying to apply a catalog.

7. On the master or in the console, approve the certificate signing request.
8. On the agent node as the non-root user, set the node's certname and the master hostname, and then run Puppet.

```
puppet config set certname <UNIQUE NON-ROOT USERNAME.HOSTNAME> --section
agent
```

```
puppet config set server <PUPPET MASTER HOSTNAME> --section agent
```

```
puppet agent -t
```

The configuration specified in the catalog is applied to the node.

Tip: If you see `Facter` facts being created in the non-root user's home directory, you have successfully created a functional non-root agent.

To confirm that the non-root agent is correctly configured, verify that:

- The agent can request certificates and apply the catalog from the master when a non-root user runs Puppet:
`puppet agent -t`
- The agent service is not running: `service puppet status`
- Non-root users can collect existing facts by running `facter` on the agent, and they can define new, external facts.

Install non-root Windows agents

Note: Unless specified otherwise, perform these steps as a root user.

1. Install the agent on each node that you want to operate as a non-root user.
2. Log in to the agent node and add the non-root user:

```
puppet resource user <UNIQUE NON-ADMIN USERNAME> ensure=present
managehome=true password="puppet" groups="Users"
```

Note: Each non-root user must have a unique name.

3. Stop the puppet service:

```
puppet resource service puppet ensure=stopped enable=false
```

By default, the puppet service runs automatically as a root user, so it must be disabled.

4. Change to the non-root user and generate a certificate signing request:

```
puppet agent -t --certname "<UNIQUE NON-ADMIN USERNAME>" --server "<MASTER
HOSTNAME>"
```

Important: This Puppet run submits a cert request to the master and creates a `/ .puppet` directory structure in the non-root user's home directory. If this directory is not created automatically, you must manually create it before continuing.

5. As the non-root user, create a configuration file at `%USERPROFILE%/.puppet/puppet.conf` to specify the agent certname and the hostname of the master:

```
[main]
certname = <UNIQUE NON-ADMIN USERNAME.hostname>
server = <MASTER HOSTNAME>
```

6. As the non-root user, submit a cert request: `puppet agent -t`.
7. On the master or in the console, approve the certificate signing request.

Important: It's possible to sign the root user certificate in order to allow that user to also manage the node. However, this introduces the possibility of unwanted behavior and security issues. For example, if your site.pp has no default node configuration, running the agent as non-admin could lead to unwanted node definitions getting generated using alt hostnames, a potential security issue. If you deploy this scenario, ensure the root and non-root users never try to manage the same resources, have clear-cut node definitions, ensure that classes scope correctly, and so forth.

8. On the agent node as the non-root user, run Puppet: `puppet agent -t`.

The configuration specified in the catalog is applied to the node.

Non-root user functionality

Non-root users can use a subset of functionality. Any operation that requires root privileges, such as installing system packages, can't be managed by a non-root agent.

*nix non-root functionality

On *nix systems, as non-root agent you can enforce these resource types:

- 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
- ssh_key
- ssh_authorized_key
- service
- augeas

Note: When running a cron job as non-root user, using the `-u` flag to set a user with root privileges causes the job to fail, resulting in this error message:

```
Notice: /Stage[main]/Main/Node[nonrootuser]/Cron[illegal_action]/ensure:
created must be privileged to use -u
```

You can also inspect these resource types (use `puppet resource <resource type>`):

- host
- mount
- package

Windows non-root functionality

On Windows systems as non-admin user, you can enforce these types :

- exec
- file

Note: A non-root agent on Windows is extremely limited as compared to non-root *nix. While you can use the above resources, you are limited on usage based on what the agent user has access to do (which isn't much). For instance, you can't create a file\directory in C:\Windows unless your user has permission to do so.

You can also inspect these resource types (use `puppet resource <resource type>`):

- host
- package
- user
- group
- service

Managing certificate signing requests

When you install a Puppet agent on a node, the agent automatically submits a certificate signing request (CSR) to the master. You must accept this request to bring the node under PE management can be added your deployment. This allows Puppet to run on the node and enforce your configuration, which in turn adds node information to PuppetDB and makes the node available throughout the console.

You can approve certificate requests from the PE console or the command line. If DNS altnames are set up for agent nodes, you must approve the CSRs on use the command line interface .

Note: Specific user permissions are required to manage certificate requests:

- To accept or reject CSRs in the console or on the command line, you need the permission **Certificate requests: Accept and reject**.
- To manage certificate requests in the console, you also need the permission **Console: View**.

Related information

[Installing agents](#) on page 147

You can install Puppet Enterprise agents on *nix, Windows, and macOS.

Managing certificate signing requests in the console

Open certificate signing requests appear in the console on the **Unsigned certs** page. Accept or reject submitted requests individually or in a batch.

- To manage requests individually, click **Accept** or **Reject**.
- To manage the entire list of requests, click **Accept All** or **Reject All**. Nodes are processed in batches. If you close the browser window or navigate to another website while processing is in progress, only the current batch is processed.

The node appears in the PE console after the next Puppet run. To make a node available immediately after you approve the request, run Puppet on demand.

Related information

[Running Puppet on demand](#) on page 434

The orchestrator gives you the ability to set up jobs in the console or on the command line to trigger on-demand Puppet runs.

Managing certificate signing requests on the command line

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

To view pending node requests on the command line:

```
$ sudo puppetserver ca list
```

To sign a pending request:

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

To sign pending requests for nodes with DNS altnames:

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

Configuring agents

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

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

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

Installing compilers

As your Puppet Enterprise infrastructure scales up to 4,000 nodes and beyond, add load-balanced compilers to your monolithic installation to increase the number of agents you can manage.

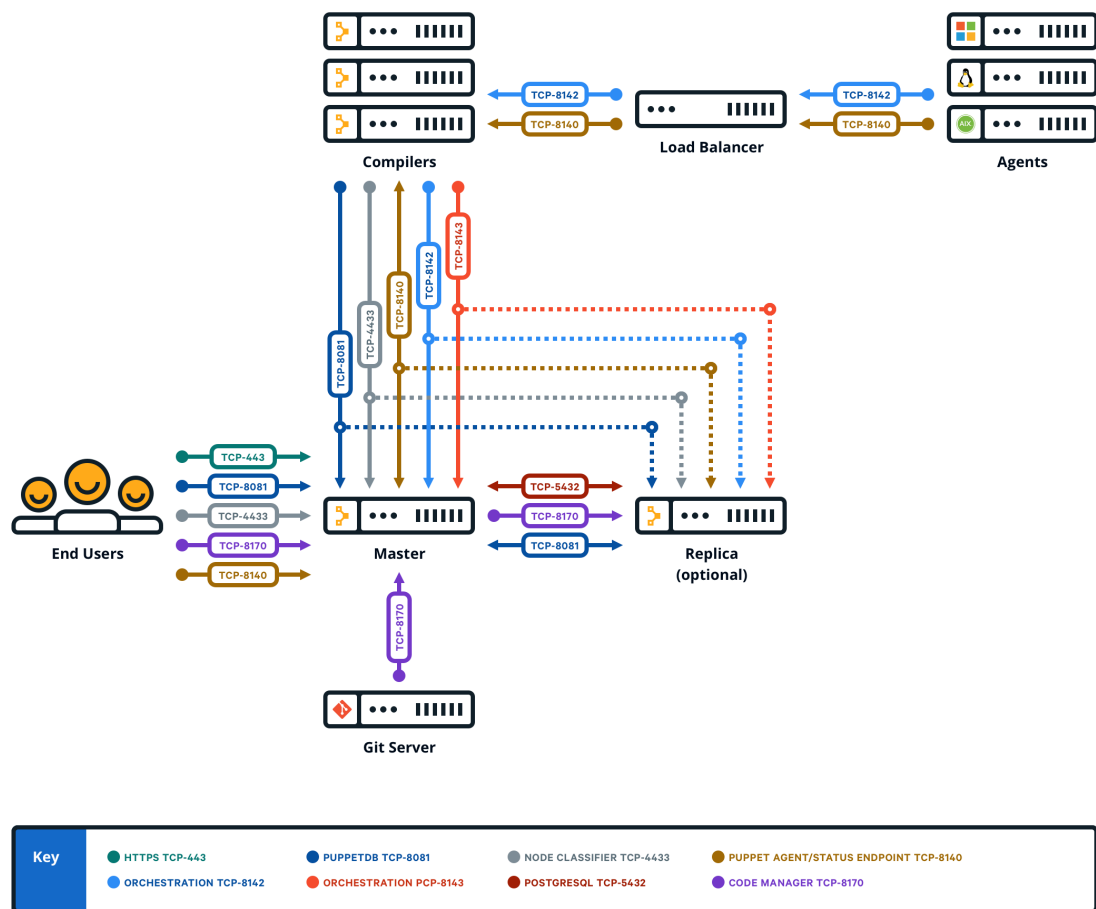
Each compiler increases capacity by 1,500 to 3,000 nodes, until you exhaust the capacity of PuppetDB or the console.

How compilers work

A single master can process requests and compile code for up to 4,000 nodes. When you exceed this scale, expand your infrastructure by adding compilers to share the workload and compile catalogs faster.

Important: Compilers must run the same OS major version, platform, and architecture as the master.

Compilers act as PCP brokers, conveying messages between the orchestrator and Puppet Execution Protocol (PXP) agents. PXP agents connect to PCP brokers running on compilers over port 8142. Status checks on compilers must be sent to port 8140, using `https://<hostname>:8140/status/v1/simple`.



Components and services running on compilers

All compilers contain a Puppet Server and a file sync client.

When triggered by a web endpoint, file sync takes changes from the working directory on the master and deploys the code to a live code directory. File sync then deploys that code to all your compilers, ensuring that all masters in a multi-master configuration remain in sync. By default, compilers check for code updates every five seconds.

The certificate authority (CA) service is disabled on compilers. A proxy service running on the compiler Puppet Server directs CA requests to the master, which hosts the CA in default installations.

Compilers also have:

- The repository for agent installation, `pe_repo`
- The controller profile used with PE client tools
- Puppet Communications Protocol (PCP) brokers to enable orchestrator scale

Logs for compilers are located at `/var/log/puppetlabs/puppetserver/`.

Logs for PCP brokers on compilers are located at `/var/log/puppetlabs/puppetserver/pcp-broker.log`.

Using load balancers with compilers

When using more than one compiler, a load balancer can help distribute the load between the compilers and provide a level of redundancy.

Specifics on how to configure a load balancer infrastructure falls outside the scope of this document, but examples of how to leverage haproxy for this purpose can be found in the HAproxy module documentation.

Load balancing

PCP brokers run on compilers and connect to PXP agents over port 8142. PCP brokers are built on websockets and require many persistent connections. If you're not using HTTP health checks, we recommend using a round robin or random load balancing algorithm for PXP agent connections to PCP brokers, because PCP brokers don't operate independent of the orchestrator and isolate themselves if they become disconnected. You can check connections with the `/status/v1/simple` endpoint for an error state.

You must also configure your load balancer to avoid closing long-lived connections that have little traffic. In the HAproxy module, you can set the `timeout tunnel` to 15m because PCP brokers disconnect inactive connections after 15 minutes.

Using health checks

The Puppet REST API exposes a status endpoint that can be leveraged from a load balancer health check to ensure that unhealthy hosts do not receive agent requests from the load balancer.

The master service responds to unauthenticated HTTP GET requests issued to `https://<hostname>:8140/status/v1/simple`. The API responds with an HTTP 200 status code if the service is healthy.

If your load balancer doesn't support HTTP health checks, a simpler alternative is to check that the host is listening for TCP connections on port 8140. This ensures that requests aren't forwarded to an unreachable instance of the master, but it does not guarantee that a host is pulled out of rotation if it's deemed unhealthy, or if the service listening on port 8140 is not a service related to Puppet.

Optimizing workload distribution

Due to the diverse nature of the network communications between the agent and the master, we recommend that you implement a load balancing algorithm that distributes traffic between compilers based on the number of open connections. Load balancers often refer to this strategy as "balancing by least connections."

Related information

[Firewall configuration for large installations with compilers](#) on page 126

These are the port requirements for large installations with compilers.

[GET /status/v1/simple](#) on page 306

The `/status/v1/simple` returns a status that reflects all services the status service knows about.

Install compilers

To install a compiler, you first install an agent and then classify that agent as a compiler.

1. SSH into the node that you want to make a compiler and install the agent:

```
curl -k https://<MASTER_HOSTNAME>:8140/packages/current/install.bash |
  sudo bash -s main:dns_alt_names=<COMMA-SEPARATED LIST OF ALT NAMES FOR
  THE COMPILER>
```

Note: Set the `dns_alt_names` value to a comma-separated list of any alternative names that agents use to connect to compilers. The installation uses puppet by default. If your `puppet.conf` file includes a `dns_alt_names` entry, you must include the `dns_alt_names` parameter.

2. From the master, sign the compiler's certificate:

```
puppetserver ca sign --certname <COMPILER_HOSTNAME>
```

3. Pin the compiler node to the **PE Master** node group.
 - a) In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Master** group.
 - b) Enter the **Certname** for the compiler, click **Pin node**, and then commit changes.
4. From the compiler, run Puppet: `puppet agent -t`
5. From the master, run Puppet: `puppet agent -t`

After installing compilers, you must configure them to appropriately route communication between your master and agent nodes.

Configure compilers

Compilers must be configured to appropriately route communication between your master and agent nodes.

Before you begin

- Install compilers and load balancers.
 - If you need DNS altnames for your load balancers, add them to the master.
 - Ensure port 8143 is open on the master or on any workstations used to run orchestrator jobs.
1. Configure `pe_repo::compile_master_pool_address` to send agent install requests to the load balancer.

Important: If you have load balancers in multiple data centers, you must configure `compile_master_pool_address` using Hiera, instead of using configuration data in the console, as described in this step. Using either of these methods updates the agent install script URL displayed in the console.

Note: If you are using a single compile master, configure `compile_master_pool_address` with the compile master's fully qualified domain name (FQDN).

- a) In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Master** group.
- b) Select the **Configuration** tab, and in the Data section, in the **pe_repo** class, specify parameters:

Parameter	Value
<code>compile_master_pool_address</code>	Load balancer hostname.

- c) Click **Add data** and commit changes.
2. Run Puppet on the compiler, and then on the master.
3. Configure infrastructure agents to connect orchestration agents to the master.
 - a) In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Infrastructure Agent** group.
 - b) If you manage your load balancers with agents, on the **Rules** tab, pin load balancers to the group.
 - c) On the **Configuration** tab, find the **puppet_enterprise::profile::agent** class and specify parameters:

Parameter	Value
<code>pcp_broker_list</code>	JSON list including the hostname for your master. If you have an HA replica, include it after the master. Hostnames must include port 8142, for example <code>["MASTER.EXAMPLE.COM:8142"]</code> .

Parameter	Value
master_uris	Provide the host name for your master, for example, ["https://MASTER.EXAMPLE.COM"]. Uris must begin with https://. This setting assumes port 8140 unless you specify otherwise with host:port.

- d) Remove any values set for **pcp_broker_ws_uris**.
- e) Commit changes.
- f) Run Puppet on all agents classified into the **PE Infrastructure Agent** group.

This Puppet run doesn't change PXP agent configuration. If you have high availability configured and haven't already pinned your load balancer to the **PE Infrastructure Agent** group, the Puppet run configures your load balancer to compile catalogs on the master.

4. Configure agents to connect orchestration agents to the load balancer.

- a) In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Agent** group.
- b) On the **Configuration** tab, find the **puppet_enterprise::profile::agent** class and specify parameters:

Parameter	Value
pcp_broker_list	JSON list including the hostname for your load balancers. Hostnames must include port 8142, for example ["LOADBALANCER1.EXAMPLE.COM:8142" , "LOADBALANCER2.EXAMPLE.COM:8142"].
master_uris	Provide a list of load balancer host names, for example, ["https://LOADBALANCER1.EXAMPLE.COM" , "https://LOADBALANCER2.EXAMPLE.COM"]. Uris must begin with https://. This setting assumes port 8140 unless you specify otherwise with host:port.

- c) Remove any values set for **pcp_broker_ws_uris**.
- d) Commit changes.
- e) Run Puppet on the master, then run Puppet on all agents, or install new agents.

This Puppet run configures PXP agents to connect to the load balancer.

Related information

[Firewall configuration for large installations with compilers](#) on page 126

These are the port requirements for large installations with compilers.

Installing PE client tools

PE client tools are a set of command line tools that let you access Puppet Enterprise services from a workstation that might or might not be managed by Puppet.

The `pe-client-tools` package is included in the PE installation tarball. When you install, the client tools are automatically installed on the same node as the master.

Client tools versions align with PE versions. For example, if you're running PE 2017.3, use the 2017.3 client tools. In some cases, we might issue patch releases ("x.y.z") for PE or the client tools. You don't need to match patch numbers between PE and the client tools. Only the "x.y" numbers need to match.

Important: When you upgrade PE to a new "x.y" version, install the appropriate "x.y" version of PE client tools.

The package includes client tools for these services:

- **Orchestrator** — Allow you to control the rollout of changes in your infrastructure, and provides the interface to the orchestration service. Tools include `puppet-job` and `puppet-app`.
- **Puppet access** — Authenticates you to the PE RBAC token-based authentication service so that you can use other capabilities and APIs.
- **Code Manager** — Provides the interface for the Code Manager and file sync services. Tools include `puppet-code`.
- **PuppetDB CLI** — Enables certain operations with PuppetDB, such as building queries and handling exports.

Because you can safely run these tools remotely, you no longer need to SSH into the master to execute commands. Your permissions to see information and to take action are controlled by PE role-based access control. Your activity is logged under your username rather than under `root` or the `pe-puppet` user.

Related information

[Orchestrator configuration files](#) on page 426

The configuration file for the orchestrator allows you to run commands from the CLI without having to pass additional flags. Whether you are running the orchestrator from the Puppet master or from a separate work station, there are two types of configuration files: a global configuration file and a user-specified configuration file.

[Configuring puppet-access](#) on page 244

The `puppet-access` command allows users to generate and manage authentication tokens from the command line of any workstation (Puppet-managed or not), without the need to SSH into the Puppet master.

[Installing and configuring puppet-code](#) on page 572

PE automatically installs and configures the `puppet-code` command on your masters as part of the included PE client tools package. You can also set up `puppet-code` on an agent node or on a workstation, customize configuration for different users, or change the configuration settings.

Supported PE client tools operating systems

The PE client tools package can be installed on these platforms.

Operating system	Versions	Arch
Red Hat Enterprise Linux	6, 7	x86_64
CentOS	6, 7	x86_64
Oracle Linux	6, 7	x86_64
Scientific Linux	6, 7	x86_64
SUSE Linux Enterprise Server	11, 12	x86_64
Ubuntu	16.04	amd64
Windows (Consumer OS)	7, 8.1, 10	x86, x64
Windows Server (Server OS)	2008	x86, x64
	2008r, 22012, 2012r2, 2012r2 core	x64
	2016	
macOS	10.10, 10.11, 10.12, 10.13	

Install PE client tools on a managed workstation

To use the client tools on a system other than the master, where they're installed by default, you can install the tools on a *controller node*.

Before you begin

Controller nodes must be running the same OS as your master and must have an agent installed.

1. In the console, create a controller classification group, for example `PE Controller`, and ensure that its **Parent name** is set to **All Nodes**.
2. Select the controller group and add the `puppet_enterprise::profile::controller` class.
3. Pin the node that you want to be a controller to the controller group.
 - a) In the controller group, on the **Rules** tab, in the **Certname** field, enter the certname of the node.
 - b) Click **Pin node** and commit changes.
4. Run Puppet on the controller machine.

Related information

[Create classification node groups](#) on page 323

Create classification node groups to assign classification data to nodes.

Install PE client tools on an unmanaged workstation

You can install the `pe-client-tools` package on any workstation running a supported OS. The workstation OS does not need to match the master OS.

Before you begin

Review prerequisites for timekeeping, name resolution, and firewall configuration, and ensure that these ports are available on the workstation.

- **8143** — The orchestrator client uses this port to communicate with orchestration services running on the master.
- **4433** — The Puppet access client uses this port to communicate with the RBAC service running on the master.
- **8170** — If you use the Code Manager service, it requires this port.

Install PE client tools on an unmanaged Linux workstation

1. On the workstation, create the directory `/etc/puppetlabs/puppet/ssl/certs`.
2. On the master, navigate to `/etc/puppetlabs/puppet/ssl/certs/` and copy `ca.pem` to the directory you created on the workstation.
3. On the workstation, make sure file permissions are correct: `chmod 444 /etc/puppetlabs/puppet/ssl/certs/ca.pem`
4. Verify that the checksum of `ca.pem` on the workstation matches the checksum of the same file on the master.
5. Download the [pe-client-tools package](#) for the platform appropriate to your workstation.
6. Unpack the tarball and navigate to the `packages/<PLATFORM>` directory.
7. Use your workstation's package management tools to install the `pe-client-tools`.
For example, on RHEL platforms: `rpm -Uvh pe-client-tools-<VERSION-and-PLATFORM>.rpm`

Install PE client tools on an unmanaged Windows workstation

You can install the client tools on a Windows workstation using the setup wizard or the command line.

To start using the client tools on your Windows workstation, open the **PE ClientTools Console** from the **Start** menu.

1. On the workstation, create the directory `C:\ProgramData\PuppetLabs\puppet\etc\ssl\certs`.
For example: `mkdir C:\ProgramData\PuppetLabs\puppet\etc\ssl\certs`
2. On the master, navigate to `/etc/puppetlabs/puppet/ssl/certs/` and copy `ca.pem` to the directory you created on the workstation.
3. On the workstation, make sure the file permissions are set to read-only for `C:\ProgramData\PuppetLabs\puppet\etc\ssl\certs\ca.pem`.
4. Verify that the checksum of `ca.pem` on the workstation matches the checksum of the same file on the master.

5. Install the client tools using guided setup or the command line.

- Guided setup
 - a. Download the Windows [pe-client-tools-package](#).
 - b. Double-click the `pe-client-tools.msi` file.
 - c. Follow prompts to accept the license agreement and select the installation location.
 - d. Click **Install**.
- Command line
 - a. Download the Windows [pe-client-tools-package](#).
 - b. From the command line, run the installer:

```
msiexec /i <PATH TO PE-CLIENT-TOOLS.MSI> TARGETDIR="<INSTALLATION
  DIRECTORY>"
```

TARGETDIR is optional.

Install PE client tools on an unmanaged macOS workstation

You can install the client tools on a macOS workstation using Finder or the command line.

1. On the workstation, create the directory `/etc/puppetlabs/puppet/ssl/certs`.
2. On the master, navigate to `/etc/puppetlabs/puppet/ssl/certs/` and copy `ca.pem` to the directory you created on the workstation.
3. On the workstation, make sure file permissions are correct: `chmod 444 /etc/puppetlabs/puppet/ssl/certs/ca.pem`
4. Verify that the checksum of `ca.pem` on the workstation matches the checksum of the same file on the master.
5. Install the client tools using Finder or the command line.

- Finder
 - a. Download the macOS [pe-client-tools-package](#).
 - b. Open the `pe-client-tools.dmg` and click the installer `.pkg`.
 - c. Follow the prompts to install the client tools.
- Command line

- a. Download the macOS [pe-client-tools-package](#).
- b. 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.

- c. Run `cd /Volumes/pe-client-tools-VERSION`.
- d. Run `sudo installer -pkg pe-client-tools-<VERSION>-installer.pkg -target /`.
- e. Run `cd ~` and then run `sudo umount /Volumes/pe-client-tools-VERSION`.

Configuring and using PE client tools

Use configuration files to customize how client tools communicate with the master.

For each client tool, you can create config files for individual machines (global) or for individual users. Configuration files are structured as JSON.

Save configuration files to these locations:

- Global
 - *nix — `/etc/puppetlabs/client-tools/`
 - Windows — `%ProgramData%\puppetlabs\client-tools`

- User
 - *nix — `~/.puppetlabs/client-tools/`
 - Windows — `%USERPROFILE%\puppetlabs\client-tools`

On managed client nodes where the operating system and architecture match the master, you can have PE manage Puppet code and orchestrator global configuration files using the `puppet_enterprise::profile::controller` class.

For example configuration files and details about using the various client tools, see the documentation for each service.

Client tool	Documentation
Orchestrator	<ul style="list-style-type: none"> • Running jobs with Puppet orchestrator on page 419 • Running Puppet on demand from the CLI on page 439 • Running tasks from the command line on page 458 • Review jobs from the command line on page 508
Puppet access	<ul style="list-style-type: none"> • Token-based authentication on page 242
Puppet code	<ul style="list-style-type: none"> • Triggering Code Manager on the command line on page 572
PuppetDB	<ul style="list-style-type: none"> • PuppetDB CLI

Related information

[Orchestrator configuration files](#) on page 426

The configuration file for the orchestrator allows you to run commands from the CLI without having to pass additional flags. Whether you are running the orchestrator from the Puppet master or from a separate work station, there are two types of configuration files: a global configuration file and a user-specified configuration file.

[Configuring puppet-access](#) on page 244

The `puppet-access` command allows users to generate and manage authentication tokens from the command line of any workstation (Puppet-managed or not), without the need to SSH into the Puppet master.

[Installing and configuring puppet-code](#) on page 572

PE automatically installs and configures the `puppet-code` command on your masters as part of the included PE client tools package. You can also set up `puppet-code` on an agent node or on a workstation, customize configuration for different users, or change the configuration settings.

Installing external PostgreSQL

By default, Puppet Enterprise includes its own database backend, PE-PostgreSQL, which is installed alongside PuppetDB. If the load on your PuppetDB node is larger than it can effectively scale to (greater than 20,000 nodes), you can install a standalone instance of PE-PostgreSQL.

In certain limited circumstances, you might choose to configure a PostgreSQL instance that's not managed by PE. Using unmanaged PostgreSQL increases complexity for maintenance and upgrades, so we recommend this configuration only for customers who can't use PE-PostgreSQL.

Install standalone PE-PostgreSQL

If the load on your PuppetDB node is larger than it can effectively scale to (greater than 20,000 nodes), you can install a standalone instance of PE-PostgreSQL.

Before you begin

You must have root access to the node on which you plan to install PE-PostgreSQL, as well as the ability to SSH and copy files to the node.

1. Prepare your `pe.conf` file by specifying parameters required for PostgreSQL.

```
"puppet_enterprise::puppet_master_host": "<MASTER_HOSTNAME>"
"puppet_enterprise::database_host": "<PE-POSTGRES SQL NODE HOSTNAME>"
```

2. Follow the instructions to [Install using text install](#) on page 136, running the installer with the `-c` flag.

The installer hangs halfway through, because it can't contact the database. Leave the process running and proceed to the next step.

3. Copy the `pe.conf` file you created to the PE-PostgreSQL node and SSH into that node.
4. Run the installer with the `-c` flag, using the same `pe.conf` file.
When the installation process finishes on the PE-PostgreSQL node, the installer automatically resumes installation on the master.

Tip: If installation fails to resume on the master automatically, connect with SSH to the master and run `puppet infrastructure configure`.

5. SSH into the master and run Puppet: `puppet agent -t`

The master is configured to use the standalone PE-PostgreSQL installation on the PE-PostgreSQL node.

Install unmanaged PostgreSQL

If you use Amazon RDS, or if your business requirements dictate that databases must be managed outside of Puppet, you can configure a PostgreSQL database that's not managed by PE.

Before you begin

You must have:

- PostgreSQL 9.6 or later.
- The complete certificate authority certificate chain for the external party CA, in PEM format.
- The DNS-addressable name, username, and password for the external PostgreSQL database.

Important: Using unmanaged PostgreSQL increases complexity for maintenance and upgrades, so we recommend it only for customers who can't use PE-PostgreSQL.

Create the unmanaged PostgreSQL instance

Create the unmanaged PostgreSQL instance, and, if you haven't already, retrieve the necessary certificate chain and credentials from your database administrator.

For example, for RDS, the root certificate is available [here](#).

Create PE databases on the unmanaged PostgreSQL instance

1. Log in to the unmanaged PostgreSQL instance with the client of your choice.
2. Create databases for the orchestrator, RBAC, activity service, and the node classifier.

```
CREATE USER "pe-inventory" PASSWORD '<PASSWORD>';
GRANT "pe-inventory" TO <ADMIN USER>;
CREATE DATABASE "pe-inventory" OWNER "pe-inventory"
```

```

ENCODING 'utf8' LC_CTYPE 'en_US.utf8' LC_COLLATE 'en_US.utf8' template
template0;

CREATE USER "pe-puppetdb" PASSWORD '<PASSWORD>';
GRANT "pe-puppetdb" TO <ADMIN USER>;
CREATE DATABASE "pe-puppetdb" OWNER "pe-puppetdb"
ENCODING 'utf8' LC_CTYPE 'en_US.utf8' LC_COLLATE 'en_US.utf8' template
template0;

CREATE USER "pe-orchestrator" PASSWORD '<PASSWORD>';
GRANT "pe-orchestrator" TO <ADMIN USER>;
CREATE DATABASE "pe-orchestrator" OWNER "pe-orchestrator"
ENCODING 'utf8' LC_CTYPE 'en_US.utf8' LC_COLLATE 'en_US.utf8' template
template0;

CREATE USER "pe-activity" PASSWORD '<PASSWORD>';
GRANT "pe-activity" TO <ADMIN USER>;
CREATE DATABASE "pe-activity" OWNER "pe-activity"
ENCODING 'utf8' LC_CTYPE 'en_US.utf8' LC_COLLATE 'en_US.utf8' template
template0;

CREATE USER "pe-classifier" PASSWORD '<PASSWORD>';
GRANT "pe-classifier" TO <ADMIN USER>;
CREATE DATABASE "pe-classifier" OWNER "pe-classifier"
ENCODING 'utf8' LC_CTYPE 'en_US.utf8' LC_COLLATE 'en_US.utf8' template
template0;

CREATE USER "pe-rbac" PASSWORD '<PASSWORD>';
GRANT "pe-rbac" TO <ADMIN USER>;
CREATE DATABASE "pe-rbac" OWNER "pe-rbac"
ENCODING 'utf8' LC_CTYPE 'en_US.utf8' LC_COLLATE 'en_US.utf8' template
template0;

\c "pe-rbac"
CREATE EXTENSION citext;
CREATE EXTENSION pg_trgm;
CREATE EXTENSION plpgsql;
CREATE EXTENSION pgcrypto;

\c "pe-orchestrator"
CREATE EXTENSION citext;
CREATE EXTENSION pg_trgm;
CREATE EXTENSION plpgsql;

\c "pe-puppetdb"
CREATE EXTENSION citext;
CREATE EXTENSION pg_trgm;
CREATE EXTENSION plpgsql;
CREATE EXTENSION pgcrypto;

\c "pe-classifier"
CREATE EXTENSION citext;
CREATE EXTENSION pg_trgm;
CREATE EXTENSION plpgsql;

\c "pe-activity"
CREATE EXTENSION citext;
CREATE EXTENSION pg_trgm;
CREATE EXTENSION plpgsql;

\c "pe-inventory"
CREATE EXTENSION citext;
CREATE EXTENSION pg_trgm;
CREATE EXTENSION plpgsql;

```

```
CREATE EXTENSION pgcrypto;
```

Next, install PE using [text mode](#), specifying [external PostgreSQL parameters](#) in `pe.conf`.

Establish SSL between PE and the unmanaged PostgreSQL instance

Before you begin

Install PE using [text mode](#), specifying [external PostgreSQL parameters](#) in `pe.conf`.

1. Log in to the master and stop the agent service:

```
/opt/puppetlabs/puppet/bin/puppet resource service puppet ensure=stopped
```

2. On the master, create a location to store the CA cert from the external PostgreSQL instance, for example `/etc/puppetlabs/puppet/ssl/`.
3. Transfer the CA cert from the unmanaged PostgreSQL instance to the directories that you created.
4. Specify ownership and permissions for the certificate and directories.

```
chown -R pe-puppet:pe-puppet <PATH TO DIRECTORY>
chmod 664 <PATH TO DIRECTORY>/<CERT NAME>
```

5. Modify your `pe.conf` file to specify database properties.

```
"puppet_enterprise::profile::console::database_properties": "?
ssl=true&sslfactory=org.postgresql.ssl.jdbc4.LibPQFactory&sslmode=verify-
full&sslrootcert=<PATH TO EXTERNAL POSTGRESQL CA CERT>"
"puppet_enterprise::profile::puppetdb::database_properties": "?
ssl=true&sslfactory=org.postgresql.ssl.jdbc4.LibPQFactory&sslmode=verify-
full&sslrootcert=<PATH TO EXTERNAL POSTGRESQL CA CERT>"
"puppet_enterprise::profile::orchestrator::database_properties": "?
ssl=true&sslfactory=org.postgresql.ssl.jdbc4.LibPQFactory&sslmode=verify-
full&sslrootcert=<PATH TO EXTERNAL POSTGRESQL CA CERT>"
"puppet_enterprise::database_ssl": true
"puppet_enterprise::database_cert_auth": false
```

6. On the master, run Puppet.
7. On the master, start the agent service: `puppet resource service puppet ensure=running`

Installing FIPS-compliant Puppet Enterprise

PE version 2019.2 complies with Federal Information Processing Standard (FIPS) 140-2 standards. Both the express and text install methods can be used to install a FIPS-compliant PE master on FIPS-compliant platforms.

The cryptographic modules included in Puppet Enterprise version 2019.2 are compliant with FIPS 140-2 and fully operable on the FIPS-compliant platforms listed below.

PE component	FIPS-compliant platforms
Master	Red Hat Enterprise Linux (RHEL) 7 in FIPS mode
Agents	Red Hat Enterprise Linux (RHEL) 7 in FIPS mode
	Windows Server 2012 R2 and newer versions in FIPS mode
	Windows 10 in FIPS mode

For general information about FIPS and more on the updates made to PE to ensure FIPS compliance, see [PE and FIPS compliance](#).

Installing a FIPS-compliant PE master

To install a FIPS-compliant PE master, follow the text or express installation instructions to install your PE master on a node with the following characteristics:

- Running Red Hat Enterprise Linux (RHEL) 7 in FIPS mode
- Configured with good sources of entropy (see the Red Hat blog post on [Entropy in RHEL](#) for more information)

Important: The installation process will fail on a node that lacks sufficient available entropy.

Installing FIPS-compliant PE agents

To install FIPS-compliant PE agents, follow the agent installation instructions appropriate to your operating system and infrastructure. Your target agent node's operating system must be one of the following:

- Windows 10 in FIPS mode
- Windows Server 2012 R2 and newer versions in FIPS mode
- Red Hat Enterprise Linux (RHEL) 7 in FIPS mode

If you wish to classify a FIPS-compliant PE agent as a compiler, the agent must be configured with good sources of entropy (see the Red Hat blog post on [Entropy in RHEL](#) for more information).

Limitations and cautions for FIPS-compliant installations

Be aware of the following when installing FIPS-compliant PE.

- Upgrading from non-FIPS-compliant versions of PE to FIPS-compliant PE version 2019.2 is not supported.
- FIPS-compliant PE version 2019.2 includes PostgreSQL version 9.6.
- High availability configurations are not supported for FIPS-compliant PE version 2019.2.
- FIPS-compliant PE does not support Razor, and the `pe-razor-server` package is not included in the installation tarball for FIPS-compliant PE.
- Due to a known issue with the `pe-client-tools` packages, `puppet code` and `puppet db` commands fail with SSL handshake errors when run on FIPS-enabled hardware. To use `puppet db` commands on a FIPS-enabled machine, install the `puppetdb_cli` Ruby gem. To use `puppet code` commands on a FIPS-enabled machine, use the [Code Manager API](#).

Uninstalling

Puppet Enterprise includes a script for uninstalling. You can uninstall component infrastructure nodes or from agent nodes.

Uninstall infrastructure nodes

The `puppet-enterprise-uninstaller` script is installed on the master. In order to uninstall, you must run the uninstaller on each infrastructure node.

By default, the uninstaller removes the software, users, logs, cron jobs, and caches, but it leaves your modules, manifests, certificates, databases, and configuration files in place, as well as the home directories of any users it removes.

1. From the infrastructure node that you want to uninstall, from the command line as root, navigate to the installer directory and run the uninstall command: `$ sudo ./puppet-enterprise-uninstaller`

Note: If you don't have access to the installer directory, you can run `/opt/puppetlabs/bin/puppet-enterprise-uninstaller`.

2. Follow prompts to uninstall.
3. (Optional) If you don't uninstall the master, and you plan to reinstall on an infrastructure node at a later date, remove the agent certificate for that component from the master. On the master: `puppetserver ca clean <PE COMPONENT CERT NAME>`.

Uninstall agents

You can remove the agent from nodes that you no longer want to manage.

Note: Uninstalling the agent doesn't remove the node from your environment. To completely remove all traces of a node, you must also purge the node.

Related information

[Adding and removing agent nodes](#) on page 314

After you install a Puppet agent on a node, accept its certificate signing request and begin managing it with Puppet Enterprise (PE). Or remove nodes that you no longer need.

Uninstall *nix agents

The *nix agent package includes an uninstall script, which you can use when you're ready to retire a node.

1. On the agent node, run the uninstall script: `run /opt/puppetlabs/bin/puppet-enterprise-uninstaller`
2. Follow prompts to uninstall.
3. (Optional) If you plan to reinstall on the node at a later date, remove the agent certificate for the agent from the master: `puppetserver ca clean <AGENT CERT NAME>`

Uninstall Windows agents

To uninstall the agent from a Windows node, use the Windows **Add or Remove Programs** interface, or uninstall from the command line.

Uninstalling the agent removes the Puppet program directory, the agent service, and all related registry keys. The data directory remains intact, including all SSL keys. To completely remove Puppet from the system, manually delete the data directory.

1. Use the Windows **Add or Remove Programs** interface to remove the agent.
Alternatively, you can uninstall from the command line if you have the original .msi file or know the product code of the installed MSI, for example: `msiexec /qn /norestart /x [puppet.msi|<PRODUCT_CODE>]`
2. (Optional) If you plan to reinstall on the node at a later date, remove the agent certificate for the agent from the master: `puppetserver ca clean <AGENT CERT NAME>`

Uninstall macOS agents

Use the command line to remove all aspects of the agent from macOS nodes.

1. On the agent node, run these commands:

```
rm -rf /var/log/puppetlabs
rm -rf /var/run/puppetlabs
pkgutil --forget com.puppetlabs.puppet-agent
launchctl remove puppet
rm -rf /Library/LaunchDaemons/com.puppetlabs.puppet.plist
launchctl remove pxp-agent
rm -rf /Library/LaunchDaemons/com.puppetlabs.pxp-agent.plist
rm -rf /etc/puppetlabs
rm -rf /opt/puppetlabs
```

2. (Optional) If you plan to reinstall on the node at a later date, remove the agent certificate for the agent from the master: `puppetserver ca clean <AGENT CERT NAME>`

Uninstaller options

You can use the following command-line flags to change the uninstaller's behavior.

- `-p` — Purge additional files. With this flag, the uninstaller also removes all configuration files, modules, manifests, certificates, the home directories of any users created by the installer, and the Puppet public GPG key used for package verification.
- `-d` — Also remove any databases created during installation.

- `-h` — Display a help message.
- `-n` — Run in noop mode; show commands that would have been run during uninstallation without actually running them.
- `-y` — Don't ask to confirm uninstallation, assuming an answer of yes.

To remove every trace of PE from a system, run:

```
$ sudo ./puppet-enterprise-uninstaller -d -p
```

Upgrading

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

Upgrading Puppet Enterprise

Upgrade your PE installation as new versions become available.

Upgrade paths

These are the valid upgrade paths for PE.

If you're on version...	Upgrade to...	Notes
2019.2.z	You're up to date!	
2019.1.z	2019.2	
2019.0.z	2019.2	
2018.1.1 or later	2019.2	
2018.1.0	2018.1.z	You must have version 2018.1.1 or later in order to complete prerequisites for upgrade to 2019.2. For details, see Upgrade cautions on page 180.
2017.3.z	2018.1	
2017.2.z	2018.1	
2017.1.z	2018.1	
2016.5.z	2018.1	
2016.4.10 or later	2018.1	
2016.4.9 or earlier	latest 2016.4.z, then 2018.1	To upgrade to 2018.1 from 2015.2.z through 2016.4.9, you must first upgrade to the latest 2016.4.z .
2016.2.z		
2016.1.z		
2015.3.z		
2015.2.z		
3.8.x	latest 2016.4.z, then 2018.1	To upgrade from 3.8.x, you must first migrate to the latest 2016.4.z . This upgrade requires a different process than upgrades from other versions.

Upgrade cautions

These are the major updates to recent PE versions that you should be aware of when upgrading.

Important: Always back up your installation before performing any upgrade.

Orchestrator memory use increase in PE 2019.2

Puppet orchestrator uses more memory in version 2019.2 than in previous versions due to the addition of a Java virtual machine (JVM), which enables new features and functionalities such as plans. If your memory use is near capacity when running PE 2019.1 or older versions, allocate additional memory before upgrading to PE 2019.2.

Additionally, take care when writing plans, as they can require more memory than is allocated to the orchestrator. To work around this issue, rewrite the plan or increase the memory allocated to the orchestrator.

PostgreSQL 11 upgrade in PE 2019.2

PE 2019.2.0 includes an upgrade from `pe-postgresql` version 9.6 to version 11. As with any major version bump of PostgreSQL, the datastore must be migrated to a format compatible with the new version of PostgreSQL. The PE installer performs this migration automatically using the PostgreSQL `pg_upgrade` utility. Because both the 9.6 datastore and the new 11 datastore remain present on disk, the partition used for PostgreSQL must have enough space for the migrated datastore (calculated with a margin as 110% of the current 9.6 datastore). The installer will issue a warning and cancel the upgrade if there is insufficient space.

The datastore migration also increases the amount of time required to complete the upgrade. The time required varies depending on your installation's size and hardware setup, but broadly, expect between two and four minutes of additional time per 10GB of datastore size.

The `pe-modules` `pe_postgresql_info` fact provides information about the size of your PostgreSQL installation as well as the size and number of available bytes for the partition. To review this fact, run `facter -p pe_postgresql_info` on the PE node that runs the `pe-postgresql` service (either the master node or the standalone PostgreSQL node in extra-large installations).

Note: If you use an external PostgreSQL instance not managed by PE, you must separately upgrade the instance to PostgreSQL 11. For more information, see [Upgrading PostgreSQL](#).

TLSv1 and v1.1 disabled in PE 2019.1

As of PE 2019.1, TLSv1 and TLSv1.1 is now disabled by default to comply with security regulations. You must [enable TLSv1](#) to upgrade agents on these platforms:

- AIX
- CentOS 5
- RHEL 5
- SLES 11
- Solaris 10
- Windows Server 2008r2

Migration of PuppetDB `resource_events` table in PE 2019.1

Upgrading from PE versions prior to 2019.1 requires PuppetDB to perform a migration of the `resource_events` table. This migration can take a couple of hours to complete for installations with thousands of agents. The migration also produces a lot of disk writes, which can increase the performance overhead of VM snapshots. If you have installations large enough to be using an external database node, consider a database backup as a rollback strategy instead of a VM snapshot.

Truncating the `resource_events` table to decrease migration time

Truncating the `resource_events` can significantly reduce migration time and lessen your downtime, especially if your table is larger than a few gigabytes (GB). If you truncate the table, the **Events** page in the PE console will be temporarily blank after the upgrade. The data will be repopulated with the restoration of regular Puppet runs.

To determine the size of your `resource_events` table, run the following command on the node where PostgreSQL is running (the master node in a standard or large installation, or the standalone PE-PostgreSQL node on an extra-large installation):

```
su pe-postgres --shell /bin/bash --command "/opt/puppetlabs/
server/bin/psql --dbname pe-puppetdb --command \"SELECT relname,
pg_size_pretty(pg_relation_size(oid)) AS size FROM pg_class WHERE
relname='resource_events';\""
```

Truncate the `resource_events` table

To truncate your `resource_events` table, run the following command on your master:

```
su - pe-postgres --shell /bin/bash --command "/opt/puppetlabs/server/bin/
psql --dbname pe-puppetdb --command 'TRUNCATE resource_events' "
```

Truncate the `resource_events` table on a high availability installation

To truncate your `resource_events` table on a high availability installation:

1. Stop PuppetDB on your master and replica:

```
sudo puppet resource service puppetdb ensure=stopped
```

2. Run the truncation command on the node where PostgreSQL is running (the master or the standalone PE-PostgreSQL node):

```
su - pe-postgres --shell /bin/bash --command "/opt/puppetlabs/server/bin/
psql --dbname pe-puppetdb --command 'TRUNCATE resource_events' "
```

3. Restart PuppetDB on your master and replica:

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

Certificate architecture and handling in PE 2019.0

PE 2019.0 and later, courtesy of Puppet Server, uses an intermediate certificate authority architecture by default. When you upgrade to PE 2019.0 or later, you can optionally [regenerate certificates](#) to adopt the intermediate certificate architecture.

To adopt the new CA architecture, both your master and agents must be upgraded, and you must regenerate certificates. You can use pre-6.x agents with a Puppet 6.x or PE 2019.0 or later master, but this combination doesn't take advantage of the new intermediate certificate authority architecture. If you don't upgrade *all* of your nodes to 6.x, don't regenerate certificates, because pre-6.x agents won't work with the new CA architecture.

MCollective removal in PE 2019.0

If you're upgrading from a 2018.1 installation with MCollective enabled, you must take additional steps to ensure a successful upgrade.

Before upgrade

- **Remove MCollective** from nodes in your infrastructure. If any nodes are configured with MCollective or ActiveMQ profiles when you attempt to upgrade, the installer halts and prompts you to remove the profiles. For example, remove PE MCollective node group and any of the deprecated parameters:
 - `mcollective_middleware_hosts`
 - `mcollective`
 - `mcollective_middleware_port`
 - `mcollective_middleware_user`
 - `mcollective_middleware_password`

Tip: If your PuppetDB includes outdated catalogs for nodes that aren't currently being managed, the installer might report that MCollective is active on those nodes. You can deactivate the nodes with `puppet node deactivate` or use Puppet to update the records.

After upgrade

- Manually remove these node groups:
 - **PE MCollective**
 - **PE ActiveMQ Broker**
 - Any custom node group you created for ActiveMQ hubs
- If you customized classification with references to MCollective or ActiveMQ profiles, remove the profiles from your classification. In this version of PE, nodes that include MCollective or ActiveMQ profiles trigger a warning during agent runs. Future versions of PE that remove the profiles completely can trigger failures in catalog compilation if you leave the profiles in place.

Removing MCollective

Remove MCollective and its related files from the nodes in your infrastructure. You must have PE version 2018.1.1 or later to complete this task.

Note: This procedure does not remove your MCollective or ActiveMQ log files.

1. In the console, click **Classification**, and select the node group **PE Infrastructure**.
2. On the **Configuration** tab, find the `puppet_enterprise` class. Select the `mcollective` parameter and edit its value to `absent`.

Class	Parameter	Value
<code>puppet_enterprise</code>	<code>mcollective</code>	<code>absent</code>

3. Click **Add parameter** and commit the change.
4. Set up a job and run on the **PE Infrastructure** node group to enforce your changes.

The server components of , including `pe-activemq` and the `peadmin` user, are removed from the master and the service on agents is stopped. You must complete the upgrade to 2019.0 or later to completely remove from agents.

Test modules before upgrade

To ensure that your modules work with the newest version of PE, update and test them with Puppet Development Kit (PDK) before upgrading.

Before you begin

If you are already using PDK, your modules should pass validation and unit tests with your currently installed version of PDK.

Update PDK with each new release to ensure compatibility with new versions of PE.

1. Download and install PDK. If you already have PDK installed, this updates PDK to its latest version. For detailed instructions and download links, see the [installing](#) instructions.

2. If you have not previously used PDK with your modules, convert them to a PDK compatible format. This makes changes to your module to enable validation and unit testing with PDK. For important usage details, see the [converting modules](#) documentation.

For example, from within the module directory, run:

```
pdk convert
```

3. If your modules are already compatible with PDK, update them to the latest module template. If you converted modules in step 2, you do not need to update the template. To learn more about updating, see the [updating module templates](#) documentation.

For example, from within the module directory, run:

```
pdk update
```

4. Validate and run unit tests for each module, specifying the version of PE you are upgrading to. When specifying a PE version, be sure to specify at least the year and the release number, such as 2018.1. For information about module validations and testing, see the [validating and testing modules](#) documentation.

For example, from within the module directory, run:

```
pdk validate
pdk test unit
```

The `pdk test unit` command verifies that testing dependencies and directories are present and runs the unit tests that you write. It does not create unit tests for your module.

5. If your module fails validation or unit tests, make any necessary changes to your code.

After you've verified that your modules work with the new PE version, you can continue with your upgrade.

Upgrade a standard installation

To upgrade, run the text-based PE installer on your master, and then upgrade any additional components. To upgrade with high availability enabled, you must also run the upgrade script on your replica.

Before you begin

Back up your installation.

If you encounter errors during upgrade, you can fix them and run the installer again.

1. [Download](#) the tarball appropriate to your operating system and architecture.
2. Unpack the installation tarball: `tar -xf <tarball>`
You need about 1 GB of space to untar the installer.
3. From the installer directory, run the installer: `sudo ./puppet-enterprise-installer`

Note: To specify a different `pe.conf` file other than the existing file, use the `-c` flag:

```
sudo ./puppet-enterprise-installer -c <FULL PATH TO pe.conf>
```

With this flag, your previous `pe.conf` is backed up to `/etc/puppetlabs/enterprise/conf.d/<TIMESTAMP>.conf` and a new `pe.conf` is created at `/etc/puppetlabs/enterprise/conf.d/pe.conf`.

4. If prompted, change the console administrator password by following the password reset link or running `puppet infrastructure console_password --password=<MY_PASSWORD>`

For security purposes, you're prompted to change the password if you didn't do so when you originally installed. If you don't change the password, the console administrator account is revoked.

5. If you have compilers, upgrade them.

SSH into each compiler and run:

```
/opt/puppetlabs/puppet/bin/curl --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem https://<PUPPET MASTER FQDN>:8140/packages/current/upgrade.bash |
sudo bash
```

6. Upgrade these additional infrastructure components.

- Agents
- client tools — Install the appropriate version of client tools that matches the version you upgraded to.

7. In high availability installations, upgrade your replica.

The replica is temporarily unavailable to serve as backup during this step, so you should time re-creating a replica to minimize risk.

- a) On your replica, run the upgrade script:

```
curl --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem https://
<MASTER_HOST>:8140/packages/current/upgrade.bash | bash
```

- b) Verify that master and replica services are operational:

```
/opt/puppetlabs/bin/puppet-infra status
```

- c) If your replica reports errors, reinitialize the replica:

```
/opt/puppetlabs/bin/puppet-infra reinitialize replica -y
```

Related information

[Backing up and restoring Puppet Enterprise](#) on page 697

Keep regular backups of your PE infrastructure. Backups allow you to more easily migrate to a new master, troubleshoot, and quickly recover in the case of system failures.

[Back up your infrastructure](#) on page 698

PE backup creates a copy of your Puppet infrastructure, including configuration, certificates, code, and PuppetDB.

[Forget a replica](#) on page 227

Forgetting a replica cleans up classification and database state, preventing degraded performance over time.

[Provision a replica](#) on page 223

Provisioning a replica duplicates specific components and services from the master to the replica.

[Enable a replica](#) on page 224

Enabling a replica activates most of its duplicated services and components, and instructs agents and infrastructure nodes how to communicate in a failover scenario.

Migrate from a split to a standard installation

Split installations, where the master, console, and PuppetDB are installed on separate nodes, are no longer supported. Migrate from an existing split installation to a standard (formerly called monolithic) installation—with or without compilers—and a standalone PE-PostgreSQL node.

Before you begin

You must be running a version of PE on all infrastructure nodes that includes the `puppet infrastructure run` command. To verify that this command is available on your systems, run `puppet infrastructure run --help`.

- The `puppet infrastructure run` command leverages built-in plans to automate certain management tasks. To use this command, you must be able to connect using SSH from your master to any nodes that the

command modifies. You can establish an SSH connection using key forwarding, a local key file, or by specifying keys in `.ssh/config` on your master. For more information, see [OpenSSH configuration options](#).

- You must have token-based authentication configured.

Important: The migration command used in this task automates a number of manual steps, including editing `pe.conf`, unpinning and uninstalling packages from affected infrastructure nodes, and running Puppet multiple times. Treat this process as you would any major migration by thoroughly testing it in an environment that's as similar to your production environment as possible.

1. On your master, verify that `pe.conf` contains correct information for `console_host`, `puppetdb_host`, and `database_host`.

The migration command uses this information to correctly migrate these nodes.

Note: If your split PE installation includes multiple standalone PuppetDB nodes, the migration command will fail with an error.

2. Make sure that the master can connect via SSH to your console node, PuppetDB node, and (if present) standalone PE-PostgreSQL node.
3. On your master logged in as root, run `puppet infrastructure run migrate_split_to_mono`
You can specify this optional parameter:

- `tmpdir` — Path to a directory to use for uploading and executing temporary files.

After completion, your master is running the console and PuppetDB services and you can retire or repurpose the old console node. If you **did not** start with a standalone PE-PostgreSQL node, your old PuppetDB node now functions in that capacity. If you **did** start with a standalone PE-PostgreSQL node, it continues to function in that capacity and you can retire or repurpose the old PuppetDB node.

Related information

[Generate a token using puppet-access](#) on page 243

Use `puppet-access` to generate an authentication tokens from the command line of any workstation (Puppet-managed or not), without the need to SSH into the Puppet master.

Upgrading PostgreSQL

If you use the default PE-PostgreSQL database installed alongside PuppetDB, you don't have to take special steps to upgrade PostgreSQL. However, if you have a standalone PE-PostgreSQL instance, or if you use a PostgreSQL instance not managed by PE, you must take extra steps to upgrade PostgreSQL.

You must upgrade a standalone PE-PostgreSQL instance each time you upgrade PE. To upgrade a standalone PE-PostgreSQL instance, simply run the installer on the PE-PostgreSQL node first, then proceed with upgrading the rest of your infrastructure.

You must upgrade a PostgreSQL instance not managed by PE only when there's an upgrade to PostgreSQL in PE, which occurred most recently in version 2019.2. To upgrade an unmanaged PostgreSQL instance, use one of these methods:

- Back up databases, wipe your old PostgreSQL installation, install the latest version of PostgreSQL, and restore the databases.
- Back up databases, set up a new node with the latest version of PostgreSQL, restore databases to the new node, and reconfigure PE to point to the new `database_host`.
- Run `pg_upgrade` to get from the older PostgreSQL version to the latest version.

Checking for updates

To see the version of PE you're currently using, run `puppet --version` on the command line. Check the PE download site to find information about the latest maintenance release.

Note: By default, the master checks for updates whenever the `pe-puppetserver` service restarts. As part of the check, it passes some basic, anonymous information to Puppet servers. You can optionally disable update checking.

Upgrading agents

Upgrade your agents as new versions of Puppet Enterprise become available. The `puppet_agent` module helps automate upgrades, and provides the safest upgrade. Alternatively, you can upgrade individual nodes using a script.

Note: Before upgrading agents, verify that the master and agent software versions are compatible. Then after upgrade, run Puppet on your agents as soon as possible to verify that agents have the correct configuration and that your systems are behaving as expected.

Setting your desired agent version

To upgrade your master but use an older agent version that is still compatible with the new master, define a `pe_repo::platform::<AGENT_OS_VERSION_ARCHITECTURE>` class with the `agent_version` variable set to your desired agent version.

To ensure your agents are always running the same version as your master, in the `puppetlabs-puppet_agent` module, set the `package_version` variable for the `puppet_agent` class to `auto`. This causes agents to automatically upgrade themselves on their first Puppet run after a master upgrade.

Related information

[Upgrading Puppet Enterprise](#) on page 179

Upgrade your PE installation as new versions become available.

Upgrade *nix or Windows agents using the puppet_agent module

The `puppetlabs-puppet_agent` module, available from the Forge, enables you to upgrade multiple agents at one time. The module handles all the latest version to version upgrades.

When upgrading agents, first test the upgrade on a subset of agents, and after you verify the upgrade, upgrade remaining agents.

1. On your master, download and install the `puppetlabs-puppet_agent` module: `puppet module install puppetlabs-puppet_agent`
2. Configure the master to download the agent version you want to upgrade.
 - a) In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Master** group.
 - b) On the **Configuration** tab, in the **Add a new class** field, enter `pe_repo`, and select the appropriate repo class from the list of classes.

Repo classes are listed as `pe_repo::platform::<AGENT_OS_VERSION_ARCHITECTURE>`.

To specify a particular agent version, set the `agent_version` variable using an X.Y.Z format (for example, 5.5.14). When their version is set explicitly, agents do not automatically upgrade when you upgrade your master.

- c) Click **Add class** and commit changes.
- d) On your master, run to configure the newly assigned class: `puppet agent -t`

The new repo is created in `/opt/puppetlabs/server/data/packages/public/<PE_VERSION>/<PLATFORM>/`.

3. Click **Classification**, click **Add group**, specify options for a new upgrade node group, and then click **Add**.
 - **Parent name** — Select the name of the classification node group that you want to set as the parent to this group, in this case, **All Nodes**.
 - **Group name** — Enter a name that describes the role of this classification node group, for example, `agent_upgrade`.
 - **Environment** — Select the environment your agents are in.
 - **Environment group** — *Do not* select this option.
4. Click the link to **Add membership rules, classes, and variables**.

- On the **Rules** tab, create a rule to add the agents that you want to upgrade to this group, click **Add Rule**, and then commit changes.

For example:

- Fact** — **osfamily**
- Operator** — **=**
- Value** — **RedHat**

- Still in the agent upgrade group, click the **Configuration** tab, and in the **Add new class** field, add the **puppet_agent** class, and click **Add class**.

If you don't immediately see the class, click **Refresh** to update the classifier.

Note: If you've changed the prefix parameter of the **pe_repo** class in your **PE Master** node group, set the **puppet-agent source** parameter of the upgrade group to `https://<MASTER_HOSTNAME>:8140/<Prefix>`.

- In the **puppet_agent** class, specify the version of the puppet-agent package version that you want to install, then commit changes.

Parameter	Value
package_version	<p>The puppet-agent package version to install, for example 5.3.3.</p> <p>Set this parameter to <code>auto</code> to install the same agent version that is installed on your master.</p>

- On the agents that you're upgrading, run Puppet: `/opt/puppet/bin/puppet agent -t`

After the Puppet run, you can verify the upgrade with `/opt/puppetlabs/bin/puppet --version`

Upgrade a *nix or Windows agent using a script

To upgrade an individual node, for example to test or troubleshoot, you can upgrade directly from the node using a script. This method relies on a package repository hosted on your master.

Note: If you encounter SSL errors during the upgrade process, ensure your agent's OpenSSL is up to date and matches the master's version. You can check the master's OpenSSL versions with `/opt/puppetlabs/puppet/bin/openssl version` and the agent's version with `openssl version`.

Upgrade a *nix agent using a script

You can upgrade an individual *nix agent using a script.

- Configure the master to download the agent version you want to upgrade.
 - In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Master** group.
 - On the **Configuration** tab, in the **Add a new class** field, enter `pe_repo`, and select the appropriate repo class from the list of classes.

Repo classes are listed as `pe_repo::platform::<AGENT_OS_VERSION_ARCHITECTURE>`.

To specify a particular agent version, set the `agent_version` variable using an X.Y.Z format (for example, 5.5.14). When their version is set explicitly, agents do not automatically upgrade when you upgrade your master.

- Click **Add class** and commit changes.
- On your master, run to configure the newly assigned class: `puppet agent -t`

The new repo is created in `/opt/puppetlabs/server/data/packages/public/<PE_VERSION>/<PLATFORM>/`.

- SSH into the agent node you want to upgrade.

3. Run the upgrade command appropriate to the operating system.

- Most

```
/opt/puppetlabs/puppet/bin/curl --cacert /etc/puppetlabs/puppet/
ssl/certs/ca.pem https://<MASTER HOSTNAME>:8140/packages/current/
install.bash | sudo bash
```

- , , and

```
curl --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem https://<MASTER
HOSTNAME>:8140/packages/current/install.bash | sudo bash
```

services restarts automatically after upgrade.

Upgrade a Windows agent using a script

You can upgrade an individual Windows agent using a script. For Windows, this method is riskier than using the `puppet_agent` module to upgrade, because you must manually complete and verify steps that the module handles automatically.

Note: The `<MASTER HOSTNAME>` portion of the installer script—as provided in the following example—refers to the FQDN of the master. The FQDN must be fully resolvable by the machine on which you're installing or upgrading the agent.

1. Stop the Puppet service and the PXP agent service.
2. On the Windows agent, run the install script:

```
[Net.ServicePointManager]::ServerCertificateValidationCallback
= {$true}; $webClient = New-Object System.Net.WebClient;
$webClient.DownloadFile('https://<MASTER HOSTNAME>:8140/packages/current/
install.ps1', 'install.ps1'); .\install.ps1
```

3. Verify that Puppet runs are complete.
4. Restart the Puppet service and the PXP agent service.

Upgrading the agent independent of PE

You can optionally upgrade the agent to a newer version than the one packaged with your current PE installation.

For details about Puppet agents versions that are tested and supported for PE, see the PE component version table.

The agent version is specified on a platform-by-platform basis in the **PE Master** node group, in any `pe_repo::platform` class, using the `agent_version` parameter.

When you install new nodes or upgrade existing nodes, the agent install script installs the version of the agent specified for its platform class. If a version isn't specified for the node's platform, the script installs the default version packaged with your current version of PE.

Note: To install nodes without internet access, download the agent tarball for the version you want to install, as specified using the `agent_version` parameter.

The platform in use on your master requires special consideration. The agent version used on your master must match the agent version used on other infrastructure nodes, including compilers and replicas, otherwise your master won't compile catalogs for these nodes.

To keep infrastructure nodes synced to the same agent version, if you specify a newer `agent_version` for your master platform, you must either:

- (Recommended) Upgrade the agent on your master—and any existing infrastructure nodes—to the newer agent version. You can upgrade these nodes by running the agent install script.
- Manually install the older agent version used on your master on any new infrastructure nodes you provision. You **can't** install these nodes using the agent install script, because the script uses the agent version specified

for the platform class, instead of the master's current agent version. Manual installation requires configuring `puppet.conf`, DNS alt names, CSR attributes, and other relevant settings.

Related information

[Component versions in recent PE releases](#) on page 21

These tables show which components are in Puppet Enterprise (PE) releases, covering recent long-term supported (LTS) releases. To see component version tables for a release that has passed its support phase, switch to the docs site for that release.

Configuring Puppet Enterprise

Methods for configuring Puppet Enterprise

After you've installed Puppet Enterprise (PE), optimize it for your environment by configuring and tuning settings as needed. For example, you might want to add your own certificate to the whitelist, increase the max-threads setting for http and https requests, or configure the number of JRuby instances.

PE shares configuration settings used in open source Puppet and documented in the [Configuration Reference](#), however PE defaults for certain settings might differ from the Puppet defaults. Some examples of settings that have different PE defaults include `disable18n`, `environment_timeout`, `always_retry_plugins`, and the Puppet Server JRuby `max-active-instances` setting. To verify PE configuration defaults, check the `puppet.conf` file after installation.

There are three main methods for configuring PE: using the console, adding a key to Hiera, or editing `pe.conf`. Be consistent with the method you choose, unless the situation calls for you to use a specific method over the others.

Important: When you enable high availability, you must use `or pe.conf` only — not the console — to specify configuration parameters. Using `pe.conf` or `ensures` that configuration is applied to both your master and replica.

Related information

[Configuring and tuning Puppet Server](#) on page 191

After you've installed Puppet Enterprise, optimize it for your environment by configuring and tuning Puppet Server settings as needed.

[Configuring and tuning PuppetDB](#) on page 200

After you've installed Puppet Enterprise, optimize it for your environment by configuring and tuning PuppetDB configuration as needed.

[Configuring and tuning the console](#) on page 195

After installing Puppet Enterprise, you can change product settings to customize the console's behavior, adjust to your team's needs, and improve performance. Many settings can be configured in the console itself.

[Configuring and tuning orchestration](#) on page 202

After installing PE, you can change some default settings to further configure the orchestrator and `pe-orchestration-services`.

[Configuring Java arguments](#) on page 205

You might need to increase the Java Virtual Machine (JVM) memory allocated to Java services to improve performance in your Puppet Enterprise (PE) deployment.

Configure settings using the console

The console allows you to use a graphical interface to configure Puppet Enterprise (PE).

Changes in the console will override your Hiera data and data in `pe.conf`. It is usually best to use the console when you want to:

- Change parameters in profile classes starting with `puppet_enterprise::profile`.

- Add any parameters in PE-managed configuration files.
- Set parameters that configure at runtime.

To configure settings in the console:

1. Click **Classification**, and select the node group that contains the class you want to work with.
2. On the **Configuration** tab, find the class you want to work with, select the **Parameter name** from the list and edit its value.

If you wanted to change the number used to identify the port your console is on from the default 443 to 500, change the parameter value in the following:

Class	Parameter	Value
puppet_enterprise::profile::console	console_listen_port	[500]

3. Click **Add parameter** and commit changes.
4. On the nodes hosting the master and console, run Puppet.

Related information

[Preconfigured node groups](#) on page 331

Puppet Enterprise includes preconfigured node groups that are used to manage your configuration.

Configure settings with Hiera

Hiera is a hierarchy-based method of configuration management that relies on a “defaults, with overrides” system. When you add a parameter or setting to your Hiera data, Hiera searches through the data in the order it was written to find the value you want to change. Once found, it overrides the default value with the new parameter or setting. You can use Hiera to manage your PE configuration settings.

Before you begin

For more information on how to use Hiera, see the [Hiera docs](#).

Changes to PE configuration in Hiera will override configuration settings in `pe.conf`, but not those set in the console. It's best to use Hiera when you want to:

- Change parameters in non-profile classes.
- Set parameters that are static and version controlled.
- Configure for high availability.

To configure a setting using Hiera:

1. Open your default data file.

The default location for Hiera data files is:

- *nix: `/etc/puppetlabs/code/environments/<ENVIRONMENT>/data/common.yaml`
- Windows: `%CommonAppData%\PuppetLabs\code\environments\<ENVIRONMENT>\data\common.yaml`

If you customize the `hiera.yaml` configuration to change location for data files (the `datadir` setting) or the path of the common data file (in the `hierarchy` section), look for the default `.yaml` file in the customized location.

2. Add your new parameter to the file in editor.

For example, to increase the number of seconds before a node is considered unresponsive from the default 3600 to 4000, add the following to your `.yaml` default file and insert your new parameter at the end.

```
Puppet_enterprise::console_services::no_longer_reporting_cutoff: 4000
```

3. To compile changes, run `puppet agent -t`

Related information

[Preconfigured node groups](#) on page 331

Puppet Enterprise includes preconfigured node groups that are used to manage your configuration.

Configure settings in `pe.conf`

Puppet Enterprise (PE) configuration data includes any data set in `/etc/puppetlabs/enterprise/conf.d/` but `pe.conf` is the file used for most configuration activities during installation.

PE configuration settings made in Hiera and the console always override settings made in `pe.conf`. Configure settings using `pe.conf` when you want to:

- Access settings during installation.
- Configure for high availability.

To configure settings using `pe.conf`:

1. Open the `pe.conf` file on your master:

```
/etc/puppetlabs/enterprise/conf.d/pe.conf
```

2. Add the parameter and new value you want to set.

For example, to change the proxy in your repo, add the following and change the parameter to your new proxy location.

```
pe_repo::http_proxy_host: "proxy.example.vlan"
```

3. Run `puppet agent -t`

Note: If PE services are stopped, run `puppet infrastructure configure` instead of `puppet agent -t`.

Configuring and tuning Puppet Server

After you've installed Puppet Enterprise, optimize it for your environment by configuring and tuning Puppet Server settings as needed.

Related information

[Methods for configuring Puppet Enterprise](#) on page 189

After you've installed Puppet Enterprise (PE), optimize it for your environment by configuring and tuning settings as needed. For example, you might want to add your own certificate to the whitelist, increase the max-threads setting for http and https requests, or configure the number of JRuby instances.

[Tuning standard installations](#) on page 208

Use these guidelines to configure your installation to maximize its use of available system (CPU and RAM) resources.

[Increase the Java heap size for PE Java services](#) on page 206

The Java heap size is the Java Virtual Machine (JVM) memory allocated to Java services in Puppet Enterprise (PE). You can use any configuration method you choose. We will use the console to change the Java heap size for console services, Puppet Server, orchestration services, or PuppetDB in the examples below.

[Configure ulimit for PE services](#) on page 207

The various services in PE require up to one file handle per connected client. On most operating system configurations, the defaults are not adequate for more than a couple hundred clients. To support more clients, you need to increase the number of allowed file handles.

Tune the maximum number of JRuby instances

The `jruby_max_active_instances` setting controls the maximum number of JRuby instances to allow on the Puppet Server.

The default used in PE is the number of CPUs - 1, expressed as `$::processorcount - 1`. One instance is the minimum value and four instances is the maximum value. Four JRuby instances works for most environments.

Because increasing the maximum number of JRuby instances also increases the amount of RAM used by `pe-puppetserver`, increase your heap size. We conservatively estimate that a JRuby process uses 512MB of RAM.

To change the number of instances using Hiera:

1. Add the following code to your default `.yaml` file and set the desired number of instances:

```
puppet_enterprise::master::puppetserver::jruby_max_active_instances:
  <number of instances>
```

2. To compile the changes, run `puppet agent -t`

Tune the maximum requests per JRuby instance

The `max_requests_per_instance` setting determines the maximum number of requests per instance of a JRuby interpreter before it is killed.

The appropriate value for this parameter depends on how busy your servers are and how much you are affected by a memory leak. By default, `max_requests_per_instance` is set to 100,000 in PE.

When a JRuby interpreter is killed, all of its memory is reclaimed and it is replaced in the pool with a new interpreter. This prevents any one interpreter from consuming too much RAM, mitigating Puppet code memory leak issues and keeping Puppet Server up.

Starting a new interpreter has a performance cost, so set the parameter to get a new interpreter no more than every few hours. There are multiple interpreters running with requests balanced across them, so the lifespan of each interpreter varies.

To increase the maximum requests per instance using Hiera:

1. Add the following code to your default `.yaml` and set the desired number of requests.

```
puppet_enterprise::master::puppetserver::jruby_max_requests_per_instance:
  <number of requests>
```

2. To compile the changes, run `puppet agent -t`

Tune the Ruby load path

The `ruby_load_path` setting determines where Puppet Server finds components such as Puppet and Facter.

The default setting is located at `$puppetserver_jruby_puppet_ruby_load_path = ['/opt/puppetlabs/puppet/lib/ruby/vendor_ruby', '/opt/puppetlabs/puppet/cache/lib']`.

To change the path to a different array in `pe.conf`:

1. Add the following to your `pe.conf` file on your master and set your new load path parameter.

```
puppet_enterprise::master::puppetserver::puppetserver_jruby_puppet_ruby_load_path
```

2. Run `puppet agent -t`

Note that if you change the `libdir` you must also change the `vardir`.

Enable or disable cached data when updating classes

The optional `environment-class-cache-enabled` setting specifies whether cached data is used when updating classes in the console. When `true`, Puppet Server refreshes classes using file sync, improving performance.

The default value for `environment-class-cache-enabled` depends on whether you use Code Manager.

- With Code Manager, the default value is enabled (`true`). File sync clears the cache automatically in the background, so clearing the environment cache manually isn't required when using Code Manager.
- Without Code Manager, the default value is disabled (`false`).

Note: If you're not using Code Manager and opt to enable this setting, make sure your code deployment method — for example `r10k` — clears the environment cache when it completes. If you don't clear the environment cache, the Node Classifier doesn't receive new class information until Puppet Server is restarted.

To enable or disable the cache using Hiera:

1. Add the following code to your default `.yaml` file and set the parameter to the appropriate setting.

```
puppet_enterprise::master::puppetserver::
  jruby_environment_class_cache_enabled: <true OR false>
```

2. To compile the changes, run `puppet agent -t`

Change the `environment_timeout` setting

The `environment_timeout` setting controls how long the master caches data it loads from an environment, determining how much time passes before changes to an environment's Puppet code are reflected in its environment.

In PE, the `environment_timeout` is set to 0. This lowers the performance of your master but makes it easy for new users to deploy updated Puppet code. Once your code deployment process is mature, change this setting to unlimited.

Note: When you install Code Manager and set the `code_manager_auto_configure` parameter to `true`, `environment_timeout` is updated to unlimited

To change the `environment_timeout` setting using `pe.conf`:

1. Add the following to your `pe.conf` file on your master and specify either 0 or unlimited:

```
puppet_enterprise::master::environment_timeout:<time>
```

2. Run `puppet agent -t`

For more information, see [Environments limitations](#)

Add certificates to the `puppet-admin` certificate whitelist

Add trusted certificates to the `puppet-admin` certificate whitelist.

To add certificates to the whitelist using `pe.conf`:

1. Add the following code to your `pe.conf` file on your master and add the desired certificates.

```
puppet_enterprise::master::puppetserver::puppet_admin_certs:'example_cert_name'
```

2. Run `puppet agent -t`

Disable update checking

Puppet Server (`pe-puppetserver`) checks for updates when it starts or restarts, and every 24 hours thereafter. It transmits basic, anonymous info to our servers at Puppet, Inc. to get update information. You can optionally turn this off.

Specifically, it transmits:

- Product name
- Puppet Server version
- IP address
- Data collection timestamp

To turn off update checking using the console:

1. Open the console, click **Classification**, and select the PE Master node group.
2. On the **Configuration** tab, find the `puppet_enterprise::profile::master` class and find the `check_for_updates` parameter from the list and change its value to `false`.

3. Click **Add parameter** and commit changes.
4. On the nodes hosting the master and console, run Puppet.

Puppet Server configuration files

At startup, Puppet Server reads all of the `.conf` files in the `conf.d` directory (`/etc/puppetlabs/puppetserver/conf.d`).

The `conf.d` directory contains the following files:

File name	Description
<code>auth.conf</code>	Contains authentication rules and settings for agents and API endpoint access.
<code>global.conf</code>	Contains global configuration settings for Puppet Server, including logging settings.
<code>metrics.conf</code>	Contains settings for Puppet Server metrics services.
<code>pe-puppet-server.conf</code>	Contains Puppet Server settings specific to Puppet Enterprise.
<code>webserver.conf</code>	Contains SSL service configuration settings.
<code>ca.conf</code>	(Deprecated) Contains rules for Certificate Authority services. Superseded by <code>webserver.conf</code> and <code>auth.conf</code> .

For information about Puppet Server configuration files, see [Puppet Server's config files](#) for the Puppet Server version you're using, and the Related information links below.

Related information

[Viewing and managing Puppet Server metrics](#) on page 294

Puppet Server can provide performance and status metrics to external services for monitoring server health and performance over time.

pe-puppet-server.conf settings

The `pe-puppet-server.conf` file contains Puppet Server settings specific to Puppet Enterprise, with all settings wrapped in a `ruby-puppet` section.

gen-home

Determines where JRuby looks for gems. It is also used by the `puppetserver gem` command line tool.

Default: `/opt/puppetlabs/puppet/cache/jruby-gems`

master-conf-dir

Sets the Puppet configuration directory's path.

Default: `/etc/puppetlabs/puppet`

master-var-dir

Sets the Puppet variable directory's path.

Default: `/opt/puppetlabs/server/data/puppetserver`

max-queued-requests

Optional. Sets the maximum number of requests that can be queued waiting to borrow a from the pool. After this limit is exceeded, a 503 `Service Unavailable` response is 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 response includes a `Retry-After` header indicating a random sleep time after which the client can retry the request.

Note: Don't use this solution if your managed infrastructure includes a significant number of agents older than Puppet 5.3. Older agents treat a 503 response as a failure, which ends their runs, causing groups of older agents to schedule their next runs at the same time, creating a thundering herd problem.

Default: 0

max-retry-delay

Optional. Sets the upper limit in seconds for the random sleep set as a `Retry-After` header on 503 responses returned when `max-queued-requests` is enabled.

Default: 1800

jruby_max_active_instances

Controls the maximum number of JRuby instances to allow on the Puppet Server.

Default: 4

max_requests_per_instance

Sets the maximum number of requests per instance of a JRuby interpreter before it is killed.

Default: 100000

ruby-load-path

Sets the Puppet configuration directory's path. The agent's `libdir` value is added by default.

Default: `['/opt/puppetlabs/puppet/lib/ruby/vendor_ruby', '/opt/puppetlabs/puppet/cache/lib']`

Configuring and tuning the console

After installing Puppet Enterprise, you can change product settings to customize the console's behavior, adjust to your team's needs, and improve performance. Many settings can be configured in the console itself.

Related information

[Methods for configuring Puppet Enterprise](#) on page 189

After you've installed Puppet Enterprise (PE), optimize it for your environment by configuring and tuning settings as needed. For example, you might want to add your own certificate to the whitelist, increase the `max-threads` setting for `http` and `https` requests, or configure the number of JRuby instances.

[Disable update checking](#) on page 193

Puppet Server (`pe-puppetserver`) checks for updates when it starts or restarts, and every 24 hours thereafter. It transmits basic, anonymous info to our servers at Puppet, Inc. to get update information. You can optionally turn this off.

[Configuring Java arguments](#) on page 205

You might need to increase the Java Virtual Machine (JVM) memory allocated to Java services to improve performance in your Puppet Enterprise (PE) deployment.

[Configure ulimit for PE services](#) on page 207

The various services in PE require up to one file handle per connected client. On most operating system configurations, the defaults are not adequate for more than a couple hundred clients. To support more clients, you need to increase the number of allowed file handles.

Configure the PE console and console-services

Configure the behavior of the console and console-services, as needed.

Note: Use the `Hiera` or `pe.conf` method to configure non-profile classes, such as `puppet_enterprise::api_port` and `puppet_enterprise::console_services::no_longer_reporting_cutoff`.

To configure settings in the console:

1. Click **Classification**, and select the node group that contains the class you want to work with.

2. On the **Configuration** tab, find the class you want to work with, select the **Parameter name** from the list and edit its value.
3. Click **Add parameter** and commit changes.
4. On the nodes hosting the master and console, run Puppet.

Related information

[Running Puppet on nodes](#) on page 320

Puppet automatically attempts to run on each of your nodes every 30 minutes. To trigger a Puppet run outside of the default 30-minute interval, you can manually run Puppet.

Console and console-services parameters

Use these parameters to customize the behavior of the console and console-services. Parameters that begin with `puppet_enterprise::profile` can be modified from the console itself. See the configuration methods documents for more information on how to change parameters in the console or Hiera.

`puppet_enterprise::profile::console::classifier_synchronization_period`

Integer representing, in seconds, the classifier synchronization period, which controls how long it takes the node classifier to retrieve classes from the master.

Default: "600" (seconds).

`puppet_enterprise::profile::console::rbac_failed_attempts_lockout`

Integer specifying how many failed login attempts are allowed on an account before that account is revoked.

Default: "10" (attempts).

`puppet_enterprise::profile::console::rbac_password_reset_expiration`

Integer representing, in hours, how long a user's generated token is valid for. An administrator generates this token for a user so that they can reset their password.

Default: "24" (hours).

`puppet_enterprise::profile::console::rbac_session_timeout`

Integer representing, in minutes, how long a user's session can last. The session length is the same for node classification, RBAC, and the console.

Default: "60" (minutes).

`puppet_enterprise::profile::console::session_maximum_lifetime`

Integer representing the maximum allowable period that a console session can be valid. To not expire before the maximum token lifetime, set to '0'.

Supported units are "s" (seconds), "m" (minutes), "h" (hours), "d" (days), "y" (years). Units are specified as a single letter following an integer, for example "1d"(1 day). If no units are specified, the integer is treated as seconds.

`puppet_enterprise::profile::console::console_ssl_listen_port`

Integer representing the port that the console is available on.

Default: [443]

`puppet_enterprise::profile::console::ssl_listen_address`

Nginx listen address for the console.

Default: "0.0.0.0"

`puppet_enterprise::profile::console::classifier_prune_threshold`

Integer representing the number of days to wait before pruning the size of the classifier database. If you set the value to "0", the node classifier service is never pruned.

`puppet_enterprise::profile::console::classifier_node_check_in_storage`

"true" to store an explanation of how nodes match each group they're classified into, or "false".

Default: "false"

puppet_enterprise::profile::console::display_local_time

"true" to display timestamps in local time, with hover text showing UTC time, or "false" to show timestamps in UTC time.

Default: "false"

Modify these configuration parameters in `Hiera` or `pe.conf`, not the console:

puppet_enterprise::api_port

SSL port that the node classifier is served on.

Default: [4433]

puppet_enterprise::console_services::no_longer_reporting_cutoff

Length of time, in seconds, before a node is considered unresponsive.

Default: 3600 (seconds)

console_admin_password

The password to log into the console, for example "myconsolepassword".

Default: Specified during installation.

Manage the HTTPS redirect

By default, the console redirects to HTTPS when you attempt to connect over HTTP. You can customize the redirect target URL or disable redirection.

Customize the HTTPS redirect target URL

By default, the redirect target URL is the same as the FQDN of your master, but you can customize this redirect URL.

To change the target URL in the console:

1. Click **Classification**, and select the **PE Infrastructure** node group.
2. On the **Configuration** tab, find the `puppet_enterprise::profile::console::proxy::http_redirect` class, select the `server_name` parameter from the list, and change its value to the desired server.
3. Click **Add parameter** and commit changes.
4. On the nodes hosting the master and console, run Puppet.

Disable the HTTPS redirect

The pe-nginx webserver listens on port 80 by default. If you need to run your own service on port 80, you can disable the HTTPS redirect.

1. Add the following to your default `.yaml` file with the parameter set to false.

```
puppet_enterprise::profile::console::proxy::http_redirect::enable_http_redirect:
false
```

2. To compile changes, run `puppet agent -t` on the master.

Tuning the PostgreSQL buffer pool size

If you are experiencing performance issues or instability with the console, adjust the buffer memory settings for PostgreSQL.

The most important PostgreSQL memory settings for PE are `shared_buffers` and `work_mem`.

Parameter	Value
<code>shared_buffers</code>	in MB. Set at about 25 percent of your hardware's RAM.

work_mem	In MB. Increase the value from the default for large deployments.
-----------------	---

To change the PostgreSQL memory settings using the console:

1. Click **Classification**, and in the **PE Infrastructure** group, select **PE Database** group.
2. On the **Configuration** tab, find the **shared_buffers** and **work_mem** parameters from the drop down and edit their values as desired.
3. Click **Add parameter** and commit changes.
4. On the nodes hosting the master and console, run Puppet.
5. Restart the PostgreSQL server: `sudo /etc/init.d/pe-postgresql restart`

Enable data editing in the console

The ability to edit configuration data in the console is enabled by default in new installations. If you upgrade from an earlier version and didn't previously have configuration data enabled, you must manually enable classifier configuration data, because enabling requires edits to your `hiera.yaml` file.

On your master, edit `/etc/puppetlabs/puppet/hiera.yaml` to add:

```
hierarchy:
- name: "Classifier Configuration Data"
  data_hash: classifier_data
```

Place any additional hierarchy entries, such as `hiera-yaml` or `hiera-eyaml` under the same hierarchy key, preferably below the `Classifier Configuration Data` entry.

Note: If you enable data editing in the console, add both **Set environment** and **Edit configuration data** to groups that set environment or modify class parameters in order for users to make changes.

If your environment is configured for high availability, you must also update `hiera.yaml` on your replica.

Configuring and tuning security settings

Ensure your PE environment is secure by configuring security settings.

Configure cipher suites

Due to regulatory compliance or other security requirements, you may need to change which cipher suites your SSL-enabled PE services use to communicate with other PE components.

SSL ciphers for core Puppet services

To add or remove cipher suites for [core Puppet services](#), use Hiera to add an array of SSL ciphers to the `puppet_enterprise::ssl_cipher_suites` parameter.

Note: Changing this parameter overrides the default list of SSL cipher suites.

The example Hiera data below replaces the default list of cipher suites to only allow the four specified.

```
puppet_enterprise::ssl_cipher_suites:
- 'SSL_RSA_WITH_NULL_MD5'
- 'SSL_RSA_WITH_NULL_SHA'
- 'TLS_DH_anon_WITH_AES_128_CBC_SHA'
- 'TLS_DH_anon_WITH_AES_128_CBC_SHA256'
```

Note: Cipher names are in [IANA RFC naming format](#).

SSL for console services

To add or remove cipher suites for console services affecting traffic on port 443, use Hiera or the console to change the `puppet_enterprise::profile::console::proxy::ssl_ciphers` parameter.

For example, to change the parameter in the console, in the **PE Console** node group, add an array of SSL ciphers to the `ssl_ciphers` parameter in the `puppet_enterprise::profile::console::proxy` class.

Configure SSL protocols

Add or remove SSL protocols in your PE infrastructure.

To change what SSL protocols your PE infrastructure uses, use Hiera or the console to add or remove protocols.

Use the parameter `puppet_enterprise::master::puppetserver::ssl_protocols` and add an array for protocols you want to include, or remove protocols you no longer want to use.

For example, to enable TLSv1.1 and TLSv1.2, set the following parameter in the **PE Infrastructure** group in the console or in your Hiera data.

```
puppet_enterprise::master::puppetserver::ssl_protocols[ "TLSv1.1", "TLSv1.2" ]
```

Note: To comply with security regulations, PE 2019.1 and later uses only version 1.2 of the Transport Layer Security (TLS) protocol.

Configure RBAC and token-based authentication settings

Tune RBAC and token-based authentication settings, like setting the number of failed attempts a user has before they are locked out of the console or changing the amount of time a token is valid for.

RBAC and token authentication settings can be changed in the **PE Infrastructure** group in the console or in your Hiera data. Below is a list of related settings.

`puppet_enterprise::profile::console::rbac_failed_attempts_lockout`

An integer specifying how many failed login attempts are allowed on an account before the account is revoked. The default is "10" (attempts).

`puppet_enterprise::profile::console::rbac_password_reset_expiration`

An integer representing, in hours, how long a user's generated token is valid for. An administrator generates this token for a user so that they can reset their password. The default is "24" (hours).

`puppet_enterprise::profile::console::rbac_session_timeout`

Integer representing, in minutes, how long a user's session can last. The session length is the same for node classification, RBAC, and the console. The default is "60" (minutes).

`puppet_enterprise::profile::console::rbac_token_auth_lifetime`

A value representing the default authentication lifetime for a token. It cannot exceed the `rbac_token_maximum_lifetime`. This is represented as a numeric value followed by "y" (years), "d" (days), "h" (hours), "m" (minutes), or "s" (seconds). The default is "5m".

`puppet_enterprise::profile::console::rbac_token_maximum_lifetime`

A value representing the maximum allowable lifetime for all tokens. This is represented as a numeric value followed by "y" (years), "d" (days), "h" (hours), "m" (minutes), or "s" (seconds). The default is "10y".

`puppet_enterprise::profile::console::rbac_account_expiry_check_minutes`

An integer that specifies, in minutes, how often the application checks for idle user accounts. The default value is "60" (minutes).

`puppet_enterprise::profile::console::rbac_account_expiry_days`

An integer that specifies, in days, the duration before an inactive user account expires. The default is undefined. To activate the feature, add a value of "1" or greater.

If a non-superuser hasn't logged into the console during this specified period, their user status updates to revoked. After creating an account, if a non-superuser hasn't logged in to the console during the specified period, their user status updates to revoked.

Note: If you do not specify the `rbac_account_expiry_days` parameter, the `rbac_account_expiry_check_minutes` parameter is ignored.

Configuring and tuning PuppetDB

After you've installed Puppet Enterprise, optimize it for your environment by configuring and tuning PuppetDB configuration as needed.

This page covers a few key topics, but additional settings and information about configuring PuppetDB is available in the [PuppetDB configuration documentation](#). Be sure to check that the PuppetDB docs version you're looking at matches the one version of PuppetDB in your PE.

Related information

[Methods for configuring Puppet Enterprise](#) on page 189

After you've installed Puppet Enterprise (PE), optimize it for your environment by configuring and tuning settings as needed. For example, you might want to add your own certificate to the whitelist, increase the max-threads setting for http and https requests, or configure the number of JRuby instances.

[Tuning standard installations](#) on page 208

Use these guidelines to configure your installation to maximize its use of available system (CPU and RAM) resources.

[Configure settings with Hieradata](#) on page 190

Hiera is a hierarchy-based method of configuration management that relies on a “defaults, with overrides” system. When you add a parameter or setting to your Hiera data, Hiera searches through the data in the order it was written to find the value you want to change. Once found, it overrides the default value with the new parameter or setting. You can use Hiera to manage your PE configuration settings.

[Configure ulimit for PE services](#) on page 207

The various services in PE require up to one file handle per connected client. On most operating system configurations, the defaults are not adequate for more than a couple hundred clients. To support more clients, you need to increase the number of allowed file handles.

[Increase the Java heap size for PE Java services](#) on page 206

The Java heap size is the Java Virtual Machine (JVM) memory allocated to Java services in Puppet Enterprise (PE). You can use any configuration method you choose. We will use the console to change the Java heap size for console services, Puppet Server, orchestration services, or PuppetDB in the examples below.

Configure agent run reports

By default, every time Puppet runs, the master generates agent run reports and submits them to PuppetDB. You can enable or disable this as needed.

To enable or disable agent run reports using the console:

1. Click **Classification**, and in the **PE Infrastructure** group, select the **PE Master** group.
2. On the **Configuration** tab, add the `puppet_enterprise::profile::master::puppetdb` class, select the `report_processor_ensure` parameter, and enter the value `present` to enable agent run reports or `absent` to disable agent run reports.
3. Click **Add parameter** and commit changes.
4. On the nodes hosting the master and console, run Puppet.

Configure command processing threads

The `command_processing_threads` setting defines how many command processing threads PuppetDB uses to sort incoming data. Each thread can process a single command at a time. The setting defaults to half the number of cores in your system.

To set the number of threads using `pe.conf`:

1. Add the following code to your `pe.conf` file on your master and set your new parameter.

```
puppet_enterprise::puppetdb::command_processing_threads: <number of threads>
```

2. Run `puppet agent -t`

Configure how long before PE stops managing deactivated nodes

Use the `node-purge-ttl` parameter to set the "length of time" value before PE automatically removes nodes that have been deactivated or expired. This also removes all facts, catalogs, and reports for the relevant nodes.

To change the amount of time before nodes are purged using the console:

1. Click **Classification**, and in the PE Infrastructure group, select the PE Database group.
2. On the **Configuration** tab, find the `puppet_enterprise::profile::puppetdb` class, find the `node_purge_ttl` parameter, and change its value to the desired amount of time.

To change the unit of time, use the following suffixes:

- d - days
- h - hours
- m - minutes
- s - seconds
- ms - milliseconds

For example, to set the purge time to 14 days:

```
puppet_enterprise::profile::puppetdb::node_purge_ttl: '14d'
```

3. Click **Add parameter** and commit changes.
4. On the nodes hosting the master and console, run Puppet.

Change the PuppetDB user password

The console uses a database user account to access its PostgreSQL database. Change it if it is compromised or to comply with security guidelines.

To change the password:

1. Stop the `pe-puppetdb` puppet service by running `puppet resource service pe-puppetdb ensure=stopped`
2. On the database server (which might or might not be the same as PuppetDB, depending on your deployment's architecture), use the PostgreSQL administration tool of your choice to change the user's password. With the standard PostgreSQL client, you can do this by running `ALTER USER console PASSWORD '<new password>';`
3. Edit `/etc/puppetlabs/puppetdb/conf.d/database.ini` on the PuppetDB server and change the `password:` line under `common` or `production`, depending on your configuration, to contain the new password.
4. Start the `pe-puppetdb` service on the console server by running `puppet resource service pe-puppetdb ensure=running`

Configure excluded facts

Use the `facts_blacklist` parameter exclude facts from being stored in the PuppetDB database.

To specify which facts you want to exclude using Hiera:

1. Add the following to your default `.yaml` file and list the facts you want to exclude. For example, to exclude the facts `system_uptime_example` and `mountpoints_example`:

```
puppet_enterprise::puppetdb::database_ini::facts_blacklist:
- 'system_uptime_example'
- 'mountpoints_example'
```

2. To compile changes, run `puppet agent -t`

Configuring and tuning orchestration

After installing PE, you can change some default settings to further configure the orchestrator and pe-orchestration-services.

Related information

[Methods for configuring Puppet Enterprise](#) on page 189

After you've installed Puppet Enterprise (PE), optimize it for your environment by configuring and tuning settings as needed. For example, you might want to add your own certificate to the whitelist, increase the max-threads setting for http and https requests, or configure the number of JRuby instances.

[Tuning standard installations](#) on page 208

Use these guidelines to configure your installation to maximize its use of available system (CPU and RAM) resources.

[Configure ulimit for PE services](#) on page 207

The various services in PE require up to one file handle per connected client. On most operating system configurations, the defaults are not adequate for more than a couple hundred clients. To support more clients, you need to increase the number of allowed file handles.

Configure the orchestrator and pe-orchestration-services

There are several optional parameters you can add to configure the behavior of the orchestrator and pe-orchestration-services. Because they are profile classes, you can change these in the console in the PE Orchestrator group.

puppet_enterprise::profile::agent::pxp_enabled

Disable or enable the PXP service by setting it to `true` or `false`. If you disable this setting you can't use the orchestrator or the **Run Puppet** button in the console.

Default: `true`

puppet_enterprise::profile::bolt_server::concurrency

An integer that determines the maximum number of concurrent requests orchestrator can make to bolt-server.

Default: The current value stored for the Bolt server.



CAUTION: Do not set a concurrency limit that is higher than the bolt-server limit. This can cause timeouts that lead to failed task runs.

puppet_enterprise::profile::orchestrator::global_concurrent_compiles

An integer that determines how many concurrent compile requests can be outstanding to the master, across all orchestrator jobs.

Default: "8" (requests)

puppet_enterprise::profile::orchestrator::job_prune_threshold

Integer that represents the number of days before job reports are removed.

Default: "30" (days)

puppet_enterprise::profile::orchestrator::pcp_timeout

An integer that represents how many seconds must pass while an agent attempts to connect to a PCP broker. If the agent can't connect to the broker in that time frame, the run times out.

Default: "30" (seconds)

puppet_enterprise::profile::orchestrator::run_service

Disable or enable orchestration services. Set to true or false.

Default: true

puppet_enterprise::profile::orchestrator::task_concurrency

Integer representing the number of tasks that can run at the same time.

Default: "250" (tasks)

puppet_enterprise::profile::orchestrator::use_application_services

Enable or disable application management. Set to true or false.

Default: false

Related information

[Running Puppet on nodes](#) on page 320

Puppet automatically attempts to run on each of your nodes every 30 minutes. To trigger a Puppet run outside of the default 30-minute interval, you can manually run Puppet.

Configure the PXP agent

Puppet Execution Protocol (PXP) is a messaging system used to request tasks and communicate task statuses. The PXP agent runs the PXP service and you can configure it using Hiera.

puppet_enterprise::pxp_agent::ping_interval

Controls how frequently (in seconds) PXP agents will ping PCP brokers. If the brokers don't respond, the agents try to reconnect.

Default: 120 (seconds)

puppet_enterprise::pxp_agent::pxp_logfile

A string that represents the path to the PXP agent log file and can be used to debug issues with orchestrator.

Default:

- *nix: /var/log/puppetlabs/pxp-agent/pxp-agent.log
- Windows: C:\Program Data\PuppetLabs\pxp-agent\var\log\pxp-agent.log

puppet_enterprise::pxp_agent::spool_dir_purge_ttl

The amount of time to keep records of old Puppet or task runs on agents. You can declare time in minutes (30m), hours (2h), and days (14d).

Default: 14d

puppet_enterprise::pxp_agent::task_cache_dir_purge_ttl

Controls how long tasks are cached after use. You can declare time in minutes (30m), hours (2h), and days (14d).

Default: 14d

Correct ARP table overflow

In larger deployments that use the PCP broker, you might encounter ARP table overflows and need to adjust some system settings.

Overflows occur when the ARP table—a local cache of IP address to MAC address resolutions—fills and starts evicting old entries. When frequently used entries are evicted, network traffic will increase to restore them, increasing network latency and CPU load on the broker.

A typical log message looks like:

```
[root@sl peadmin]# tail -f /var/log/messages
Aug 10 22:42:36 sl kernel: Neighbour table overflow.
Aug 10 22:42:36 sl kernel: Neighbour table overflow.
Aug 10 22:42:36 sl kernel: Neighbour table overflow.
```

To work around this issue:

Increase sysctl settings related to ARP tables.

For example, the following settings are appropriate for networks hosting up to 2000 agents:

```
# Set max table size
net.ipv6.neigh.default.gc_thresh3=4096
net.ipv4.neigh.default.gc_thresh3=4096
# Start aggressively clearing the table at this threshold
net.ipv6.neigh.default.gc_thresh2=2048
net.ipv4.neigh.default.gc_thresh2=2048
# Don't clear any entries until this threshold
net.ipv6.neigh.default.gc_thresh1=1024
net.ipv4.neigh.default.gc_thresh1=1024
```

Configuring proxies

You can work around limited internet access by configuring proxies at various points in your infrastructure, depending on your connectivity limitations.

The examples provided here assume an unauthenticated proxy running at `proxy.example.vlan` on port 8080.

Related information

[Methods for configuring Puppet Enterprise](#) on page 189

After you've installed Puppet Enterprise (PE), optimize it for your environment by configuring and tuning settings as needed. For example, you might want to add your own certificate to the whitelist, increase the max-threads setting for http and https requests, or configure the number of JRuby instances.

Downloading agent installation packages through a proxy

If your master doesn't have internet access, it can't download agent installation packages. If you want to use package management to install agents, set up a proxy and specify its connection details so that `pe_repo` can access agent tarballs.

In the `pe_repo` class of the **PE Master** node group, specify values for `pe_repo::http_proxy_host` and `pe_repo::http_proxy_port` settings.

If you want to specify these settings in `pe.conf`, add the following to your `pe.conf` file with your desired parameters:

```
"pe_repo::http_proxy_host": "proxy.example.vlan",
"pe_repo::http_proxy_port": 8080
```

Tip: To test proxy connections to `pe_repo`, run:

```
curl -x http://proxy.example.vlan:8080 -I https://pm.puppetlabs.com
```

Setting a proxy for agent traffic

General proxy settings in `puppet.conf` manage HTTP connections that are directly initiated by the agent.

To configure agents to communicate through a proxy using `pe.conf`, specify values for the `http_proxy_host` and `http_proxy_port` settings in `/etc/puppetlabs/puppet/puppet.conf`.

```
http_proxy_host = proxy.example.vlan
http_proxy_port = 8080
```

For more information about HTTP proxy host options, see the Puppet [configuration reference](#).

Setting a proxy for Code Manager traffic

Code Manager has its own set of proxy configuration options which you can use to set a proxy for connections to the Git server or the Forge. These settings are unaffected by the proxy settings in `puppet.conf`, because Code Manager is run by Puppet Server.

Note: To set a proxy for Code Manager connections, you must use an HTTP URL for your `r10k` remote and for all Puppetfile module entries.

Use a proxy for all HTTP connections, including both Git and the Forge, when configuring Code Manager.

To configure Code Manager to use a proxy using Hiera, add the following code to your default `.yaml` and specify your proxy name. For example:

```
puppet_enterprise::profile::master::r10k_proxy: "http://
proxy.example.vlan:8080"
```

Tip: To test proxy connections to Git or the Forge, run one of these commands:

```
curl -x http://proxy.example.vlan:8080 -I https://github.com
```

```
curl -x http://proxy.example.vlan:8080 -I https://forgeapi.puppet.com
```

For detailed information about configuring proxies for Code Manager traffic, see the Code Manager documentation.

Related information

[Configuring proxies](#) on page 568

To configure proxy servers, use the proxy setting. You can set a global proxy for all HTTP(S) operations, for all Git or Forge operations, or for a specific Git repository only.

Configuring Java arguments

You might need to increase the Java Virtual Machine (JVM) memory allocated to Java services to improve performance in your Puppet Enterprise (PE) deployment.

Important: When you enable high availability, you must use `pe.conf` only — not the console — to specify configuration parameters. Using `pe.conf` ensures that configuration is applied to both your master and replica.

Related information

[Methods for configuring Puppet Enterprise](#) on page 189

After you've installed Puppet Enterprise (PE), optimize it for your environment by configuring and tuning settings as needed. For example, you might want to add your own certificate to the whitelist, increase the `max-threads` setting for `http` and `https` requests, or configure the number of JRuby instances.

[Tuning standard installations](#) on page 208

Use these guidelines to configure your installation to maximize its use of available system (CPU and RAM) resources.

Increase the Java heap size for PE Java services

The Java heap size is the Java Virtual Machine (JVM) memory allocated to Java services in Puppet Enterprise (PE). You can use any configuration method you choose. We will use the console to change the Java heap size for console services, Puppet Server, orchestration services, or PuppetDB in the examples below.

Note: Ensure that you have sufficient free memory before increasing the memory that is used by a service. The increases shown below are only examples.

1. In the console, click **Classification**. In the **PE Infrastructure** node group, select the appropriate node group.

Service	Node group	Class
pe-console-services	PE Console	puppet_enterprise::profile::console
Puppet Server	PE Master	puppet_enterprise::profile::master
pe-orchestration-services	PE Orchestrator	puppet_enterprise::profile::orchestration
PuppetDB	PE PuppetDB	puppet_enterprise::profile::puppetdb
ActiveMQ	PE ActiveMQ Broker	puppet_enterprise::profile::amq::broker

2. Click **Configuration** and scroll down to the appropriate class.
3. Click the **Parameter name** list and select `java_args`. Increase the heap size by replacing the parameter with the appropriate JSON string.

Service	Default heap size	New heap size	JSON string
pe-console-services	256 MB	512 MB	{ "Xmx" : "512m" , "Xms" : "512m" }
Puppet Server	2 GB	4 GB	{ "Xmx" : "4096m" , "Xms" : "4096m" }
orchestration-services	192 MB	1000 MB	{ "Xmx" : "1000m" , "Xms" : "1000m" }
PuppetDB	256 MB	512 MB	{ "Xmx" : "512m" , "Xms" : "512m" }
ActiveMQ	512 MB	1024 MB	{ "Xmx" : "1024m" , "Xms" : "1024m" }

4. Click **Add Parameter** and then commit changes.
5. Run Puppet on the appropriate nodes to apply the change. If you're running it on the console node, the console will be unavailable briefly while `pe-console-services` restarts.

Service	Node
pe-console-services	console
Puppet Server	master and compile masters
pe-orchestration-services	master
PuppetDB	PuppetDB

Related information

[Running Puppet on nodes](#) on page 320

Puppet automatically attempts to run on each of your nodes every 30 minutes. To trigger a Puppet run outside of the default 30-minute interval, you can manually run Puppet.

Disable Java garbage collection logging

Java garbage collection logs can be useful when diagnosing performance issues with JVM-based PE services. Garbage collection logs are enabled by default, and the results are captured in the support script, but you can disable them.

Use Hiera or `pe.conf` to disable garbage collection logging by adding or changing the following parameters, as needed:

```
puppet_enterprise::console_services::enable_gc_logging: false
puppet_enterprise::master::puppetserver::enable_gc_logging: false
puppet_enterprise::profile::orchestrator::enable_gc_logging: false
puppet_enterprise::puppetdb::enable_gc_logging: false
puppet_enterprise::profile::amq::broker::enable_gc_logging: false
```

Configuring ulimit for PE services

As your infrastructure grows and you bring more agents under management, you might need to increase the number allowed file handles per client.

Related information

[Methods for configuring Puppet Enterprise](#) on page 189

After you've installed Puppet Enterprise (PE), optimize it for your environment by configuring and tuning settings as needed. For example, you might want to add your own certificate to the whitelist, increase the max-threads setting for http and https requests, or configure the number of JRuby instances.

Configure ulimit for PE services

The various services in PE require up to one file handle per connected client. On most operating system configurations, the defaults are not adequate for more than a couple hundred clients. To support more clients, you need to increase the number of allowed file handles.

You can increase the limits for the following services:

- `pe-orchestration-services`
- `pe-puppetdb`
- `pe-console-services`
- `pe-puppetserver`
- `pe-puppet`

The location and method for configuring ulimit depends on your agent's platform. You might use `systemd`, `upstart`, or some other init system.

In the following instructions, replace `<PE SERVICE>` with the specific service you're editing. The examples show setting a limit of 32768, which you can also change according to what you need.

Configure ulimit using systemd

With `systemd`, the number of open file handles allowed is controlled by a setting in the service file at `/usr/lib/systemd/system/<PE SERVICE>.service`.

1. To increase the limit, run the following commands, setting the `LimitNOFILE` value to the new number:

```
mkdir /etc/systemd/system/<PE SERVICE>.service.d
echo "[Service]
LimitNOFILE=32768" > /etc/systemd/system/<PE SERVICE>.service.d/
limits.conf
systemctl daemon-reload
```

2. Confirm the change by running: `systemctl show <PE SERVICE> | grep LimitNOFILE`

Configure ulimit using upstart

For Ubuntu and Red Hat systems, the number of open file handles allowed for is controlled by settings in service files.

The service files are:

- Ubuntu: `/etc/default/<PE SERVICE>`
- Red Hat: `/etc/sysconfig/<PE SERVICE>`

For both Ubuntu and Red Hat, set the last line of the file as follows:

```
ulimit -n 32678
```

This sets the number of open files allowed at 32,678.

Configure ulimit on other init systems

The ulimit controls the number of processes and file handles that the PE service user can open and process.

To increase the ulimit for a PE service user:

Edit `/etc/security/limits.conf` so that it contains the following lines:

```
<PE SERVICE USER> soft nofile 32768
<PE SERVICE USER> hard nofile 32768
```

Tuning standard installations

Use these guidelines to configure your installation to maximize its use of available system (CPU and RAM) resources.

PE is composed of multiple services on one or more infrastructure hosts. Each service has multiple settings that can be configured to maximize use of system resources and optimize performance. The default settings for each service are conservative, because the set of services sharing resources on each host varies depending on your infrastructure.

Optimized settings vary depending on the complexity and scale of your infrastructure. For example, you might need to allocate more memory per JRuby depending on the number of environments, or the number of agents and their run intervals.

Configure settings after an install or upgrade, or after making changes to infrastructure hosts, including changing the system resources of existing hosts, or adding new hosts, including compilers.

Related information

[Hardware requirements](#) on page 114

These hardware requirements are based on internal testing at Puppet and are meant only as guidelines to help you determine your hardware needs.

[Methods for configuring Puppet Enterprise](#) on page 189

After you've installed Puppet Enterprise (PE), optimize it for your environment by configuring and tuning settings as needed. For example, you might want to add your own certificate to the whitelist, increase the max-threads setting for http and https requests, or configure the number of JRuby instances.

[Tune the maximum number of JRuby instances](#) on page 191

The `jruby_max_active_instances` setting controls the maximum number of JRuby instances to allow on the Puppet Server.

[Configure command processing threads](#) on page 201

The `command_processing_threads` setting defines how many command processing threads PuppetDB uses to sort incoming data. Each thread can process a single command at a time. The setting defaults to half the number of cores in your system.

[Increase the Java heap size for PE Java services](#) on page 206

The Java heap size is the Java Virtual Machine (JVM) memory allocated to Java services in Puppet Enterprise (PE). You can use any configuration method you choose. We will use the console to change the Java heap size for console services, Puppet Server, orchestration services, or PuppetDB in the examples below.

[Tuning the PostgreSQL buffer pool size](#) on page 197

If you are experiencing performance issues or instability with the console, adjust the buffer memory settings for PostgreSQL.

Master tuning

These are the default and recommended tuning settings for your master or high availability replica.

Tuning for 4 cores, 8 GB of RAM

Install type	Puppet Server			PuppetDB		Console	Orchestrator		PostgreSQL		CPU totals		Memory totals	
	JRuby max active instances	Java heap (MB)	Reserved code cache (MB)	Command processing threads (MB)	Java heap (MB)		Java heap (MB)	Java heap (MB)	Shared buffers (MB)	Work memory (MB)	Used	Free	Used (MB)	Free (MB)
Default	3	2048	512	2	256		256	192	2048	4	5	-1	5312	2880
Recommended	2	1024	512	1	512		512	512	2048	4	3	1	5120	3072
With compilers	2	1024	512	2	1024		512	512	2048	4	4	0	5632	2560

Tuning for 8 cores, 16 GB of RAM

Install type	Puppet Server			PuppetDB		Console	Orchestrator		PostgreSQL		CPU totals		Memory totals	
	JRuby max active instances	Java heap (MB)	Reserved code cache (MB)	Command processing threads (MB)	Java heap (MB)		Java heap (MB)	Java heap (MB)	Shared buffers (MB)	Work memory (MB)	Used	Free	Used (MB)	Free (MB)
Default	4	2048	1024	4	256		256	192	4096	4	8	0	7872	8512
Recommended	2	3840	1024	2	1024		768	768	4096	4	7	1	11520	4864
With compilers	2	1536	1024	4	3072		768	768	4096	4	6	2	11264	5120

Tuning 16 cores, 32 GB of RAM

Install type	Puppet Server			PuppetDB		Console	Orchestrator		PostgreSQL		CPU totals		Memory totals	
	JRuby max active instances	Java heap (MB)	Reserved code cache (MB)	Command processing threads (MB)	Java heap (MB)		Java heap (MB)	Java heap (MB)	Shared buffers (MB)	Work memory (MB)	Used	Free	Used (MB)	Free (MB)
Default	4	2048	2048	8	256		256	192	4096	4	12	4	8896	23872
Recommended	4	11264	2048	4	2048		1024	1024	8192	4	15	1	25600	7168
With compilers	4	4096	2048	8	5120		1024	1024	8192	4	12	4	21504	11264

Compiler tuning

These are the default and recommended tuning settings for compilers.

Tuning 4 cores, 8 GB of RAM

Install type	Puppet Server			CPU totals		Memory totals	
	JRuby max active instances	Java heap (MB)	Reserved code cache (MB)	Used	Free	Used (MB)	Free (MB)
Default	3	2048	512	3	1	2560	5632
Recommended	3	1536	512	3	1	2048	6144

Tuning 8 cores, 16 GB of RAM

Install type	Puppet Server			CPU totals		Memory totals	
	JRuby max active instances	Java heap (MB)	Reserved code cache (MB)	Used	Free	Used (MB)	Free (MB)
Default	4	2048	1024	4	4	3072	13312
Recommended	7	5376	1024	7	1	6400	9984

Tuning 16 cores, 32 GB of RAM

Install type	Puppet Server			CPU totals		Memory totals	
	JRuby max active instances	Java heap (MB)	Reserved code cache (MB)	Used	Free	Used (MB)	Free (MB)
Default	4	2048	2048	4	12	4096	28672
Recommended	15	15360	2048	15	1	17408	15360

Using the `puppet infrastructure tune` command

The `puppet infrastructure tune` command outputs optimized settings for PE services based on recommended guidelines.

When you run `puppet infrastructure tune` on your master, it queries PuppetDB to identify infrastructure hosts and their processor and memory facts, and outputs settings in YAML format for use in Hiera.

The `puppet infrastructure tune` command optimizes based on available system resources, not agent load or environment complexity. You can add the option `--memory_per_jruby <MB>` to optimize the Puppet Server service for environment complexity.

With the `--current` option, you can review currently specified settings for PE services. Settings might be specified in either the console or in Hiera, with console settings taking precedence over Hiera settings. You should specify settings in the console or Hiera, but not both. The `--current` option identifies duplicate settings found in both places.

The `puppet infrastructure tune` command is compatible with infrastructures with or without compilers, external PostgreSQL hosts, and replica hosts. You can run the command on your master, but not on compilers or high availability replicas. The command must be run as root.

For more information about the `tune` command, run `puppet infrastructure tune --help`.

Writing configuration files

Puppet supports two formats for configuration files that configure settings: valid JSON and Human-Optimized Config Object Notation (HOCON), a JSON superset.

For more information about HOCON itself, see the [HOCON documentation](#).

Configuration file syntax

Refer to these examples when you're writing configuration files to identify correct JSON or HOCON syntax.

Brackets

In HOCON, you can omit the brackets ({ }) around a root object.

JSON example	HOCON example
<pre>{ "authorization": { "version": 1 } }</pre>	<pre>"authorization": { "version": 1 }</pre>

Quotes

In HOCON, double quotes around key and value strings are optional in most cases. However, double quotes are required if the string contains the characters *, ^, +, :, or =.

JSON example	HOCON example
<pre>"authorization": { "version": 1 }</pre>	<pre>authorization: { version: 1 }</pre>

Commas

When writing a map or array in HOCON, you can use a new line instead of a comma.

	JSON example	HOCON example
Map	<pre>rbac: { password-reset- expiration: 24, session-timeout: 60, failed-attempts- lockout: 10, }</pre>	<pre>rbac: { password-reset- expiration: 24 session-timeout: 60 failed-attempts- lockout: 10 }</pre>
Array	<pre>http-client: { ssl-protocols: [TLSv1, TLSv1.1, TLSv1.2] }</pre>	<pre>http-client: { ssl-protocols: [TLSv1 TLSv1.1 TLSv1.2] }</pre>

Comments

Add comments using either `//` or `#`. Inline comments are supported.

HOCON example

```
authorization: {
  version: 1
  rules: [
    {
      # Allow nodes to retrieve their own catalog
      match-request: {
        path: "^/puppet/v3/catalog/([^/]+)$"
        type: regex
        method: [get, post]
      }
    }
  ]
}
```

Analytics data collection

Some components automatically collect data about how you use Puppet Enterprise. If you want to opt out of providing this data, you can do so, either during or after installing.

Related information

[Methods for configuring Puppet Enterprise](#) on page 189

After you've installed Puppet Enterprise (PE), optimize it for your environment by configuring and tuning settings as needed. For example, you might want to add your own certificate to the whitelist, increase the max-threads setting for http and https requests, or configure the number of JRuby instances.

What data does Puppet Enterprise collect?

Puppet Enterprise (PE) collects the following data when Puppet Server starts or restarts, and again every 24 hours.

License, version, master, and agent information:

- License UUID
- Number of licensed nodes
- Product name
- PE version
- Master's operating system
- Master's public IP address
- Whether the master is running on Microsoft Azure
- The hypervisor the master is running on, if applicable
- Number of nodes in deployment
- Agent operating systems
- Number of agents running each operating system
- Agent versions
- Number of agents running each version of Puppet agent
- All-in-One (AIO) puppet-agent package versions
- Number of agents running on Microsoft Azure or Google Cloud Platform, if applicable
- Number of configured high availability replicas, if applicable

Puppet Enterprise feature use information:

- Number of node groups in use
- Number of nodes used in orchestrator jobs after last orchestrator restart

- Mean nodes per orchestrator job
- Maximum nodes per orchestrator job
- Minimum nodes per orchestrator job
- Total orchestrator jobs created after last orchestrator restart
- Number of non-default user roles in use
- Type of certificate autosigning in use
- Number of nodes in the job that were run over Puppet Communications Protocol
- Number of nodes in the job that were run over SSH.
- Number of nodes in the job that were run over WinRM.
- List of Puppet task jobs
- List of Puppet deploy jobs
- List of Puppet task jobs run by plans
- List of file upload jobs run by plans
- List of script jobs run by plans
- List of command jobs run by plans
- List of wait jobs run by plans
- How nodes were selected for the job
- Whether the job was started by the PE admin account
- Number of nodes in the job
- Length of description applied to the job
- Length of time the job ran
- User agent used to start the job (to distinguish between the console, command line, and API)
- UUID used to correlate multiple jobs run by the same user
- Time the task job was run
- How nodes were selected for the job
- Whether the job was started by the PE admin account
- Number of nodes in the job
- Length of description applied to the job
- Whether the job was asked to override agent-configured no-operation (no-op) mode
- Whether app-management was enabled in the orchestrator for this job
- Time the deploy job was run
- Type of version control system webhook
- Whether the request was to deploy all environments
- Whether code-manager will wait for all deploys to finish or error to return a response
- Whether the deploy is a dry-run
- List of environments requested to deploy
- List of deploy requests
- Total time elapsed for all deploys to finish or error
- List of total wait times for deploys specifying `--wait` option
- Name of environment deployed
- Time needed for r10k to run
- Time spent committing to file sync
- Time elapsed for all environment hooks to run
- List of individual environment deploys
- Puppet classes applied from publicly available modules, with node counts per class

Backup and Restore information:

- Whether user used `--force` option when running restore
- Scope of restore
- Time in seconds for various restore functions

- Time to check for disk space to restore
- Time to stop PE related services
- Time to restore PE file system components
- Time to migrate PE configuration for new server
- Time to configure PE on newly restored master
- Time to update PE classification for new server
- Time to deactivate the old master node
- Time to restore the pe-orchestrator database
- Time to restore the pe-rbac database
- Time to restore the pe-classifier database
- Time to restore the pe-activity database
- Time to restore the pe-puppetdb database
- Total time to restore
- List of puppet backup restore jobs
- Whether user used `--force` option when running `puppet-backup create`
- Whether user used `--dir` option when running `puppet-backup create`
- Whether user used `--name` option when running `puppet-backup create`
- Scope of backup
- Time in seconds for various back up functions
- Time needed to estimate backup size, disk space needed, and disk space available
- Time to create file system backup
- Time to back up the pe-orchestrator database
- Time to back up the pe-rbac database
- Time to back up the pe-classifier database
- Time to back up the pe-activity database
- Time to back up the pe-puppetdb database
- Time to compress archive file to backup directory
- Time to back up PE related classification
- Total time to back up
- List of puppet-backup create jobs

Puppet Server performance information:

- Total number of JRuby instances
- Maximum number of active JRuby instances
- Maximum number of requests per JRuby instance
- Average number of instances not in use over the process's lifetime
- Average wait time to lock the JRuby pool
- Average time the JRuby pool held a lock
- Average time an instance spent handling requests
- Average time spent waiting to reserve an instance from the JRuby pool
- Number of requests that timed out while waiting for a JRuby instance
- Amount of memory the JVM starts with
- Maximum amount of memory the JVM is allowed to request from the operating system

Installer information:

- Installation method (express, text, web, repair)
- Current version, if upgrading
- Target version
- Success or failure, and limited information on the type of failure, if an

If PE is installed using an Amazon Web Services Marketplace Image:

- The marketplace name
- Marketplace image billing mode (bring your own license or pay as you go)

While in use, the console collects the following information:

- Pageviews
- Link and button clicks
- Page load time
- User language
- Screen resolution
- Viewport size
- Anonymized IP address

The console *does not* collect user inputs such as node or group names, user names, rules, parameters, or variables

The collected data is tied to a unique, anonymized identifier for each master and your site as a whole. No personally identifiable information is collected, and the data we collect is never used or shared outside Puppet, Inc.

How does sharing this data benefit you?

We use the data to identify organizations that could be affected by a security issue, alert them to the issue, and provide them with steps to take or fixes to download. In addition, the data helps us understand how people use the product, which helps us improve the product to meet your needs.

How does Puppet use the collected data?

The data we collect is one of many methods we use for learning about our customers. For example, knowing how many nodes you manage helps us develop more realistic product testing. And learning which operating systems are the most and the least used helps us decide where to prioritize new functionality. By collecting data, we begin to understand you as a customer.

Opt out during the installation process

To opt out of data collection during installation, you can set the `DISABLE_ANALYTICS` environment variable when you run the installer script. This opt-out method works for all installation methods: express, text, and web-based.

Setting the `DISABLE_ANALYTICS` environment variable during installation sets `puppet_enterprise::send_analytics_data: false` in `pe.conf`, opting you out of data collection.

Follow the instructions for your chosen installation method, adding `DISABLE_ANALYTICS=1` when you call the installer script, for example:

```
sudo DISABLE_ANALYTICS=1 ./puppet-enterprise-installer
```

Opt out after installing

If you've already installed PE and want to disable data collection, follow these steps.

1. In the console, click **Classification**, and then click **PE Infrastructure**.
2. On the **Configuration** tab, on the `puppet_enterprise::profile::master` class, add `send_analytics_data` as a parameter and set the **Value** to `false`.
3. On the **Inventory** page, select your master node and click **Run Puppet**.
4. Select your console node and click **Run Puppet**.

After Puppet runs to enforce the changes on the master and console nodes, you have opted out of data collection.

Static catalogs in Puppet Enterprise

A catalog is a document that describes the desired state for each resource that Puppet manages on a node. A master typically compiles a catalog from manifests of Puppet code. A static catalog is a specific type of Puppet catalog that includes metadata that specifies the desired state of any file resources containing `source` attributes pointing to `puppet:///` locations on a node.

The metadata in a static catalog can refer to a specific version of the file (not just the latest version), and can confirm that the agent is applying the appropriate version of the file resource for the catalog. Because the metadata is provided in the catalog, agents make fewer requests to the master.

See the open source Puppet documentation more information about [Resources](#), [File types](#), and [Catalog compilation](#).

Related information

[Methods for configuring Puppet Enterprise](#) on page 189

After you've installed Puppet Enterprise (PE), optimize it for your environment by configuring and tuning settings as needed. For example, you might want to add your own certificate to the whitelist, increase the `max-threads` setting for `http` and `https` requests, or configure the number of JRuby instances.

Enabling static catalogs

When a master produces a non-static catalog, the catalog doesn't specify the version of file resources. When the agent applies the catalog, it always retrieves the latest version of that file resource, or uses a previously retrieved version if it matches the latest version's contents.

This potential problem affects file resources that use the `source` attribute. File resources that use the `content` attribute are not affected, and their behavior does not change in static catalogs.

When a manifest depends on a file whose contents change more frequently than the agent receives new catalogs, a node might apply a version of the referenced file that doesn't match the instructions in the catalog. In Puppet Enterprise (PE), such situations are particularly likely if you've configured your agents to run off cached catalogs for participation in application orchestration services.

Consequently, the agent's Puppet runs might produce different results each time the agent applies the same catalog. This often causes problems because Puppet generally expects a catalog to produce the same results each time it's applied, regardless of any code or file content updates on the master.

Additionally, each time an agent applies a normal cached catalog that contains file resources sourced from `puppet:///` locations, the agent requests file metadata from the master each time the catalog's applied, even though nothing's changed in the cached catalog. This causes the master to perform unnecessary resource-intensive checksum calculations for each file resource.

Static catalogs avoid these problems by including metadata that refers to a specific version of the resource's file. This prevents a newer version from being incorrectly applied, and avoids having the agent regenerate the metadata on each Puppet run. The metadata is delivered in the form of a unique hash maintained, by default, by the file sync service.

We call this type of catalog "static" because it contains all of the information that an agent needs to determine whether the node's configuration matches the instructions and state of file resources at the static point in time when the catalog was generated.

Differences in catalog behavior

Without static catalogs enabled:

- The agent sends facts to the master and requests a catalog.
- The master compiles and returns the agent's catalog.
- The agent applies the catalog by checking each resource the catalog describes. If it finds any resources that are not in the desired state, it makes the necessary changes.

With static catalogs enabled:

- The agent sends facts to the master and requests a catalog.

- The master compiles and returns the agent's catalog, including metadata that specifies the desired state of the node's file resources.
- The agent applies the catalog by checking each resource the catalog describes. If the agent finds any resources that are not in the desired state, it makes the necessary changes based on the state of the file resources at the static point in time when the catalog was generated.
- If you change code on the master, file contents are not updated until the agent requests a new catalog with new file metadata.

Enabling file sync

In PE, static catalogs are disabled across all environments for new installations. To use static catalogs in PE, you must enable file sync. After file sync is enabled, Puppet Server automatically creates static catalogs containing file metadata for eligible resources, and agents running Puppet 1.4.0 or newer can take advantage of the catalogs' new features.

If you do not enable file sync and Code Manager, you can still use static catalogs, but you need to create some custom scripts and set a few parameters in the console.

Enforcing change with static catalogs

When you are ready to deploy new Puppet code and deliver new static catalogs, you don't need to wait for agents to check in. Use the Puppet orchestrator to enforce change and deliver new catalogs across your PE infrastructure, on a per-environment basis.

When aren't static catalogs applied?

In the following scenarios, either agents won't apply static catalogs, or catalogs won't include metadata for file resources.

- Static catalogs are globally disabled.
- Code Manager and file sync are disabled, and `code_id` and `code_content` aren't configured.
- Your agents aren't running PE 2016.1 or later (Puppet agent version 1.4.0 or later).

Additionally, Puppet won't include metadata for a file resource if it:

- Uses the `content` attribute instead of the `source` attribute.
- Uses the `source` attribute with a non-Puppet scheme (for example `source => 'http://host:port/path/to/file'`).
- Uses the `source` attribute without the built-in modules mount point.
- Uses the `source` attribute, but the file on the master is not in `/etc/puppetlabs/code/environments/<environment>/**/*.files/**`. For example, module files are typically in `/etc/puppetlabs/code/environments/<environment>/modules/<module_name>/files/**`.

Agents continue to process the catalogs in these scenarios, but without the benefits of inlined file metadata or file resource versions.

Related information

[Enabling or disabling file sync](#) on page 607

File sync is normally enabled or disabled automatically along with Code Manager.

[Running jobs with Puppet orchestrator](#) on page 419

With the Puppet orchestrator, you can run two types of "jobs": on-demand Puppet runs or Puppet tasks.

Disabling static catalogs globally with Hiera

You can turn off all use of static catalogs with a Hiera setting.

To disable static catalogs using Hiera:

1. Add the following code to your default `.yaml` file and set the parameter to `false`:

```
puppet_enterprise::master::static_catalogs: false
```

2. To compile changes, run `puppet agent -t`

Using static catalogs without file sync

To use static catalogs without enabling file sync, you must set the `code_id` and `code_content` parameters in Puppet, and then configure the `code_id_command`, `code_content_command`, and `file_sync_enabled` parameters in the console.

1. Set `code_id` and `code_content` by following the instructions in the [open source Puppet static catalogs documentation](#). Then return to this page to set the remaining parameters.
2. In the console, click **Classification**, and in the **PE Infrastructure** node group, select the **PE Master** node group.
3. On the **Configuration** tab, locate the `puppet_enterprise::profile::master` class, and select **file_sync_enabled** from its **Parameter** list.
4. In the **Value** field, enter `false`, and click **Add parameter**.
5. Select the **code_id_command** parameter, and in the **Value** field, enter the absolute path to the `code_id` script, and click **Add parameter**.
6. Select the **code_content_command** parameter, and in the **Value** field, add the absolute to the `code_content` script, and click **Add parameter**.
7. Commit changes.
8. Run Puppet on the master.

Related information

[Running Puppet on nodes](#) on page 320

Puppet automatically attempts to run on each of your nodes every 30 minutes. To trigger a Puppet run outside of the default 30-minute interval, you can manually run Puppet.

Configuring high availability

Enabling high availability for Puppet Enterprise ensures that your system remains operational even if certain infrastructure components become unreachable.

High availability

High availability configuration creates a replica of your master.

You can have only one provisioned replica at a time.

Tip: You can add high availability to an installation with or without compilers. High availability isn't supported with standalone PE-PostgreSQL or FIPS-compliant installations.

There are two main advantages to enabling high availability:

- If your master fails, the replica takes over, continuing to perform critical operations.
- If your master can't be repaired, you can promote the replica to master. Promotion establishes the replica as the new, permanent master.

Related information

[What happens during failovers](#) on page 219

Failover occurs when the replica takes over services usually performed by the master.

High availability architecture

The replica is not an exact copy of the master. Rather, the replica duplicates specific infrastructure components and services. Hieradata and other custom configurations are not replicated.

Replication can be *read-write*, meaning that data can be written to the service or component on either the master or the replica, and the data is synced to both nodes. Alternatively, replication can be *read-only*, where data is written only to the master and synced to the replica. Some components and services, like Puppet Server and the console service UI, are not replicated because they contain no native data.

Some components and services are activated immediately when you enable a replica; others aren't active until you promote a replica. After you provision and enable a replica, it serves as a compiler, redirecting PuppetDB and cert requests to the master.

Component or service	Type of replication	Activated when replica is...
Puppet Server	none	enabled
File sync client	read-only	enabled
PuppetDB	read-write	enabled
Certificate authority	read-only	promoted
RBAC service	read-only	enabled
Node classifier service	read-only	enabled
Activity service	read-only	enabled
Orchestration service	read-only	promoted
Console service UI	none	promoted

Important: When you enable high availability, you must use `orpe.conf` only — not the console — to specify configuration parameters. Using `pe.conf` or `ensures` that configuration is applied to both your master and replica.

In a standard installation, when a Puppet run fails over, agents communicate with the replica instead of the master. In a large or extra-large installation with compilers, agents communicate with load balancers or compilers, which communicate with the master or replica.

Related information

[Configure settings with Hiera](#) on page 190

Hiera is a hierarchy-based method of configuration management that relies on a “defaults, with overrides” system. When you add a parameter or setting to your Hiera data, Hiera searches through the data in the order it was written to find the value you want to change. Once found, it overrides the default value with the new parameter or setting. You can use Hiera to manage your PE configuration settings.

What happens during failovers

Failover occurs when the replica takes over services usually performed by the master.

Failover is automatic — you don't have to take action to activate the replica. With high availability enabled, Puppet runs are directed first to the master. If the master is either fully or partially unreachable, runs are directed to the replica.

In partial failovers, Puppet runs can use the server, node classifier, or PuppetDB on the replica if those services aren't reachable on the master. For example, if the master's node classifier fails, but its Puppet Server is still running, agent runs use the Puppet Server on the master but fail over to the replica's node classifier.

What works during failovers:

- Scheduled Puppet runs
- Catalog compilation

- Viewing classification data using the node classifier API
- Reporting and queries based on PuppetDB data

What doesn't work during failovers:

- Deploying new Puppet code
- Editing node classifier data
- Using the console
- Certificate functionality, including provisioning new agents, revoking certificates, or running the `puppet certificate` command
- Most CLI tools
- Application orchestration

System and software requirements

Your Puppet infrastructure must meet specific requirements in order to configure high availability.

Component	Requirement
Operating system	All supported PE master platforms.
Software	<ul style="list-style-type: none"> • You must use Code Manager so that code is deployed to both the master and the replica after you enable a replica. • You must use the default PE node classifier so that high availability classification can be applied to nodes. • Orchestrator must be enabled so that agents are updated when you provision or enable a replica. Orchestrator is enabled by default.
Replica	<ul style="list-style-type: none"> • Must be an agent node that doesn't have a specific function already. You can decommission a node, uninstall all puppet packages, and re-commission the node to be a replica. However, a compiler cannot perform two functions, for example, as a compiler and a replica. • Must have the same hardware specifications and capabilities as your master node.
Firewall	<p>Both the master and the replica must comply with these port requirements:</p> <ul style="list-style-type: none"> • Firewall configuration for installations with compilers. These requirements apply whether your HA environment uses a single master or compilers. • Port 5432 must be open to facilitate database replication by console services.
Node names	<ul style="list-style-type: none"> • You must use resolvable domain names when specifying node names for the master and replica.

Component	Requirement
RBAC tokens	<ul style="list-style-type: none"> You must have an admin RBAC token when running <code>puppet infrastructure</code> commands, including <code>provision</code>, <code>enable</code>, and <code>forget</code>. You can generate a token using the <code>puppet-access</code> command. <p>Note: You don't need an RBAC token to promote a replica.</p>

Related information

[Managing and deploying Puppet code](#) on page 550

Puppet Enterprise (PE) includes built-in tools for managing and deploying your Puppet infrastructure. Code Manager and r10k are code management tools that automatically install modules, create and maintain environments, and deploy new code to your masters, all based on version control of your Puppet code and data.

[Configure the orchestrator and pe-orchestration-services](#) on page 202

There are several optional parameters you can add to configure the behavior of the orchestrator and pe-orchestration-services. Because they are profile classes, you can change these in the console in the PE Orchestrator group.

[Firewall configuration for large installations with compilers](#) on page 126

These are the port requirements for large installations with compilers.

[Generate a token using puppet-access](#) on page 243

Use `puppet-access` to generate an authentication tokens from the command line of any workstation (Puppet-managed or not), without the need to SSH into the Puppet master.

Classification changes in high availability installations

When you provision and enable a replica, the system makes a number of classification changes in order to manage high availability without affecting existing configuration.

These preconfigured node groups are added in high availability installations:

Node group	Matching nodes	Inherits from
PE HA Master	master	PE Master node group
PE HA Replica	master replicas	PE Infrastructure node group

These parameters are used to configure high availability installations:

Parameter	Purpose	Node group	Classes	Notes
<code>agent-server-urls</code>	Specifies the list of servers that agents contact, in order.	PE Agent PE Infrastructure Agent	<code>puppet_enterprise::profile::agent</code>	In large installations with compilers, agents must be configured to communicate with the load balancers or compilers. Important: Setting agents to communicate directly with the replica in order to use the replica as a compiler is not a supported configuration.
<code>replication_mode</code>	Sets replication type and direction on masters and replicas.	PE Master (none) HA Master (source) HA Replica (replica)	<code>puppet_enterprise::profile::master</code> <code>puppet_enterprise::profile::database</code> <code>puppet_enterprise::profile::console</code>	System parameter. Don't modify.
<code>ha_enabled_replicas</code>	Tracks replica nodes that are failover ready.	PE Infrastructure Agent	<code>puppet_enterprise</code>	System parameter. Don't modify. Updated when you enable a replica.
<code>pcp_broker_list</code>	Specifies the list of Puppet Communications Protocol brokers that Puppet Execution Protocol agents contact, in order.	PE Agent PE Infrastructure Agent	<code>puppet_enterprise::profile::agent</code>	

Load balancer timeout in high availability installations

High availability configuration uses timeouts to determine when to fail over to the replica. If the load balancer timeout is shorter than the server and agent timeout, connections from agents might be terminated during failover.

To avoid timeouts, set the timeout option for load balancers to four minutes or longer. This duration allows compilers enough time for required queries to PuppetDB and the node classifier service. You can set the load balancer timeout option using parameters in the haproxy or f5 modules.

Configure high availability

To configure high availability, you must provision and enable a replica to serve as backup during failovers. If your master is permanently disabled, you can then promote a replica.

Before you begin

Apply [high availability system and software requirements](#).

Tip: High availability is configured and managed with `puppet infrastructure` commands, many of which require a valid admin RBAC token. For details about these commands, on the command line, run `puppet infrastructure help <ACTION>`, for example, `puppet infrastructure help provision`.

Related information

[Generate a token using puppet-access](#) on page 243

Use `puppet-access` to generate an authentication tokens from the command line of any workstation (Puppet-managed or not), without the need to SSH into the Puppet master.

Provision a replica

Provisioning a replica duplicates specific components and services from the master to the replica.

Before you begin

Ensure you have a valid admin RBAC token.

Note: While completing this task, the master is unavailable to serve catalog requests. Time completing this task accordingly.

1. Ensure that the node you're provisioning as a replica is set to use the master as its Puppet Server.

On the prospective replica node, in the `/etc/puppetlabs/puppet/puppet.conf` file's main section, set the `server` variable to the node name of the master. For example:

```
[main]
certname = <REPLICA NODE NAME>
server = <MASTER NODE NAME>
```

2. On the master, as the root user, run `puppet infrastructure provision replica <REPLICA NODE NAME>`

After the provision command completes, services begin syncing from the master to the replica. The amount of time the sync takes depends on the size of your PuppetDB and the capability of your hardware. Typical installations take 10-30 minutes. With large data sets, you can optionally do a manual PuppetDB replication to speed installation.

Note: All `puppet infrastructure` commands must be run from a root session. Running with elevated privileges via `sudo puppet infrastructure` is not sufficient. Instead, start a root session by running `sudo su -`, and then run the `puppet infrastructure` command.

3. (Optional) Verify that all services running on the master are also running on the replica:
 - a) From the master, run `puppet infrastructure status --verbose` to verify that the replica is available.
 - b) From any managed node, run `puppet agent -t --noop --server_list=<REPLICA HOSTNAME>`. If the replica is correctly configured, the Puppet run succeeds and shows no changed resources.

When provisioning is complete, you must enable the replica to complete your HA configuration.

Manually copy PuppetDB to speed replication

For large PuppetDB installations, you can speed initial replication by manually copying the database from the master to the replica. If you have already started automatic provisioning, you can manually copy your PuppetDB at any time during sync.

The size of your PuppetDB correlates with the number of nodes and resources in your Puppet catalogs. To optionally examine the size of your database, on the PuppetDB PostgreSQL node, run `sudo -u pe-postgres /opt/puppetlabs/server/bin/psql -c '\l+ "<DB_NAME>" '.`

Note: By default, `<DB_NAME>` is `pe-puppetdb`.

1. On the PuppetDB node, export the database:

```
sudo -u pe-postgres /opt/puppetlabs/server/bin/pg_dump --format=custom --
compress=3 --file=<DUMP_OUTPUT> --dbname="<DB_NAME>"
```

2. On the PuppetDB node, transfer the output using your preferred tool, such as SCP:

```
scp -r <DUMP_OUTPUT> <REMOTE_USER>@<REPLICA_HOST>:<REPLICA_DUMP_OUTPUT>
```

3. On the primary replica node, restore PuppetDB:

```
sudo puppet resource service puppet ensure=stopped
sudo puppet resource service pe-puppetdb ensure=stopped
sudo -u pe-postgres /opt/puppetlabs/server/bin/pg_restore --clean --
jobs=<PROCESSOR_COUNT> --dbname="<DB_NAME>" <REPLICA_DUMP_OUTPUT>
sudo puppet resource service puppet ensure=running
sudo puppet resource service pe-puppetdb ensure=running
sudo puppet agent -t
```

After manual export and restore, PuppetDB automatically updates the replica with any changes that occurred on the master in the meantime.

Enable a replica

Enabling a replica activates most of its duplicated services and components, and instructs agents and infrastructure nodes how to communicate in a failover scenario.

Before you begin

- Back up your classifier hierarchy, because enabling a replica alters classification.
- Ensure you have a valid admin RBAC token.

Note: While completing this task, the master is unavailable to serve catalog requests. Time completing this task accordingly.

1. On the master, as the root user, run `puppet infrastructure enable replica <REPLICA NODE NAME>`, then follow the prompts to instruct Puppet how to configure your deployment.
2. Deploy updated configuration to nodes by running Puppet or waiting for the next scheduled Puppet run.

Note: If you use the direct workflow, where agents use cached catalogs, you must manually deploy the new configuration by running `puppet job run --no-enforce-environment --query 'nodes {deactivated is null and expired is null}'`

3. Optional: Perform any tests you feel are necessary to verify that Puppet runs continue to work during failover. For example, to simulate an outage on the master:
 - a) Prevent the replica and a test node from contacting the master. For example, you might temporarily shut down the master or use `iptables` with drop mode.
 - b) Run `puppet agent -t` on the test node. If the replica is correctly configured, the Puppet run succeeds and shows no changed resources. Runs might take longer than normal when in failover mode.
 - c) Reconnect the replica and test node.
4. If you've specified any tuning parameters for your master using the console, move them to Hiera instead.

Using Hiera ensures configuration is applied to both your master and replica.

Related information

[Back up your infrastructure](#) on page 698

PE backup creates a copy of your Puppet infrastructure, including configuration, certificates, code, and PuppetDB.

[Running Puppet on nodes](#) on page 320

Puppet automatically attempts to run on each of your nodes every 30 minutes. To trigger a Puppet run outside of the default 30-minute interval, you can manually run Puppet.

[Direct Puppet: a workflow for controlling change](#) on page 430

The orchestrator—used alongside other PE tools, such as Code Manager—allows you to control when and how infrastructure changes are made before they reach your production environment.

[Configure settings with Hiera](#) on page 190

Hiera is a hierarchy-based method of configuration management that relies on a “defaults, with overrides” system. When you add a parameter or setting to your Hiera data, Hiera searches through the data in the order it was written to find the value you want to change. Once found, it overrides the default value with the new parameter or setting. You can use Hiera to manage your PE configuration settings.

[Classification changes in high availability installations](#) on page 221

When you provision and enable a replica, the system makes a number of classification changes in order to manage high availability without affecting existing configuration.

Managing agent communication in geo-diverse installations

Typically, when you enable a replica using `puppet infrastructure enable replica`, the configuration tool automatically sets the same communication parameters for all agents. In *geo-diverse installations*, with load balancers or compilers in multiple locations, you must manually configure agent communication settings so that agents fail over to the appropriate load balancer or compiler.

To skip automatically configuring which Puppet servers and PCP brokers agents communicate with, use the `--skip-agent-config` flag when you enable a replica, for example:

```
puppet infrastructure enable replica example.puppet.com --skip-agent-config
```

To manually configure which load balancer or compiler agents communicate with, use one of these options:

- CSR attributes
 1. For each node, include a CSR attribute that identifies the location of the node, for example `pp_region` or `pp_datacenter`.
 2. Create child groups off of the **PE Agent** node group for each location.
 3. In each child node group, include the `puppet_enterprise::profile::agent` module and set the `server_list` parameter to the appropriate load balancer or compiler hostname.
 4. In each child node group, add a rule that uses the trusted fact created from the CSR attribute.
- Hiera

For each node or group of nodes, create a key/value pair that sets the `puppet_enterprise::profile::agent::server_list` parameter to be used by the **PE Agent** node group.
- Custom method that sets the `server_list` parameter in `puppet.conf`.

Promote a replica

If your master can't be restored, you can promote the replica to master to establish the replica as the new, permanent master.

1. Verify that the master is permanently offline.

If the master comes back online during promotion, your agents can get confused trying to connect to two active masters.

2. On the replica, as the root user, run `puppet infrastructure promote replica`

Promotion can take up to the amount of time it took to install PE initially. Don't make code or classification changes during or after promotion.

3. When promotion is complete, update any systems or settings that refer to the old master, such as PE client tool configurations, Code Manager hooks, Razor brokers, and CNAME records.

4. Deploy updated configuration to nodes by running Puppet or waiting for the next scheduled run.

Note: If you use the direct workflow, where agents use cached catalogs, you must manually deploy the new configuration by running `puppet job run --no-enforce-environment --query 'nodes {deactivated is null and expired is null}'`

5. (Optional) Provision a new replica in order to maintain high availability.

Note: Agent configuration must be updated before provisioning a new replica. If you re-use your old master's node name for the new replica, agents with outdated configuration might use the new replica as a master before it's fully provisioned.

Related information

[Running Puppet on nodes](#) on page 320

Puppet automatically attempts to run on each of your nodes every 30 minutes. To trigger a Puppet run outside of the default 30-minute interval, you can manually run Puppet.

[Direct Puppet: a workflow for controlling change](#) on page 430

The orchestrator—used alongside other PE tools, such as Code Manager—allows you to control when and how infrastructure changes are made before they reach your production environment.

Enable a new replica using a failed master

After promoting a replica, you can use your old master as a new replica, effectively swapping the roles of your failed master and promoted replica.

Before you begin

- The `puppet infrastructure run` command leverages built-in plans to automate certain management tasks. To use this command, you must be able to connect using SSH from your master to any nodes that the command modifies. You can establish an SSH connection using key forwarding, a local key file, or by specifying keys in `.ssh/config` on your master. For more information, see [OpenSSH configuration options](#).
- You must have token-based authentication configured.

To view all available parameters, use the `--help` flag. The logs for all `puppet infrastructure run` Bolt plans are located at `/var/log/puppetlabs/installer/bolt_info.log`.

You must be able to reach the failed master via SSH from the current master.

On your promoted replica logged in as root, run `puppet infrastructure run enable_ha_failover`, specifying these parameters:

- `host` — Hostname of the failed master. This node becomes your new replica.
- `topology` — Architecture used in your environment, either `mono` or `mono-with-compile`
- `replication_timeout_secs` — Optional. The number of seconds allowed to complete provisioning and enabling of the new replica before the command fails.
- `tmpdir` — Optional. Path to a directory to use for uploading and executing temporary files.

For example:

```
puppet infrastructure run enable_ha_failover host=<FAILED_MASTER_HOSTNAME>
topology=mono
```

The failed master is repurposed as a new replica.

Related information

[Generate a token using puppet-access](#) on page 243

Use `puppet-access` to generate an authentication tokens from the command line of any workstation (Puppet-managed or not), without the need to SSH into the Puppet master.

Forget a replica

Forgetting a replica cleans up classification and database state, preventing degraded performance over time.

Before you begin

Ensure you have a valid admin RBAC token. See [Generate a token using puppet-access](#) on page 243.

Run the `forget` command whenever a replica node is destroyed, even if you plan to replace it with a replica with the same name.

1. Verify that the replica to be removed is permanently offline.
2. On the master, as the root user, run `puppet infrastructure forget <REPLICA NODE NAME>`
The replica is decommissioned, the node is purged as an agent, and a Puppet run is completed on the master.

Reinitialize a replica

If you encounter certain errors on your replica after provisioning, you can reinitialize the replica. Reinitializing destroys and re-creates replica databases, as specified.

Before you begin

Your master must be fully functional and the replica must be able to communicate with the master.



CAUTION: If you reinitialize a functional replica that you already enabled, the replica is unavailable to serve as backup in a failover during reinitialization.

Reinitialization is not intended to fix slow queries or intermittent failures. Reinitialize your replica only if it's inoperational or you see replication errors.

1. On the replica, reinitialize databases as needed:
 - All databases: `puppet infrastructure reinitialize replica`
 - Specific databases: `puppet infrastructure reinitialize replica --db <DATABASE>`
where `<DATABASE>` is `pe-activity`, `pe-classifier`, `pe-orchestrator`, or `pe-rbac`.
2. Follow prompts to complete the reinitialization.

Accessing the console

The console is the web interface for Puppet Enterprise.

Use the console to:

- Manage node requests to join the Puppet deployment.
- Assign Puppet classes to nodes and groups.
- Run Puppet on specific groups of nodes.
- View reports and activity graphs.
- Browse and compare resources on your nodes.
- View package and inventory data.
- Manage console users and their access privileges.

Reaching the console

The console is served as a website over SSL, on whichever port you chose when installing the console component.

Let's say your console server is `console.domain.com`. If you chose to use the default port (443), you can omit the port from the URL and reach the console by navigating to `https://console.domain.com`.

If you chose to use port 8443, you reach the console at `https://console.domain.com:8443`.

Remember: Always use the `https` protocol handler. You cannot reach the console over plain `http`.

Accepting the console's certificate

The console uses an SSL certificate created by your own local Puppet certificate authority. Because this authority is specific to your site, web browsers won't know it or trust it, and you must add a security exception in order to access the console.

Adding a security exception for the console is safe to do. Your web browser warns you that the console's identity hasn't been verified by one of the external authorities it knows of, but that doesn't mean it's untrustworthy. Because you or another administrator at your site is in full control of which certificates the Puppet certificate authority signs, the authority verifying the site is *you*.

When your browser warns you that the certificate authority is invalid or unknown:

- In Chrome, click **Advanced**, then **Proceed to <CONSOLE ADDRESS>**.
- In Firefox, click **Advanced**, then **Add exception**.
- In Internet Explorer or Microsoft Edge, click **Continue to this website (not recommended)**.
- In Safari, click **Continue**.

Logging in

Accessing the console requires a username and password.

If you are an administrator setting up the console or accessing it for the first time, use the username and password you chose when you installed the console. Otherwise, get credentials from your site's administrator.

Because the console is the main point of control for your infrastructure, it is a good idea to prohibit your browser from storing the login credentials.

Generate a user password reset token

When users forget passwords or lock themselves out of the console by attempting to log in with incorrect credentials too many times, you need to generate a password reset token.

1. In the console, click **Access control > Users**.
2. Click the name of the user who needs a password reset token.
3. Click **Generate password reset**. Copy the link provided in the message and send it to the user.

Reset the console administrator password

To reset the administrator password for console access, use the `puppet infrastructure` command.

Log into the node running console services and reset the console admin password: `puppet infrastructure console_password --password=<MY_PASSWORD>`

Troubleshooting login to the PE admin account

If your directory contains multiple users with a login name of "admin," the PE admin account is unable to log in.

If you are locked out of PE as the admin user and there are no other users with administrator access who you can ask to reset the access control settings in the console, SSH into the box and use curl commands to reset the directory service settings.

For a box named centos7 the curl call looks like this:

```
curl -X PUT --cert /etc/puppetlabs/puppet/ssl/certs/centos7.pem --key /etc/puppetlabs/puppet/ssl/private_keys/centos7.pem --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem -H "Content-Type: application/json" -d {} https://centos7:4433/rbac-api/v1/ds
```

Managing access

Role-based access control, more succinctly called RBAC, is used to grant individual users the permission to perform specific actions. Permissions are grouped into user roles, and each user is assigned at least one user role.

By using permissions, you give the appropriate level of access and agency to each user. For example, you can grant users:

- The permission to grant password reset tokens to other users who have forgotten their passwords
- The permission to edit a local user's metadata
- The permission to deploy Puppet code to specific environments
- The permission to edit class parameters in a node group

You can do access control tasks in the console or using the RBAC API.

User permissions and user roles

The "role" in role-based access control refers to a system of user roles, which are assigned to user groups and their users. Those roles contain permissions, which define what a user can or can't do within PE.

When you add new users to PE, they can't actually do anything until they're associated with a user role, either explicitly through role assignment or implicitly through group membership and role inheritance. When a user is added to a role, they receive all the permissions of that role.

There are four default user roles: Administrators, Code Deployers, Operators, and Viewers. In addition, you can create custom roles.

For example, you might want to create a user role that grants users permission to view but not edit a specific subset of node groups. Or you might want to divide up administrative privileges so that one user role is able to reset passwords while another can edit roles and create users.

Permissions are additive. If a user is associated with multiple roles, that user is able to perform all of the actions described by all of the permissions on all of the applied roles.

Structure of user permissions

User permissions are structured as a triple of *type*, *permission*, and *object*.

- **Types** are everything that can be acted on, such as node groups, users, or user roles.
- **Permissions** are what you can do with each type, such as create, edit, or view.
- **Objects** are specific instances of the type.

Some permissions added to the Administrators user role might look like this:

Type	Permission	Object	Description
Node groups	View	PE Master	Gives permission to view the PE Master node group.
User roles	Edit	All	Gives permission to edit all user roles.

When no object is specified, a permission applies to all objects of that type. In those cases, the object is “All”. This is denoted by "*" in the API.

In both the console and the API, "*" is used to express a permission for which an object doesn't make sense, such as when creating users.

User permissions

The table below lists the available object types and their permissions, as well as an explanation of the permission and how to use it.

Note that types and permissions have both a display name, which is the name you see in the console interface, and a system name, which is the name you use to construct permissions in the API. In the following table, the display names are used.

Type	Permission	Definition
Certificate request	Accept and reject	Accept and reject certificate signing requests. Object must always be "*".
Console	View	View the PE console. Object must always be "*".
Directory service	View, edit, and test	View, edit, and test directory service settings. Object must always be "*".
Job orchestrator	Start, stop and view jobs	Start and stop jobs and tasks, view jobs and job progress, and view an inventory of nodes that are connected to the PCP broker.
Node groups	Create, edit, and delete child groups	Create new child groups, delete existing child groups, and modify every attribute of child groups except environment. This permission is inherited by all descendents of the node group.
Node groups	Edit child group rules	Edit the rules of descendents of a node group. This does not grant the ability to edit the rules of the group in the object field, only children of that group. This permission is inherited by all descendents of the node group.
Node groups	Edit classes, parameters, and variables	Edit every attribute of a node group except its environment and rule. This permission is inherited by all descendents of the node group.

Type	Permission	Definition
Node groups	Edit configuration data	Edit parameterized configuration data on a node group. This permission is inherited by all descendents of the node group.
Node groups	Edit parameters and variables	Edit the class parameters and variables of a node group's classes. This permission is inherited by all descendents of the node group.
Node groups	Set environment	Set the environment of a node group. This permission is inherited by all descendents of the node group.
Node groups	View	See all attributes of a node group, most notably the values of class parameters and variables. This permission is inherited by all descendents of the node group.
Nodes	Edit node data from PuppetDB	Edit node data imported from PuppetDB. Object must always be " * ".
Nodes	View node data from PuppetDB	View node data imported from PuppetDB. Object must always be " * ".
Nodes	View sensitive connection information in inventory service	View sensitive parameters stored in the inventory service for a connection. For example, user credentials. Object must always be " * ".
Plans	Run plans	Run specific plans on all nodes.
Puppet agent	Run Puppet on agent nodes	Trigger a Puppet run from the console or orchestrator. Object must always be " * ".
Puppet environment	Deploy code	Deploy code to a specific PE environment.
Puppet Server	Compile catalogs for remote nodes	Compile a catalog for any node managed by this PE instance. This permission is required to run impact analysis tasks in Continuous Delivery for Puppet Enterprise.

Type	Permission	Definition
Tasks	Run tasks	Run specific tasks on all nodes, a selected node group, or nodes that match a PQL query. Important: A task must be permitted to run on all nodes in order to run on nodes that are outside of the PuppetDB (over SSH or WinRM for example). As a result, users with such permissions can run tasks on any nodes they have the credentials to access.
User groups	Delete	Delete a user group. This can be granted per group.
User groups	Import	Import groups from the directory service for use in RBAC. Object must always be " * ".
User roles	Create	Create new roles. Object must always be " * ".
User roles	Edit	Edit and delete a role. Object must always be " * ".
User roles	Edit members	Change which users and groups a role is assigned to. This can be granted per role.
Users	Create	Create new local users. Remote users are "created" by that user authenticating for the first time with RBAC. Object must always be " * ".
Users	Edit	Edit a local user's data, such as name or email, and delete a local or remote user from PE. This can be granted per user.
Users	Reset password	Grant password reset tokens to users who have forgotten their passwords. This process also reinstates a user after the use has been revoked. This can be granted per user.
Users	Revoke	Revoke or disable a user. This means the user is no longer able to authenticate and use the console, node classifier, or RBAC. This permission also includes the ability to revoke the user's authentication tokens. This can be granted per user.

Display names and corresponding system names

The following table provides both the display and system names for the types and all their corresponding permissions.

Type (display name)	Type (system name)	Permission (display name)	Permission (system name)
Certificate requests	cert_requests	Accept and reject	accept_reject
Console	console_page	View	view
Directory service	directory_service	View, edit, and test	edit
Job orchestrator	orchestrator	Start, stop and view jobs	view
Node groups	node_groups	Create, edit, and delete child groups	modify_children
Node groups	node_groups	Edit child group rules	edit_child_rules
Node groups	node_groups	Edit classes, parameters, and variables	edit_classification
Node groups	node_groups	Edit configuration data	edit_config_data
Node groups	node_groups	Edit parameters and variables	edit_params_and_vars
Node groups	node_groups	Set environment	set_environment
Node groups	node_groups	View	view
Nodes	nodes	Edit node data from PuppetDB	edit_data
Nodes	nodes	View node data from PuppetDB	view_data
Nodes	nodes	View sensitive connection information in inventory service	view_inventory_sensitive
Plans	plans	Run Plans	run
Puppet agent	puppet_agent	Run Puppet on agent nodes	run
Puppet environment	environment	Deploy code	deploy_code
Puppet Server	puppetserver	Compile catalogs for remote nodes	compile_catalogs
Tasks	tasks	Run Tasks	run
User groups	user_groups	Import	import
User roles	user_roles	Create	create
User roles	user_roles	Edit	edit
User roles	user_roles	Edit members	edit_members
Users	users	Create	create
Users	users	Edit	edit
Users	users	Reset password	reset_password
Users	users	Revoke	disable

Related information

[Permissions endpoints](#) on page 259

You assign permissions to user roles to manage user access to objects. The `permissions` endpoints enable you to get information about available objects and the permissions that can be constructed for those objects.

Working with node group permissions

Node groups in the node classifier are structured hierarchically; therefore, node group permissions inherit. Users with specific permissions on a node group implicitly receive the permissions on any child groups below that node group in the hierarchy.

Two types of permissions affect a node group: those that affect a group itself, and those that affect the group's child groups. For example, giving a user the "Set environment" permission on a group allows the user to set the environment for that group and all of its children. On the other hand, assigning "Edit child group rules" to a group allows a user to edit the rules for any child group of a specified node group, but not for the node group itself. This allows some users to edit aspects of a group, while other users can be given permissions for all children of that group without being able to affect the group itself.

Due to the hierarchical nature of node groups, if a user is given a permission on the default (All) node group, this is functionally equivalent to giving them that permission on " * ".

Best practices for assigning permissions

Working with user permissions can be a little tricky. You don't want to grant users permissions that essentially escalate their role, for example. The following sections describe some strategies and requirements for setting permissions.

Grant edit permissions to users with create permissions

Creating new objects doesn't automatically grant the creator permission to view those objects. Therefore, users who have permission to create roles, for example, must also be given permission to edit roles, or they won't be able to see new roles that they create. Our recommendation is to assign users permission to edit all objects of the type that they have permission to create. For example:

Type	Permission	Object
User roles	Edit members	All (or " * ")
Users	Edit	All (or " * ")

Avoid granting overly permissive permissions

Operators, a default role in PE, have many of the same permissions as Administrators. However, we've intentionally limited this role's ability to edit user roles. This way, members of this group can do many of the same things as Administrators, but they can't edit (or enhance) their own permissions.

Similarly, avoid granting users more permissions than their roles allow. For example, if users have the `roles:edit:*` permission, they are able to add the `node_groups:view:*` permission to the roles they belong to, and subsequently see all node groups.

Give permission to edit directory service settings to the appropriate users

The directory service password is not redacted when the settings are requested in the API. Give `directory_service:edit:*` permission only to users who are allowed see the password and other settings.

The ability to reset passwords should be given only with other password permissions

The ability to help reset passwords for users who forgot them is granted by the `users:reset_password:<instance>` permission. This permission has the side effect of reinstating revoked users after the reset token is used. As such, the reset password permission should be given only to users who are also allowed to revoke and reinstate other users.

Creating and managing local users and user roles

Puppet Enterprise's role-based access control (RBAC) enables you to manage users—what they can create, edit, or view, and what they can't—in an organized, high-level way that is vastly more efficient than managing user permissions on a per-user basis. User roles are sets of permissions you can apply to multiple users. You can't assign permissions to single users in PE, only to user roles.

Remember: Each user must be assigned to one or more roles before they can log in and use PE.

Note: Puppet stores local accounts and directory service integration credentials securely. Local account passwords are hashed using SHA-256 multiple times along with a 32-bit salt. Directory service lookup credentials configured for directory lookup purposes are encrypted using AES-128. Puppet does not store the directory credentials used for authenticating to Puppet. These are different from the directory service lookup credentials.

Create a new user

These steps add a local user.

To add users from an external directory, see [Working with user groups from an external directory](#).

1. In the console, click **Access control** > **Users**.
2. In the **Full name** field, enter the user's full name.
3. In the **Login** field, enter a username for the user.
4. Click **Add local user**.

Give a new user access to the console

When you create new local users, you need to send them a password reset token so that they can log in for the first time.

1. On the **Users** page, click the user's full name.
2. Click **Generate password reset**.
3. Copy the link provided in the message and send it to the new user.

Create a new user role

RBAC has four predefined roles: Administrators, Code Deployers, Operators, and Viewers. You can also define your own custom user roles.

Users with the appropriate permissions, such as Administrators, can define custom roles. To avoid potential privilege escalation, only users who are allowed all permissions should be given the permission to edit user roles.

1. In the console, click **Access control** > **User roles**.
2. In the **Name** field, enter a name for the new user role.
3. (Optional) In the **Description** field, enter a description of the new user role.
4. Click **Add role**.

Assign permissions to a user role

You can mix and match permissions to create custom user roles that provide users with precise levels of access to PE actions.

Before you begin

Review [User permissions and user roles](#), which includes important information about how permissions work in PE.

1. On the **User roles** page, click a user role.
2. Click **Permissions**.
3. In the **Type** field, select the type of object you want to assign permissions for, such as **Node groups**.

4. In the **Permission** field, select the permission you want to assign, such as **View**.
5. In the **Object** field, select the specific object you want to assign the permission to. For example, if you are setting a permission to view node groups, select a specific node group this user role has permissions to view.
6. Click **Add permission**, and commit changes.

Related information

[Best practices for assigning permissions](#) on page 234

Working with user permissions can be a little tricky. You don't want to grant users permissions that essentially escalate their role, for example. The following sections describe some strategies and requirements for setting permissions.

Add a user to a user role

When you add users to a role, the user gains the permissions that are applied to that role. A user can't do anything in PE until they have been assigned to a role.

1. On the **User roles** page, click a user role.
2. Click **Member users**.
3. In the **User name** field, select the user you want to add to the user role.
4. Click **Add user**, and commit changes.

Remove a user from a user role

You can change a user's permissions by removing them from a user role. The user loses the permissions associated with the role, and won't be able to do anything in PE until they are assigned to a new role.

1. On the **User roles** page, click a user role.
2. Click **Member users**.
3. Locate the user you want to remove from the user role. Click **Remove**, and commit changes.

Revoke a user's access

If you want to remove a user's access to PE but not delete their account, you can revoke them. Revocation is also what happens when a user is locked out from too many incorrect password attempts.

1. In the console, click **Access control > Users**.
2. In the **Full name** column, select the user you want to revoke.
3. Click **Revoke user access**.

Tip: To unrevoke a user, follow the steps above and click **Reinstate user access**.

Delete a user

You can delete a user through the console. Note, however, that this action deletes only the user's Puppet Enterprise account, not the user's listing in any external directory service.

Deletion removes all data about the user except for their activity data, which continues to be stored in the database and remains viewable through the API.

1. In the console, click **Access control > Users**.
2. In the **Full name** column, locate the user you want to delete.
3. Click **Remove**.

Note: Users with superuser privileges cannot be deleted, and the **Remove** button does not appear for these users.



CAUTION: If a user is deleted from the console and then recreated with the same full name and login, PE issues the recreated user a new unique user ID. In this instance, queries for the login to the API's activity database return information on both the deleted user and the new user. However, in the console, the new user's **Activity** tab does not display information about the deleted user's account.

Delete a user role

You can delete a user role through the console.

When you delete a user role, users lose the permissions that the role gives them. This can impact their access to Puppet Enterprise if they have not been assigned other user roles.

1. In the console, click **Access control** > **User roles**.
2. In the **Name** column, locate the role you want to delete.
3. Click **Remove**.

Connecting external directory services to PE

Puppet Enterprise connects to external Lightweight Directory Access Protocol (LDAP) directory services through its role-based access control (RBAC) service. Because PE integrates with cloud LDAP service providers such as Okta, you can use existing users and user groups that have been set up in your external directory service.

Specifically, you can:

- Authenticate external directory users.
- Authorize access of external directory users based on RBAC permissions.
- Store and retrieve the groups and group membership information that has been set up in your external directory.

Note: Puppet stores local accounts and directory service integration credentials securely. Local account passwords are hashed using SHA-256 multiple times along with a 32-bit salt. Directory service lookup credentials configured for directory lookup purposes are encrypted using AES-128. Puppet does not store the directory credentials used for authenticating to Puppet. These are different from the directory service lookup credentials.

PE supports OpenLDAP and Active Directory. If you have predefined groups in your Active Directory or OpenLDAP directory, you can import these groups into the console and assign user roles to them. Users in an imported group inherit the permissions specified in assigned user roles. If new users are added to the group in the external directory, they also inherit the permissions of the role to which that group belongs.

Note: The connection to your external LDAP directory is read-only. If you want to make changes to remote users or user groups, you need to edit the information directly in the external directory.

Connect to an external directory service

PE connects to the external directory service when a user logs in or when groups are imported. The supported directory services are OpenLDAP and Active Directory.

1. In the console, click **Access control** > **External directory**.
2. Fill in the directory information.

All fields are required, except for **Login help**, **Lookup user**, **Lookup password**, **User relative distinguished name**, and **Group relative distinguished name**.

If you do not enter **User relative distinguished name** or **Group relative distinguished name**, RBAC searches the entire base DN for the user or group.

3. Click **Test connection** to ensure that the connection has been established. Save your settings after you have successfully tested them.

Note: This only tests the connection to the LDAP server. It does not test or validate LDAP queries.

External directory settings

The table below provides examples of the settings used to connect to an Active Directory service and an OpenLDAP service to PE. Each setting is explained in more detail below the table.

Important: The settings shown in the table are examples. You need to substitute these example settings with the settings used in your directory service.

Name	Example Active Directory settings	Example OpenLDAP settings
Directory name	My Active Directory	My Open LDAP Directory
Login help (optional)	https://myweb.com/ldaploginhelp	https://myweb.com/ldaploginhelp
Hostname	myhost.delivery.exampleservice.net	myhost.delivery.exampleservice.net
Port	389 (636 for LDAPS)	389 (636 for LDAPS)
Lookup user (optional)	cn=queryuser,cn=Users,dc=puppetlabs,	cn=admin,dc=delivery,dc=puppetlabs,dc=net
Lookup password (optional)	The lookup user's password.	The lookup user's password.
Connection timeout (seconds)	10	10
Connect using:	SSL	StartTLS
Validate the hostname?	Default is yes.	Default is yes.
Allow wildcards in SSL certificate?	Default is no.	Default is no.
Base distinguished name	dc=puppetlabs,dc=com	dc=puppetlabs,dc=com
User login attribute	sAMAccountName	cn
User email address	mail	mail
User full name	displayName	displayName
User relative distinguished name (optional)	cn=users	ou=users
Group object class	group	groupOfUniqueNames
Group membership field	member	uniqueMember
Group name attribute	name	displayName
Group lookup attribute	cn	cn
Group relative distinguished name (optional)	cn=groups	ou=groups
Turn off LDAP_MATCHING_RULE_IN_CHAIN?	Default is no.	Default is no.
Search nested groups?	Default is no.	Default is no.

Explanation of external directory settings

Directory name The name that you provide here is used to refer to the external directory service anywhere it is used in the PE console. For example, when you view a remote user in the console, the name that you provide in this field is listed in the console as the source for that user. Set any name of your choice.

Login help (optional) If you supply a URL here, a "Need help logging in?" link is displayed on the login screen. The href attribute of this link is set to the URL that you provide.

Hostname The FQDN of the directory service to which you are connecting.

Port The port that PE uses to access the directory service. The port is generally 389, unless you choose to connect using SSL, in which case it is generally 636.

Lookup user (optional) The distinguished name (DN) of the directory service user account that PE uses to query information about users and groups in the directory server. If a username is supplied, this user must have read access for all directory entries that are to be used in the console. We recommend that this user is restricted to read-only access to the directory service.

If your LDAP server is configured to allow anonymous binding, you do not need to provide a lookup user. In this case, the RBAC service binds anonymously to your LDAP server.

Lookup password (optional) The lookup user's password.

If your LDAP server is configured to allow anonymous binding, you do not need to provide a lookup password. In this case, the RBAC service binds anonymously to your LDAP server.

Connection timeout (seconds) The number of seconds that PE attempts to connect to the directory server before timing out. Ten seconds is fine in the majority of cases. If you are experiencing timeout errors, make sure the directory service is up and reachable, and then increase the timeout if necessary.

Connect using: Select the security protocol you want to use to connect to the external directory: **SSL** and **StartTLS** encrypt the data transmitted. **Plain text** is not a secure connection. In addition, to ensure that the directory service is properly identified, configure the `ds-trust-chain` to point to a copy of the public key for the directory service. For more information, see [Verify directory server certificates](#) on page 241.

Validate the hostname? Select **Yes** to verify that the Directory Services hostname used to connect to the LDAP server matches the hostname on the SSL certificate. This option is not available when you choose to connect to the external directory using plain text.

Allow wildcards in SLL certificate? Select **Yes** to allow a connection to a Directory Services server with a SSL certificates that use a wildcard (*) specification. This option is not available when you choose to connect to the external directory using plain text.

Base distinguished name When PE constructs queries to your external directory (for example to look up user groups or users), the queries consist of the relative distinguished name (RDN) (optional) + the base distinguished name (DN), and are then filtered by lookup/login attributes. For example, if PE wants to authenticate a user named Bob who has the RDN `ou=bob, ou=users`, it sends a query in which the RDN is concatenated with the DN specified in this field (for example, `dc=puppetlabs, dc=com`). This gives a search base of `ou=bob, ou=users, dc=puppetlabs, dc=com`.

The base DN that you provide in this field specifies where in the directory service tree to search for groups and users. It is the part of the DN that all users and groups that you want to use have in common. It is commonly the root DN (example `dc=example, dc=com`) but in the following example of a directory service entry, you could set the base DN to `ou=Puppet, dc=example, dc=com` because both the group and the user are also under the organizational unit `ou=Puppet`.

Example directory service entry

```
# A user named Harold
dn: cn=harold,ou=Users,ou=Puppet,dc=example,dc=com
objectClass: organizationalPerson
cn: harold
displayName: Harold J.
mail: harold@example.com
memberOf: inspectors
sAMAccountName: harold11

# A group Harold is in
dn: cn=inspectors,ou=Groups,ou=Puppet,dc=example,dc=com
objectClass: group
cn: inspectors
displayName: The Inspectors
member: harold
```

User login attribute This is the directory attribute that the user uses to log in to PE. For example, if you specify `sAMAccountName` as the user login attribute, Harold logs in with the username "harold11" because `sAMAccountName=harold11` in the example directory service entry provided above.

The value provided by the user login attribute must be unique among all entries under the User RDN + Base DN search base you've set up.

For example, say you've selected the following settings:

```
base DN = dc=example,dc=com
user RDN = null
user login attribute = cn
```

When Harold tries to log in, the console searches the external directory for any entries under `dc=example,dc=com` that have the attribute/value pair `cn=harold`. (This attribute/value pair does not need to be contained within the DN). However, if there is another user named Harold who has the DN `cn=harold,ou=OtherUsers,dc=example,dc=com`, two results are returned and the login does not succeed because the console does not know which entry to use. Resolve this issue by either narrowing your search base such that only one of the entries can be found, or using a value for login attribute that you know to be unique. This makes `sAMAccountName` a good choice if you're using Active Directory, as it must be unique across the entire directory.

User email address The directory attribute to use when displaying the user's email address in PE.

User full name The directory attribute to use when displaying the user's full name in PE.

User relative distinguished name (optional) The user RDN that you set here is concatenated with the base DN to form the search base when looking up a user. For example, if you specify `ou=users` for the user RDN, and your base DN setting is `ou=Puppet,dc=example,dc=com`, PE finds users that have `ou=users,ou=Puppet,dc=example,dc=com` in their DN.

This setting is optional. If you choose not to set it, PE searches for the user in the base DN (example: `ou=Puppet,dc=example,dc=com`). Setting a user RDN is helpful in the following situations:

- When you experience long wait times for operations that contact the directory service (either when logging in or importing a group for the first time). Specifying a user RDN reduces the number of entries that are searched.
- When you have more than one entry under your base DN with the same login value.

Tip: It is not currently possible to specify multiple user RDNs. If you want to filter RDNs when constructing your query, we suggest creating a new lookup user who only has read access for the users and groups you want to use in PE.

Group object class The name of an object class that all groups have.

Group membership field Tells PE how to find which users belong to which groups. This is the name of the attribute in the external directory groups that indicates who the group members are.

Group name attribute The attribute that stores the display name for groups. This is used for display purposes only.

Group lookup attribute The value used to import groups into PE. Given the example directory service entry provided above, the group lookup attribute would be `cn`. When specifying the Inspectors group in the console to import it, provide the name `inspectors`.

The value for this attribute must be unique under your search base. If you have users with the same login as the lookup of a group that you want to use, you can narrow the search base, use a value for the lookup attribute that you know to be unique, or specify the **Group object class** that all of your groups have in common but your users do not.

Tip: If you have a large number of nested groups in your group hierarchy, or you experience slowness when logging in with RBAC, we recommend disabling nested group search unless you need it for your authorization schema to work.

Group relative distinguished name (optional) The group RDN that you set here is concatenated with the base DN to form the search base when looking up a group. For example, if you specify `ou=groups` for the group RDN, and your base DN setting is `ou=Puppet,dc=example,dc=com`, PE finds groups that have `ou=groups,ou=Puppet,dc=example,dc=com` in their DN.

This setting is optional. If you choose not to set it, PE searches for the group in the base DN (example: `ou=Puppet,dc=example,dc=com`). Setting a group RDN is helpful in the following situations:

- When you experience long wait times for operations that contact the directory service (either when logging in or importing a group for the first time). Specifying a group RDN reduces the number of entries that are searched.
- When you have more than one entry under your base DN with the same lookup value.

Tip: It is not currently possible to specify multiple group RDNs. If you want to filter RDNs when constructing your query, create a new lookup user who only has read access for the users and groups you plan to use in PE.

Note: At present, PE supports only a single Base DN. Use of multiple user RDNs or group RDNs is not supported.

Turn off LDAP_MATCHING_RULE_IN_CHAIN? Select **Yes** to turn off the LDAP matching rule that looks up the chain of ancestry for an object until it finds a match. For organizations with a large number of group memberships, matching rule in chain can slow performance.

Search nested groups? Select **Yes** to search for groups that are members of an external directory group. For organizations with a large number of nested group memberships, searching nested groups can slow performance.

Related information

[PUT /ds](#) on page 264

Replaces current directory service connection settings. Authentication is required.

Verify directory server certificates

To ensure that RBAC isn't being subjected to a Man-in-the Middle (MITM) attack, verify the directory server's certificate.

When you select SSL or StartTLS as the security protocol to use for communications between PE and your directory server, the connection to the directory is encrypted. To ensure that the directory service is properly identified, configure the `ds-trust-chain` to point to a copy of the public key for the directory service.

The RBAC service verifies directory server certificates using a trust store file, in Java Key Store (JKS), PEM, or PKCS12 format, that contains the chain of trust for the directory server's certificate. This file needs to exist on disk in a location that is readable by the user running the RBAC service.

To turn on verification:

1. In the console, click **Classification**.
2. Open the **PE Infrastructure** node group and select the **PE Console** node group.
3. Click **Configuration**. Locate the `puppet_enterprise::profile::console` class.
4. In the **Parameter** field, select `rbac_ds_trust_chain`.
5. In the **Value** field, set the absolute path to the trust store file.
6. Click **Add parameter**, and commit changes.
7. To make the change take effect, run Puppet. Running Puppet restarts pe-console-services.

After this value is set, the directory server's certificate is verified whenever RBAC is configured to connect to the directory server using SSL or StartTLS.

Working with user groups from an external directory service

You don't explicitly add remote users to PE. Instead, after the external directory service has been successfully connected, remote users must log into PE, which creates their user record.

If the user belongs to an external directory group that has been imported into PE and then assigned to a role, the user is assigned to that role and gains the privileges of the role. Roles are additive: You can assign users to more than one role, and they gain the privileges of all the roles to which they are assigned.

Import a user group from an external directory service

You import existing external directory groups to PE explicitly, which means you add the group by name.

1. In the console, click **Access control > User groups**.

User groups is available only if you have established a connection with an external directory.

2. In the **Login** field, enter the name of a group from your external directory.

3. Click **Add group**.

Remember: No user roles are listed until you add this group to a role. No users are listed until a user who belongs to this group logs into PE.

Troubleshooting: A PE user and user group have the same name

If you have both a PE user and an external directory user group with the exact same name, PE throws an error when you try to log on as that user or import the user group.

To work around this problem, you can change your settings to use different RDNs for users and groups. This works as long as all of your users are contained under one RDN that is unique from the RDN that contains all of your groups.

Assign a user group to a user role

After you've imported a group, you can assign it a user role, which gives each group member the permissions associated with that role. You can add user groups to existing roles, or you can create a new role, and then add the group to the new role.

1. In the console, click **Access control > User roles**.
2. Click the role you want to add the user group to.
3. Click **Member groups**. In the **Group name** field, select the user group you want to add to the user role.
4. Click **Add group**, and commit changes.

Delete a user group

You can delete a user group in the console. Users who were part of the deleted group lose the permissions associated with roles assigned to the group.

Remember: This action removes the group only from Puppet Enterprise, not from the associated external directory service.

1. In the console, click **Access control > User groups**.
User groups is available only if you have established a connection with an external directory.
2. Locate the group that you wish to delete.
3. Click **Remove**.

Removing a remote user's access to PE

In order to fully revoke the remote user's access to Puppet Enterprise, you must also remove the user from the external directory groups accessed by PE.

Deleting a remote user's PE account does not automatically prevent that user from accessing PE in the future. So long as the remote user is still a member of a group in an external directory that PE is configured to access, the user retains the ability to log into PE.

If you delete a user from your external directory service but not from PE, the user can no longer log in, but any generated tokens or existing console sessions continue to be valid until they expire. To invalidate the user's tokens or sessions, revoke the user's PE account, which also automatically revokes all tokens for the user. You must manually delete the user for their account record to disappear.

Token-based authentication

Puppet Enterprise users generate tokens to authenticate their access to certain PE command-line tools and API endpoints. Authentication tokens are tied to the permissions granted to the user through RBAC, and provide the user with the appropriate access to HTTP requests.

Authentication tokens manage access to the following PE services:

- Puppet orchestrator

- Code Manager
- Node classifier
- Role-based access control (RBAC)
- Activity service
- PuppetDB

You can generate tokens with the `puppet-access` command or with the API endpoint.

Remember: For security reasons, authentication tokens can be generated only for revocable users. The admin user and `api_user` cannot be revoked.

Related information

[Installing PE client tools](#) on page 169

PE client tools are a set of command line tools that let you access Puppet Enterprise services from a workstation that might or might not be managed by Puppet.

[User permissions](#) on page 230

The table below lists the available object types and their permissions, as well as an explanation of the permission and how to use it.

Generate a token using puppet-access

Use `puppet-access` to generate an authentication tokens from the command line of any workstation (Puppet-managed or not), without the need to SSH into the Puppet master.

Before you begin

Install the PE client tools package and configure `puppet-access`.

1. Choose one of the following options, depending on how long you need your token to last:

- To generate a token with the default five-minute lifetime, run `puppet-access login`.
- To generate a token with a specific lifetime, run `puppet-access login --lifetime <TIME PERIOD>`.

For example, `puppet-access login --lifetime 5h` generates a token that lasts five hours.

2. When prompted, enter the same username and password that you use to log into the PE console.

The `puppet-access` command contacts the token endpoint in the RBAC v1 API. If your login credentials are correct, the RBAC service generates a token.

3. The token is generated and stored in a file for later use. The default location for storing the token is `~/.puppetlabs/token`. You can print the token at any time using `puppet-access show`.

You can continue to use this token until it expires, or until your access is revoked. The token has the exact same set of permissions as the user that generated it.



CAUTION: If you run the login command with the `--debug` flag, the client outputs the token, as well as the username and password. For security reasons, exercise caution when using the `--debug` flag with the login command.

Related information

[Setting a token-specific lifetime](#) on page 246

Tokens have a default lifetime of five minutes, but you can set a different lifetime for your token when you generate it. This allows you to keep one token for multiple sessions.

Generate a token for use by a service

If you need to generate a token for use by a service and the token doesn't need to be saved, use the `--print` option.

Before you begin

Install the PE client tools package and configure `puppet-access`.

Run `puppet-access login [username] --print`.

This command generates a token, and then displays the token content as stdout (standard output) rather than saving it to disk.

Tip: When generating a token for a service, consider specifying a longer token lifetime so that you don't have to regenerate the token too frequently.

Configuring puppet-access

The `puppet-access` command allows users to generate and manage authentication tokens from the command line of any workstation (Puppet-managed or not), without the need to SSH into the Puppet master.

The configuration file for `puppet-access` allows you to generate tokens from the CLI without having to pass additional flags.

Whether you are running `puppet-access` on a PE-managed server or installing it on a separate work station, you need a global configuration file and a user-specified configuration file.

Global configuration file

The global configuration file is located at:

- **On *nix systems:** `/etc/puppetlabs/client-tools/puppet-access.conf`
- **On Windows systems:** `C:/ProgramData/PuppetLabs/client-tools/puppet-access.conf`

On machines managed by Puppet Enterprise, this global configuration file is created for you. The configuration file is formatted in JSON. For example:

```
{
  "service-url": "https://<CONSOLE HOSTNAME>:4433/rbac-api",
  "token-file": "~/.puppetlabs/token",
  "certificate-file": "/etc/puppetlabs/puppet/ssl/certs/ca.pem"
}
```

PE determines and adds the `service-url` setting.

If you're running `puppet-access` from a workstation not managed by PE, you must create the global file and populate it with the configuration file settings.

User-specified configuration file

The user-specified configuration file is located at `~/.puppetlabs/client-tools/puppet-access.conf` for both *nix and Windows systems. You must create the user-specified file and populate it with the configuration file settings. A list of configuration file settings is found in the next section.

The user-specified configuration file always takes precedence over the global configuration file. For example, if the two files have contradictory settings for the `token-file`, the user-specified settings prevail.

Important: User-specified configuration files must be in JSON format; HOCON and INI-style formatting are not supported.

Configuration file settings for puppet-access

As needed, you can manually add configuration settings to your user-specified or global `puppet-access` configuration file.

The class that manages the global configuration file is `puppet_enterprise::profile::controller`.

You can also change configuration settings by specifying flags when using the `puppet-access` command in the command line.

Setting	Description	Command line flag
token-file	The location for storing authentication tokens. Defaults to <code>~/.puppetlabs/token</code> .	<code>-t, --token-file</code>
certificate-file	The location of the CA that signed the console-services server's certificate. Defaults to the PE CA cert location, <code>/etc/puppetlabs/puppet/ssl/certs/ca.pem</code> .	<code>--ca-cert</code>
config-file	Changes the location of your configuration file. Defaults to <code>~/.puppetlabs/client-tools/puppet-access.conf</code> .	<code>-c, --config-file</code>
service-url	The URL for your RBAC API. Defaults to the URL automatically determined during the client tools package installation process, generally <code>https://<CONSOLE HOSTNAME>:4433/rbac-api</code> . You typically need to change this only if you are moving the console server.	<code>--service-url</code>

Generate a token using the API endpoint

The RBAC v1 API includes a token endpoint, which allows you to generate a token using curl commands.

1. On the command line, post your RBAC user login credentials using the token endpoint.

```
curl -k -X POST -H 'Content-Type: application/json' -d '{"login":
"<YOUR PE USER NAME>", "password": "<YOUR PE PASSWORD>"}' https://
$<HOSTNAME>:4433/rbac-api/v1/auth/token
```

The command returns a JSON object containing the key `token` and the token itself.



CAUTION: If you are using curl commands with the `-k` insecure SSL connection option, keep in mind that you are vulnerable to a person-in-the-middle attack.

2. Save the token. Depending on your workflow, either:

- Copy the token to a text file.
- Save the token as an environment variable using `export TOKEN=<PASTE THE TOKEN HERE>`.

You can continue to use this token until it expires, or until your access is revoked. The token has the exact same set of permissions as the user that generated it.

Use a token with the PE API endpoints

The example below shows how to use a token in an API request. In this example, you use the `/users/current` endpoint of the RBAC v1 API to get information about the current authenticated user.

Before you begin

Generate a token and save it as an environment variable using `export TOKEN=<PASTE THE TOKEN HERE>`.

Run the following command: `curl -k -X GET https://<HOSTNAME>:4433/rbac-api/v1/users/current -H "X-Authentication:$TOKEN"`

The command above uses the `X-Authentication` header to supply the token information.

In some cases, such as GitHub webhooks, you might need to supply the token in a token parameter by specifying the request as follows: `curl -k -X GET https://<HOSTNAME>:4433/rbac-api/v1/users/current?token=$TOKEN`



CAUTION: If you are using curl commands with the `-k` insecure SSL connection option, keep in mind that you are vulnerable to a person-in-the-middle attack.

Change the token's default lifetime

Tokens have a default authentication lifetime of five minutes, but this default value can be adjusted in the console. You can also change a token's maximum authentication lifetime, which defaults to 10 years.

1. In the console, click **Classification**.
2. Open the **PE Infrastructure** node group and click the **PE Console** node group.
3. On the **Configuration** tab, find the **puppet_enterprise::profile::console** class.
4. In the **Parameter** field, select **rbac_token_auth_lifetime** to change the default lifetime of all tokens, or **rbac_token_maximum_lifetime** to adjust the maximum allowable lifetime for all tokens.
5. In the **Value** field, enter the new default authentication lifetime.
Specify a numeric value followed by "y" (years), "d" (days), "h" (hours), "m" (minutes), or "s" (seconds). For example, "12h" generates a token valid for 12 hours.
Do not add a space between the numeric value and the unit of measurement. If you do not specify a unit, it is assumed to be seconds.
The **rbac_token_auth_lifetime** cannot exceed the **rbac_token_maximum_lifetime** value.
6. Click **Add parameter**, and commit changes.

Setting a token-specific lifetime

Tokens have a default lifetime of five minutes, but you can set a different lifetime for your token when you generate it. This allows you to keep one token for multiple sessions.

Specify a numeric value followed by "y" (years), "d" (days), "h" (hours), "m" (minutes), or "s" (seconds). For example, "12h" generates a token valid for 12 hours.

Do not add a space between the numeric value and the unit of measurement. If you do not specify a unit, it is assumed to be seconds.

Use the `--lifetime` parameter if using `puppet-access` to generate your token. For example: `puppet-access login --lifetime 1h`.

Use the `lifetime` value if using the RBAC v1 API to generate your token. For example: `{"login": "<YOUR PE USER NAME>", "password": "<YOUR PE PASSWORD>", "lifetime": "1h"}`.

Set a token-specific label

You can affix a plain-text, user-specific label to tokens you generate with the RBAC v1 API. Token labels allow you to more readily refer to your token when working with RBAC API endpoints, or when revoking your own token.

Token labels are assigned on a per-user basis: two users can each have a token labelled "my token", but a single user cannot have two tokens both labelled "my token." You cannot use labels to refer to other users' tokens.

Generate a token using the `token` endpoint of the RBAC API, using the `label` value to specify the name of your token.

```
{"login": "<YOUR PE USER NAME>", "password": "<YOUR PE PASSWORD>", "label": "Ava's token"}
```

Labels must be no longer than 200 characters, must not contain commas, and must contain something besides whitespace.

Revoking a token

Revoke tokens by username, label, or full token through the `token` endpoint of the v2 RBAC API. All token revocation attempts are logged in the activity service, and can be viewed on the user's **Activity** tab in the console.

You can revoke your own token by username, label, or full token. You can also revoke any other full token you possess. Users with the permission to revoke other users can also revoke those users' tokens, as the `users:disable` permission includes token revocation. Revoking a user's tokens does not revoke the user's PE account.

If a user's account is revoked, all tokens associated with that user account are also automatically revoked.

Note: If a remote user generates a token and then is deleted from your external directory service, the deleted user cannot log into the console. However, because the token has already been authenticated, the RBAC service does not contact the external directory service again when the token is used in the future. To fully remove the token's access, you need to manually revoke or delete the user from PE.

Delete a token file

If you logged into `puppet-access` to generate a token, you can remove the file that stores that token simply by running the `delete-token-file` command. This is useful if you are working on a server that is used by other people.

Deleting the token file prevents other users from using your authentication token, but does not actually revoke the token. After the token has expired, there's no risk of obtaining the contents of the token file.

From the command line, run one of the following commands, depending on the path to your token file:

- If your token is at the default token file location, run `puppet-access delete-token-file`.
- If you used a different path to store your token file, run `puppet-access delete-token-file --token-path <YOUR TOKEN PATH>`.

View token activity

Token activity is logged by the activity service. You can see recent token activity on any user's account in the console.

1. In the console, click **Access control** > **Users**. Click the full name of the user you are interested in.
2. Click the **Activity** tab.

RBAC API v1

The role-based access control (RBAC) service enables you to manage users, directory groups, and roles.

Related information

[API index](#) on page 33

APIs allow you to interact with Puppet and Puppet Enterprise (PE) applications from your own code or application integration hooks.

Endpoints

The role-based access control (RBAC) service enables you to manage users, directory groups, and roles.

For general information about forming HTTP requests to the API, see the forming requests page. For information on errors encountered while using the RBAC v1 API, see the RBAC service errors page.

Tip: In addition to the endpoints on this page and in the v2 RBAC service API, there are endpoints that you can use to check the health of the RBAC service. These are available through the status API documentation.

Endpoint	Use
users	Manage local users as well as those from a directory service, get lists of users, and create new local users.
groups	Get lists of groups and add a new remote user group.
roles	Get lists of user roles and create new roles.
permissions	Get information about available objects and the permissions that can be constructed for those objects.
ds (Directory service)	Get information about the directory service, test your directory service connection, and replace directory service connection settings.
password	Generate password reset tokens and update user passwords.
token	Generate the authentication tokens used to access PE.
rbac-service	Check the status of the RBAC service.

Forming RBAC API requests

Token-based authentication is required to access the RBAC API. You can authenticate requests by using either user authentication tokens or whitelisted certificates.

By default, the RBAC service listens on port 4433. All endpoints are relative to the `/rbac-api/` path. So, for example, the full URL for the `/v1/users` endpoint on localhost is `https://localhost:4433/rbac-api/v1/users`.

Authentication using tokens

Insert a user authentication token in an RBAC API request.

1. Generate a token: `puppet-access login`
2. Print the token and copy it: `puppet-access show`
3. Save the token as an environment variable: `export TOKEN=<PASTE THE TOKEN HERE>`
4. Include the token variable in your API request:

```
curl -k -X GET https://<HOSTNAME>:<PORT>/rbac-api/v1/events?
service_id=classifier -H "X-Authentication:$TOKEN"
```

The example above uses the X-Authentication header to supply the token information. In some cases, such as GitHub webhooks, you might need to supply the token in a token parameter. To supply the token in a token parameter, specify the request as follows:

```
curl -k -X GET https://<HOSTNAME>:<PORT>/rbac-api/v1/users/current?token=
$TOKEN
```



CAUTION: Be aware when using the token parameter method that the token parameter might be recorded in server access logs.

Authentication using whitelisted certificate

You can also authenticate requests using a certificate listed in RBAC's certificate whitelist, located at `/etc/puppetlabs/console-services/rbac-certificate-whitelist`. Note that if you edit this file, you must reload the `pe-console-services` service (run `sudo service pe-console-services reload`) for your changes to take effect.

Attach the certificate using the command line, as demonstrated in the example curl query below. You must have the whitelisted certificate name (which must match a name in the `/etc/puppetlabs/console-services/rbac-certificate-whitelist` file) and the private key to run the script.

```
curl -X GET https://<HOSTNAME>:<PORT>/rbac-api/v1/users \
  --cert /etc/puppetlabs/puppet/ssl/certs/<WHITELISTED CERTNAME>.pem \
  --key /etc/puppetlabs/puppet/ssl/private_keys/<WHITELISTED CERTNAME>.pem \
  --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem -H "Content-Type:
  application/json"
```

You do not need to use an agent certificate for authentication. You can use `puppet cert generate` to create a new certificate specifically for use with the API.

Related information

[Token-based authentication](#) on page 242

Puppet Enterprise users generate tokens to authenticate their access to certain PE command-line tools and API endpoints. Authentication tokens are tied to the permissions granted to the user through RBAC, and provide the user with the appropriate access to HTTP requests.

Content-type headers in the RBAC API

RBAC accepts only JSON payloads in PUT and POST requests.

If a payload is provided, it is important to specify that the content is in JSON format. Thus, all PUT and POST requests with non-empty bodies should have the `Content-Type` header set to `application/json`.

Users endpoints

RBAC enables you to manage local users as well as those who are created remotely, on a directory service. With the users endpoints, you can get lists of users and create new local users.

Users keys

The following keys are used with the RBAC v1 API's users endpoints.

Key	Explanation	Example
id	A UUID string identifying the user.	"4fee7450-54c7-11e4-916c-0800200c9a"
login	A string used by the user to log in. Must be unique among users and groups.	"admin"
email	An email address string. Not currently utilized by any code in PE.	"hill@example.com"
display_name	The user's name as a string.	"Kalo Hill"
role_ids	An array of role IDs indicating which roles should be directly assigned to the user. An empty array is valid.	[3 6 5]
is_group	These flags indicate the type of user. <code>is_group</code> should always be false for a user.	true/false
is_remote		
is_superuser		
is_revoked	Setting this flag to <code>true</code> prevents the user from accessing any routes until the flag is unset or the user's password is reset via token.	true/false

Key	Explanation	Example
last_login	This is a timestamp in UTC-based ISO-8601 format (YYYY-MM-DDThh:mm:ssZ) indicating when the user last logged in. If the user has never logged in, this value is null.	"2014-05-04T02:32:00Z"
inherited_role_ids (remote users only)	An array of role IDs indicating which roles a remote user inherits from their groups.	[9 1 3]
group_ids (remote users only)	An array of UUIDs indicating which groups a remote user inherits roles from.	["3a96d280-54c9-11e4-916c-0800200c9a66"]

GET /users

Fetches all users, both local and remote (including the superuser). Supports filtering by ID through query parameters. Authentication is required.

Request format

To request all the users:

```
GET /users
```

To request specific users, add a comma-separated list of user IDs:

```
GET /users?
id=fe62d770-5886-11e4-8ed6-0800200c9a66,1cadd0e0-5887-11e4-8ed6-0800200c9a66
```

Response format

The response is a JSON object that lists the metadata for all requested users.

For example:

```
[ {
  "id": "fe62d770-5886-11e4-8ed6-0800200c9a66",
  "login": "Kalo",
  "email": "kalohill@example.com",
  "display_name": "Kalo Hill",
  "role_ids": [1,2,3...],
  "is_group" : false,
  "is_remote" : false,
  "is_superuser" : true,
  "is_revoked": false,
  "last_login": "2014-05-04T02:32:00Z"
}, {
  "id": "07d9c8e0-5887-11e4-8ed6-0800200c9a66",
  "login": "Jean",
  "email": "jeanjackson@example.com",
  "display_name": "Jean Jackson",
  "role_ids": [2, 3],
  "inherited_role_ids": [5],
  "is_group" : false,
  "is_remote" : true,
  "is_superuser" : false,
  "group_ids": [ "2ca57e30-5887-11e4-8ed6-0800200c9a66" ],
  "is_revoked": false,
  "last_login": "2014-05-04T02:32:00Z"
}
```

```
{, {
  "id": "1cadd0e0-5887-11e4-8ed6-0800200c9a66",
  "login": "Amari",
  "email": "amariperez@example.com",
  "display_name": "Amari Perez",
  "role_ids": [2, 3],
  "inherited_role_ids": [5],
  "is_group" : false,
  "is_remote" : true,
  "is_superuser" : false,
  "group_ids": ["2ca57e30-5887-11e4-8ed6-0800200c9a66"],
  "is_revoked": false,
  "last_login": "2014-05-04T02:32:00Z"
}]
```

GET /users/<sid>

Fetches a single user by its subject ID (sid). Authentication is required.

Response format

For all users, the user contains an ID, a login, an email, a display name, a list of role-ids the user is directly assigned to, and the last login time in UTC-based ISO-8601 format (YYYY-MM-DDThh:mm:ssZ), or null if the user has not logged in yet. It also contains an "is_revoked" field, which, when set to true, prevents a user from authenticating.

For example:

```
{ "id": "fe62d770-5886-11e4-8ed6-0800200c9a66",
  "login": "Amari",
  "email": "amariperez@example.com",
  "display_name": "Amari Perez",
  "role_ids": [1,2,3...],
  "is_group" : false,
  "is_remote" : false,
  "is_superuser" : false,
  "is_revoked": false,
  "last_login": "2014-05-04T02:32:00Z" }
```

For remote users, the response additionally contains a field indicating the IDs of the groups the user inherits roles from and the list of inherited role IDs.

For example:

```
{ "id": "07d9c8e0-5887-11e4-8ed6-0800200c9a66",
  "login": "Jean",
  "email": "jeanjackson@example.com",
  "display_name": "Jean Jackson",
  "role_ids": [2,3...],
  "inherited_role_ids": [],
  "is_group" : false,
  "is_remote" : true,
  "is_superuser" : false,
  "group_ids": ["b28b8790-5889-11e4-8ed6-0800200c9a66"],
  "is_revoked": false,
  "last_login": "2014-05-04T02:32:00Z" }
```

GET /users/current

Fetches the data about the current authenticated user, with the exact same behavior as GET /users/<sid>, except that <sid> is assumed from the authentication context. Authentication is required.

POST /users

Creates a new local user. You can add the new user to user roles by specifying an array of roles in `role_ids`. You can set a password for the user in `password`. For the password to work in the PE console, it needs to be a minimum of six characters. Authentication is required.

Request format

Accepts a JSON body containing entries for `email`, `display_name`, `login`, `role_ids`, and `password`. The `password` field is optional. The created account is not useful until the password is set.

For example:

```
{ "login": "Kalo",
  "email": "kalohill@example.com",
  "display_name": "Kalo Hill",
  "role_ids": [1,2,3],
  "password": "yabbdabba" }
```

Response format

If the create operation is successful, a 201 Created response with a location header that points to the new resource is returned.

Error responses

If the email or login for the user conflicts with an existing user's login, a 409 Conflict response is returned.

PUT /users/<sid>

Replaces the user with the specified ID (sid) with a new user object. Authentication is required.

Request format

Accepts an updated user object with all keys provided when the object is received from the API. The behavior varies based on user type. All types have a `role_id` array that indicates all the roles the user should belong to. An empty roles array removes all roles directly assigned to the group.

The below examples show what keys must be submitted. Keys marked with asterisks are the only ones that can be changed via the API.

An example for a local user:

```
{ "id": "c8b2c380-5889-11e4-8ed6-0800200c9a66",
  **"login": "Amari",**
  **"email": "amariperez@example.com",**
  **"display_name": "Amari Perez",**
  **"role_ids": [1, 2, 3],**
  "is_group" : false,
  "is_remote" : false,
  "is_superuser" : false,
  **"is_revoked": false,**
  "last_login": "2014-05-04T02:32:00Z" }
```

An example for a remote user:

```
{ "id": "3271fde0-588a-11e4-8ed6-0800200c9a66",
  "login": "Jean",
```

```
"email": "jeanjackson@example.com",
"display_name": "Jean Jackson",
**"role_ids": [4, 1]**,
"inherited_role_ids": [],
"group_ids": [],
"is_group" : false,
"is_remote" : true,
"is_superuser" : false,
**"is_revoked": false,**
"last_login": "2014-05-04T02:32:00Z"}
```

Response format

The request returns a 200 OK response, along with the user object with the changes made.

Error responses

If the login for the user conflicts with an existing user login, a 409 Conflict response is returned.

DELETE /users/<sid>

Deletes the user with the specified ID (sid), regardless of whether they are a user defined in RBAC or a user defined by a directory service. In the case of directory service users, while this action removes a user from the console, that user is still able to log in (at which point they are re-added to the console) if they are not revoked. Authentication is required.

Remember: The admin user and the api_user cannot be deleted.

Request format

For example, to delete a user with the ID 3982a629-1278-4e38-883e-12a7cac91535 by using a curl command:

```
curl -X DELETE \
--cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem \
--cert /opt/puppet/share/puppet-dashboard/certs/pe-internal-
dashboard.cert.pem \
--key /opt/puppet/share/puppet-dashboard/certs/pe-internal-
dashboard.private_key.pem \
https://$(hostname -f):4433/rbac-api/v1/
users/3982a629-1278-4e38-883e-12a7cac91535
```

Response format

If the user is successfully deleted, a 204 No Content response with an empty body is returned.

Error responses

If the current user does not have the `users:edit` permission for this user group, a 403 Forbidden response is returned.

If a user with the provided identifier does not exist, a 404 Not Found response is returned.

User group endpoints

Groups are used to assign roles to a group of users, which is vastly more efficient than managing roles for each user individually. The `groups` endpoints enable you to get lists of groups, and to add a new directory group.

Remember: Group membership is determined by the directory service hierarchy and as such, local users cannot be in directory groups.

User group keys

The following keys are used with the RBAC v1 API's groups endpoints.

Key	Explanation	Example
id	A UUID string identifying the group.	"c099d420-5557-11e4-916c-0800200c9a66"
login	the identifier for the user group on the directory server.	"poets"
display_name	The group's name as a string.	"Poets"
role_ids	An array of role IDs indicating which roles should be inherited by the group's members. An empty array is valid. This is the only field that can be updated via RBAC; the rest are immutable or synced from the directory service.	[3 6 5]
is_group	These flags indicate that the group is a group.	true, true, false, respectively
is_remote		
is_superuser		
is_revoked	Setting this flag to true currently does nothing for a group.	true/false
user_ids	An array of UUIDs indicating which users inherit roles from this group.	["3a96d280-54c9-11e4-916c-0800200c9a66"]

GET /groups

Fetches all groups. Supports filtering by ID through query parameters. Authentication is required.

Request format

The following requests all the groups:

```
GET /groups
```

To request only some groups, add a comma-separated list of group IDs:

```
GET /groups?
id=65a068a0-588a-11e4-8ed6-0800200c9a66,75370a30-588a-11e4-8ed6-0800200c9a66
```

Response format

The response is a JSON object that lists the metadata for all requested groups.

For example:

```
[ {
  "id": "65a068a0-588a-11e4-8ed6-0800200c9a66",
  "login": "hamsters",
  "display_name": "Hamster club",
  "role_ids": [2, 3],
  "is_group" : true,
  "is_remote" : true,
  "is_superuser" : false,
  "user_ids": [ "07d9c8e0-5887-11e4-8ed6-0800200c9a66" ] }
], {
```

```

    "id": "75370a30-588a-11e4-8ed6-0800200c9a66",
    "login": "chinchilla",
    "display_name": "Chinchilla club",
    "role_ids": [2, 1],
    "is_group" : true,
    "is_remote" : true,
    "is_superuser" : false,
    "user_ids":
  [ "1cadd0e0-5887-11e4-8ed6-0800200c9a66", "5c1ab4b0-588b-11e4-8ed6-0800200c9a66" ]
}, {
  "id": "ccdbde50-588a-11e4-8ed6-0800200c9a66",
  "login": "wombats",
  "display_name": "Wombat club",
  "role_ids": [2, 3],
  "is_group" : true,
  "is_remote" : true,
  "is_superuser" : false,
  "user_ids": []
}]

```

GET /groups/<sid>

Fetches a single group by its subject ID (sid). Authentication is required.

Response format

The response contains an ID for the group and a list of `role_ids` the group is directly assigned to.

For directory groups, the response contains the display name, the login field, a list of `role_ids` directly assigned to the group, and `user_ids` containing IDs of the remote users that belong to that group.

For example:

```

{ "id": "65a068a0-588a-11e4-8ed6-0800200c9a66",
  "login": "hamsters",
  "display_name": "Hamster club",
  "role_ids": [2, 3],
  "is_group" : true,
  "is_remote" : true,
  "is_superuser" : false,
  "user_ids": [ "07d9c8e0-5887-11e4-8ed6-0800200c9a66" ] }

```

Error responses

If the user who submits the GET request has not successfully authenticated, a 401 Unauthorized response is returned.

If the current user does not have the appropriate user permissions to request the group data, a 403 Forbidden response is returned.

POST /groups

Creates a new remote group and attaches to the new group any roles specified in its roles list. Authentication is required.

Request format

Accepts a JSON body containing an entry for `login`, and an array of `role_ids` to assign to the group initially.

For example:

```

{ "login": "Augmentators",
  "role_ids": [1,2,3] }

```

Response format

If the create operation is successful, a 201 Created response with a location header that points to the new resource is returned.

Error responses

If the login for the group conflicts with an existing group login, a 409 Conflict response is returned.

PUT /groups/<sid>

Replaces the group with the specified ID (sid) with a new group object. Authentication is required.

Request format

Accepts an updated group object containing all the keys that were received from the API initially. The only updatable field is `role_ids`. An empty roles array indicates a desire to remove the group from all the roles it was directly assigned to. Any other changed values are ignored.

For example:

```
{ "id": "75370a30-588a-11e4-8ed6-0800200c9a66",
  "login": "chinchilla",
  "display_name": "Chinchillas",
  **"role_ids": [2, 1]**,
  "is_group" : true,
  "is_remote" : true,
  "is_superuser" : false,
  "user_ids":
    [ "1cadd0e0-5887-11e4-8ed6-0800200c9a66", "5c1ab4b0-588b-11e4-8ed6-0800200c9a66" ] }
```

Response format

If the operation is successful, a 200 OK response including a group object with updated roles is returned.

DELETE /groups/<sid>

Deletes the user group with the specified ID (sid) from RBAC without making any changes to the directory service. Authentication required.

Response format

If the user group is successfully deleted, a 204 No Content with an empty body is returned.

Error responses

If the current user does not have the `user_groups:delete` permission for this user group, a 403 Forbidden response is returned.

If a group with the provided identifier does not exist, a 404 Not Found response is returned.

User roles endpoints

By assigning roles to users, you can manage them in sets that are granted access permissions to various PE objects. This makes tracking user access more organized and easier to manage. The `roles` endpoints enable you to get lists of roles and create new roles.

User role keys

The following keys are used with the RBAC v1 API's roles endpoints.

Key	Explanation	Example
<code>id</code>	An integer identifying the role.	18

Key	Explanation	Example
display_name	The role's name as a string.	"Viewers"
description	A string describing the role's function.	"View-only permissions"
permissions	An array containing permission objects that indicate what permissions a role grants. An empty array is valid. See Permission keys for more information.	[]
user_ids	An array of UUIDs indicating which users and groups are directly assigned to the role. An empty array is valid.	["fc115750-555a-11e4-916c-0800200c9a66", ...]
group_ids		

GET /roles

Fetches all roles with user and group ID lists and permission lists. Authentication is required.

Response format

Returns a JSON object containing all roles with user and group ID lists and permission lists.

For example:

```
[{"id": 123,
  "permissions": [{"object_type": "node_groups",
                    "action": "edit_rules",
                    "instance": "*"}, ...],
  "user_ids": [
    "1cadd0e0-5887-11e4-8ed6-0800200c9a66", "5c1ab4b0-588b-11e4-8ed6-0800200c9a66"],
  "group_ids": [ "2ca57e30-5887-11e4-8ed6-0800200c9a66" ],
  "display_name": "A role",
  "description": "Edit node group rules"},
...]
```

GET /roles/<rid>

Fetches a single role by its ID (rid). Authentication is required.

Response format

Returns a 200 OK response with the role object with a full list of permissions and user and group IDs.

For example:

```
{"id": 123,
  "permissions": [{"object_type": "node_groups",
                    "action": "edit_rules",
                    "instance": "*"}, ...],
  "user_ids": [
    "1cadd0e0-5887-11e4-8ed6-0800200c9a66", "5c1ab4b0-588b-11e4-8ed6-0800200c9a66"],
  "group_ids": [ "2ca57e30-5887-11e4-8ed6-0800200c9a66" ],
  "display_name": "A role",
  "description": "Edit node group rules"}
```

POST /roles

Creates a role, and attaches to it the specified permissions and the specified users and groups. Authentication is required.

Request format

Accepts a new role object. Any of the arrays can be empty and "description" can be null.

For example:

```
{ "permissions": [{ "object_type": "node_groups",
                    "action": "edit_rules",
                    "instance": "*" }, ... ],
  "user_ids":
  [ "1cadd0e0-5887-11e4-8ed6-0800200c9a66", "5c1ab4b0-588b-11e4-8ed6-0800200c9a66" ],
  "group_ids": [ "2ca57e30-5887-11e4-8ed6-0800200c9a66" ],
  "display_name": "A role",
  "description": "Edit node group rules" }
```

Response format

Returns a 201 Created response with a location header pointing to the new resource.

Error responses

Returns a 409 Conflict response if the role has a name that collides with an existing role.

PUT /roles/<rid>

Replaces a role at the specified ID (rid) with a new role object. Authentication is required.

Request format

Accepts the modified role object.

For example:

```
{ "id": 123,
  "permissions": [{ "object_type": "node_groups",
                    "action": "edit_rules",
                    "instance": "*" }, ... ],
  "user_ids":
  [ "1cadd0e0-5887-11e4-8ed6-0800200c9a66", "5c1ab4b0-588b-11e4-8ed6-0800200c9a66" ],
  "group_ids": [ "2ca57e30-5887-11e4-8ed6-0800200c9a66" ],
  "display_name": "A role",
  "description": "Edit node group rules" }
```

Response format

Returns a 200 OK response with the modified role object.

Error responses

Returns a 409 Conflict response if the new role has a name that collides with an existing role.

DELETE /roles/<rid>

Deletes the role identified by the role ID (rid). Users with this role immediately lose the role and all permissions granted by it, but their session is otherwise unaffected. Access to the next request that the user makes is determined by the new set of permissions the user has without this role.

Response format

Returns a 200 OK response if the role identified by <rid> has been deleted.

Error responses

Returns a 404 Not Found response if no role exists for <rid>.

Returns a 403 Forbidden response if the current user lacks permission to delete the role identified by <rid>.

Permissions endpoints

You assign permissions to user roles to manage user access to objects. The permissions endpoints enable you to get information about available objects and the permissions that can be constructed for those objects.

Permissions keys

The following keys are used with the RBAC v1 API's permissions endpoints. The available values for these keys are available from the /types endpoint (see below).

Key	Explanation	Example
object_type	A string identifying the type of object this permission applies to.	"node_groups"
action	A string indicating the type of action this permission permits.	"modify_children"
instance	A string containing the primary ID of the object instance this permission applies to, or "*" indicating that it applies to all instances. If the given action does not allow instance specification, "*" should always be used.	"cec7e830-555b-11e4-916c-0800200c9a" or "*" "

GET /types

Lists the objects that integrate with RBAC and demonstrates the permissions that can be constructed by picking the appropriate object_type, action, and instance triple. Authentication is required.

The has_instances flag indicates that the action permission is instance-specific if true, or false if this action permission does not require instance specification.

Response format

Returns a 200 OK response with a listing of types.

For example:

```
[{ "object_type": "node_groups",
  "display_name": "Node Groups",
  "description": "Groups that nodes can be assigned to.",
  "actions": [{ "name": "view",
    "display_name": "View",
    "description": "View the node groups",
    "has_instances": true
  }, {
    "name": "modify",
```

```

        "display_name": "Configure",
        "description": "Modify description, variables and classes",
        "has_instances": true
    }, ...]
}, ...]

```

Error responses

Returns a 401 Unauthorized response if no user is logged in.

Returns a 403 Forbidden response if the current user lacks permissions to view the types.

POST /permitted

Checks an array of permissions for the subject identified by the submitted identifier.

Request format

This endpoint takes a "token" in the form of a user or a user group's UUID and a list of permissions. This returns true or false for each permission queried, representing whether the subject is permitted to take the given action.

The full evaluation of permissions is taken into account, including inherited roles and matching general permissions against more specific queries. For example, a query for `users:edit:1` returns true if the subject has `users:edit:1` or `users:edit:*`.

In the following example, the first permission is querying whether the subject specified by the token is permitted to perform the `edit_rules` action on the instance of `node_groups` identified by the ID 4. Note that in reality, node groups and users use UUIDs as their IDs.

```

{
  "token": "<subject uuid>",
  "permissions": [
    {
      "object_type": "node_groups",
      "action": "edit_rules",
      "instance": "4"
    },
    {
      "object_type": "users",
      "action": "disable",
      "instance": "1"
    }
  ]
}

```

Response format

Returns a 200 OK response with an array of Boolean values representing whether each submitted action on a specific object type and instance is permitted for the subject. The array always has the same length as the submitted array and each returned Boolean value corresponds to the submitted permission query at the same index.

The example response below was returned from the example request in the previous section. This return means the subject is permitted `node_groups:edit_rules:4` but not permitted `users:disable:1`.

```
[true, false]
```

Token endpoints

A user's access to PE services can be controlled using authentication tokens. Users can generate their own authentication tokens using the `token` endpoint.

Related information

[Token-based authentication](#) on page 242

Puppet Enterprise users generate tokens to authenticate their access to certain PE command-line tools and API endpoints. Authentication tokens are tied to the permissions granted to the user through RBAC, and provide the user with the appropriate access to HTTP requests.

Token keys

The following keys are used with the `token` endpoint.

Key	Explanation
<code>login</code>	The user's login for the PE console (required).
<code>password</code>	The user's password for the PE console (required).
<code>lifetime</code>	The length of time the token is active before expiration (optional).
<code>description</code>	Additional metadata about the requested token (optional).
<code>client</code>	Additional metadata about the client making the token request (optional).
<code>label</code>	A user-defined label for the token (optional).

The lifetime key

When setting a token's lifetime, specify a numeric value followed by `y` (years), `d` (days), `h` (hours), `m` (minutes), or `s` (seconds). For example, a value of `12h` is 12 hours. Do not add a space between the numeric value and the unit of measurement. If you do not specify a unit, it is assumed to be seconds. If you do not want the token to expire, set the lifetime to 0. Setting it to zero gives the token a lifetime of approximately 10 years.

Tip: The default lifetime for all tokens is also configurable. See [Change the token's default lifetime](#) for configuration instructions.

The label key

You can choose to select a label for the token that can be used with other RBAC token endpoints. Labels:

- Must be no longer than 200 characters.
- Must not contain commas.
- Must contain something other than whitespace. (Whitespace is trimmed from the beginning and end of the label, though it is allowed elsewhere.)
- Must not be the same as a label for another token for the same user.

Token labels are assigned on a per-user basis: two users can each have a token labelled `my_token`, but a single user cannot have two tokens both labelled `my_token`. You cannot use labels to refer to other users' tokens.

POST /auth/token

Generates an access token for the user whose login information is POSTed. This token can then be used to authenticate requests to PE services using either the `X-Authentication` header or the `token` query parameter.

This route is intended to require zero authentication. While HTTPS is still required (unless PE is explicitly configured to permit HTTP), neither a whitelisted cert nor a session cookie is needed to POST to this endpoint.

Request format

Accepts a JSON object or curl command with the user's login and password information. The token's lifetime, a user-specified label, and additional metadata can be added, but are not required.

An example JSON request:

```
{ "login": "jeanjackson@example.com",
```

```
"password": "1234",
"lifetime": "4m",
"label": "personal workstation token"}
```

An example curl command request:

```
curl -cacert $(puppet config print cacert) -X POST -H 'Content-Type:
application/json' -d '{"login": "<LOGIN>", "password": "<PASSWORD>",
"lifetime": "4h", "label": "four-hour token"}' https://<HOSTNAME>:4433/
rbac-api/v1/auth/token
```

The various parts of this curl command request are explained as follows:

- `-cacert [FILE]`: Specifies a CA certificate as described in [Forming requests for the node classifier](#). Alternatively, you could use the `-k` flag to turn off SSL verification of the RBAC server so that you can use the HTTPS protocol without providing a CA cert. If you do not provide one of these options in your cURL request, cURL complains about not being able to verify the RBAC server.
- **Note:** The `-k` flag is shown as an example only. You should use your own discretion when choosing the appropriate server verification method for the tool that you are using.
- `-X POST`: This is an HTTP POST request to provide your login information to the RBAC service.
- `-H 'Content-Type: application/json'`: sets the `Content-Type` header to `application/json`, which indicates to RBAC that the data being sent is in JSON format.
- `-d '{"login": "<YOUR PE USER NAME>", "password": "<YOUR PE PASSWORD>", "lifetime": "4m", "label": "four-minute token"}'`: Provide the user name and password that you use to log in to the PE console. Optionally, set the token's lifetime and label.
- `https://<HOSTNAME>:<PORT>/rbac-api/v1/auth/token`: Sends the request to the token endpoint. For `HOSTNAME`, provide the FQDN of the server that is hosting the PE console service. If you are making the call from the console server, you can use "localhost." For `PORT`, provide the port that the PE services (node classifier service, RBAC service, and activity service) listen on. The default port is 4433.

Response format

Returns a 200 OK response if the credentials are good and the user is not revoked, along with a token.

For example:

```
{ "token": "asd0u0=2jdi jasodj-
w0duwdhjashd,kjsahdasoi0d9hw0hduashd0a9wdy0whdkaudhaksdhc9chakdh92..." }
```

Error responses

Returns a 401 Unauthenticated response if the credentials are bad or the user is revoked.

Returns a 400 Malformed response if something is wrong with the request body.

Directory service endpoints

Use the `ds` (directory service) API endpoints to get information about the directory service, test your directory service connection, and replace directory service connection settings.

To connect to the directory service anonymously, specify `null` for the lookup user and lookup password or leave these fields blank.

Related information

[External directory settings](#) on page 237

The table below provides examples of the settings used to connect to an Active Directory service and an OpenLDAP service to PE. Each setting is explained in more detail below the table.

GET /ds

Get the connected directory service information. Authentication is required.

Return format

Returns a 200 OK response with an object representing the connection.

For example:

```
{ "display_name": "AD",
  "hostname": "ds.foobar.com",
  "port": 10379, ... }
```

If the connection settings have not been specified, returns a 200 OK response with an empty JSON object.

For example:

```
{ }
```

GET /ds/test

Runs a connection test for the connected directory service. Authentication is required.

Return format

If the connection test is successful, returns a 200 OK response with information about the test run.

For example:

```
{ "elapsed": 10 }
```

Error responses

- 400 Bad Request if the request is malformed.
- 401 Unauthorized if no user is logged in.
- 403 Forbidden if the current user lacks the permissions to test the directory settings.

```
{ "elapsed": 20, "error": "..."} }
```

PUT /ds/test

Performs a connection test with the submitted settings. Authentication is required.

Request format

Accepts the full set of directory settings keys with values defined.

Return format

If the connection test is successful, returns information about the test run.

For example:

```
{ "elapsed": 10 }
```

Error responses

If the request times out or encounters an error, returns information about the test run.

For example:

```
{"elapsed": 20, "error": "..."}

```

PUT /ds

Replaces current directory service connection settings. Authentication is required.

When changing directory service settings, you must specify all of the required directory service settings in the PUT request, including the required settings that are remaining the same. You do not need to specify optional settings unless you are changing them.

Request format

Accepts directory service connection settings. To "disconnect" the DS, PUT either an empty object ("{}") or the settings structure with all values set to null.

For example:

```
{ "hostname": "ds.somehost.com",
  "name"
  "port": 10389,
  "login": "frances", ... }

```

Working with nested groups

When authorizing users, the RBAC service has the ability to search nested groups. Nested groups are groups that are members of external directory groups. For example, if your external directory has a "System Administrators" group, and you've given that group the "Superusers" user role in RBAC, then RBAC also searches any groups that are members of the "System Administrators" group and assign the "Superusers" role to users in those member groups.

By default, RBAC does not search nested groups. To enable nested group searches, specify the `search_nested_groups` field and give it a value of `true`.

Note: In PE 2015.3 and earlier versions, RBAC's default was to search nested groups. When **upgrading** from one of these earlier versions, this default setting is preserved and RBAC continues to search nested groups. If you have a large number of nested groups, you might experience a slowdown in performance when users are logging in because RBAC is searching the entire hierarchy of groups. To avoid performance issues, change the `search_nested_groups` field to `false` for a more shallow search in which RBAC searches only the groups it has been configured to use for user roles.

Using StartTLS connections

To use a StartTLS connection, specify `"start_tls": true`. When set to `true`, StartTLS is used to secure your connection to the directory service, and any certificates that you have configured through the DS trust chain setting is used to verify the identity of the directory service. When specifying StartTLS, make sure that you don't also have SSL set to `true`.

Disabling matching rule in chain

When PE detects an Active Directory that supports the `LDAP_MATCHING_RULE_IN_CHAIN` feature, it automatically uses it. Under some specific circumstances, you might need to disable this setting. To disable it, specify `"disable_ldap_matching_rule_in_chain": true` in the PUT request. This is an optional setting.

Return format

Returns a 200 OK response with an object showing the updated connection settings.

For example:

```
{ "hostname": "ds.somehost.com",

```



```
"port": 10389,
"login": "frances", ...}
```

Password endpoints

When local users forget passwords or lock themselves out of PE by attempting to log in with incorrect credentials too many times, you must generate a password reset token for them. The password endpoints enable you to generate password reset tokens for a specific local user or with a token that contains a temporary password in the body.

Tip: The PE console admin password can also be reset using a password reset script available on the PE console node.

Tip: 10 is the default number of login attempts that can be made with incorrect credentials before a user is locked out. You can change the value by configuring the `failed-attempts-lockout` parameter.

Related information

[Reset the console administrator password](#) on page 228

To reset the administrator password for console access, use the `puppet infrastructure` command.

POST /users/:sid/password/reset

Generates a single-use password reset token for the specified local user.

The generated token can be used to reset the password only one time and it has a limited lifetime. The lifetime is based on a configuration value `puppet_enterprise::profile::console::rbac_password_reset_expiration` (number of hours). The default value is 24 hours. Authentication is required.

This token is to be given to the appropriate user for use with the `/auth/reset` endpoint.

Response format

Returns a 201 Created response. For example:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJhcGlfdXNlciIsInNlYiI6ImE3YzA4MTY3LWE1MDUwLWp0aUc0dU95J9NGEAPtTwsDcux0_33hPz03m0Rp3hDSjblQwyTBTuf2rpgymaYo1VRw-Eou60dUiqBAAqkQ9rMtOtRtO6YTnQ8M6p9RxiFk_u5YjZDrhLZSqYxg-LY-cY2IFow_XFIwc9vWBuDX1LwLB9TqQJoI2NVNBymoHABzoKrnV-dKGFRawF-gBTwdtvs-oDFSx1kDBwkangGq2jb2Ghszhh8jdK9wWQ_8fnOcUk_kL1bE7Wr0htK045tT9BPazwZaZWagHiojI_YFyJfBiB_cF3XquRE-4C2EbejQld_u-WZGnJpZV_HGK4Spd56V-anuDq1AakaW3IzDJkJuPI4CxUE8_xpfFFLRdofTHrLPpwo5dszbKK-9xw3JfIMhnsK4e2H_5Nywk0w951vonKuY
```

Error responses

Returns a 403 Forbidden response if the user does not have permissions to create a reset path for the specified user, or if the user is a remote user.

Returns a 404 Not Found response if a user with the given identifier does not exist.

POST /auth/reset

Resets a local user's password using a one-time token obtained via the `/users/:sid/password/reset` endpoint, with the new password in the body. No authentication is required to use this endpoint.

Request format

The appropriate user is identified in the payload of the token. This endpoint does not establish a valid logged-in session for the user.

For example:

```
{"token": "text of token goes here",
"password": "someotherpassword"}
```

Response format

Returns a 200 OK response if the token is valid and the password has been successfully changed.

Error responses

Returns a 403 Permission Denied response if the token has already been used or is invalid.

PUT /users/current/password

Changes the password for the current local user. A payload containing the current password must be provided. Authentication is required.

Request format

The current and new passwords must both be included.

For example:

```
{ "current_password": "somepassword",
  "password": "someotherpassword" }
```

Response format

Returns a 200 OK response if the password has been successfully changed.

Error responses

Returns a 403 Forbidden response if the user is a remote user, or if `current_password` doesn't match the current password stored for the user. The body of the response includes a message that specifies the cause of the failure.

RBAC service errors

You're likely to encounter some errors when using the RBAC API. You'll want to familiarize yourself with the error response descriptions and the general error responses.

Error response format

When the client specifies an `accept` header in the request with type `application/json`, the RBAC service returns errors in a standard format.

Each response is an object containing the following keys:

Key	Definition
<code>kind</code>	A string classifying the error. It should be the same for all errors that have the same type of information in their <code>details</code> key.
<code>msg</code>	A human-readable message describing the error.
<code>details</code>	Additional machine-readable information about the error condition. The format of this key's value varies between kinds of errors, but is the same for each kind of error.

When returning errors in `text/html`, the body is the contents of the `msg` field.

General error responses

Any endpoint accepting a JSON body can return several kinds of 400 Bad Request responses.

Response	Status	Description
malformed-request	400	The submitted data is not valid JSON. The <code>details</code> key consists of one field, <code>error</code> , which contains the error message from the JSON parser.
schema-violation	400	<p>The submitted data has an unexpected structure, such as invalid fields or missing required fields. The <code>msg</code> contains a description of the problem. The <code>details</code> are an object with three keys:</p> <ul style="list-style-type: none"> • <code>submitted</code>: The submitted data as it was seen during schema validation. • <code>schema</code>: The expected structure of the data. • <code>error</code>: A structured description of the error.
inconsistent-id	400	Data was submitted to an endpoint where the ID of the object is a part of the URL and the submitted data contains an <code>id</code> field with a different value. The <code>details</code> key consists of two fields, <code>url-id</code> and <code>body-id</code> , showing the IDs from both sources.
invalid-id-filter	400	A URL contains a filter on the ID with an invalid format. No details are given with this error.
invalid-uuid	400	An invalid UUID was submitted. No details are given with this error.
user-unauthenticated	401	An unauthenticated user attempted to access a route that requires authentication.
user-revoked	401	A user who has been revoked attempted to access a route that requires authentication.
api-user-login	401	A person attempted to log in as the <code>api_user</code> with a password (<code>api_user</code> does not support username/password authentication).

Response	Status	Description
remote-user-conflict	401	<p>A remote user who is not yet known to RBAC attempted to authenticate, but a local user with that login already exists.</p> <p>The solution is to change either the local user's login in RBAC, or to change the remote user's login, either by changing the <code>user_lookup_attr</code> in the DS settings or by changing the value in the directory service itself.</p>
permission-denied	403	A user attempted an action that they are not permitted to perform.
admin-user-immutable	403	A user attempted to edit metadata or associations belonging to the default roles ("Administrators", "Operators", "Code Deployers", or "Viewers") or default users ("admin" or "api_user") that they are not allowed to change.
admin-user-not-in-admin-role		
default-roles-immutable		
conflict	409	A value for a field that is supposed to be unique was submitted to the service and another object has that value. For example, when a user is created with the same login as an existing user.
invalid-associated-id	422	An object was submitted with a list of associated IDs (for example, <code>user_ids</code>) and one or more of those IDs does not correspond to an object of the correct type.
no-such-user-LDAP	422	An object was submitted with a list of associated IDs (for example, <code>user_ids</code>) and one or more of those IDs does not correspond to an object of the correct type.
no-such-group-LDAP		
non-unique-lookup-attr	422	A login was attempted but multiple users are found via LDAP for the given username. The directory service settings must use a <code>user_lookup_attr</code> that is guaranteed to be unique within the provided user's RDN.
server-error	500	Occurs when the server throws an unspecified exception. A message and stack trace should be available in the logs.

Configuration options

There are various configuration options for the RBAC service. Each section can exist in its own file or in separate files.

RBAC service configuration

You can configure the RBAC service's settings to specify the duration before inactive user accounts expire, adjust the length of user sessions, the number of times a user can attempt to log in, and the length of time a password reset token is valid. You can also define a whitelist of certificates.

These configuration parameters are not required, but when present must be under the `rbac` section, as in the following example:

```
rbac: {
  # Duration in days before an inactive account expires
  account-expiry-days: 1
  # Duration in minutes that idle user accounts are checked
  account-expiry-check-minutes: 60
  # Duration in hours that a password reset token is viable
  password-reset-expiration: 24
  # Duration in minutes that a session is viable
  session-timeout: 60
  failed-attempts-lockout: 10
}
```

account-expiry-days

This parameter is a positive integer that specifies the duration, in days, before an inactive user account expires. The default value is undefined. To activate this feature, add a value of 1 or greater.

If a non-superuser hasn't logged into the console during this specified period, their user status updates to revoked. After creating an account, if a non-superuser hasn't logged in to the console during the specified period, their user status updates to revoked.

Important: If the `account-expiry-days` parameter is not specified, or has a value of less than 1, the `account-expiry-check-minutes` parameter is ignored.

account-expiry-check-minutes

This parameter is a positive integer that specifies how often, in minutes, the application checks for idle user accounts. The default value is 60 minutes.

password-reset-expiration

When a user doesn't remember their current password, an administrator can generate a token for them to change their password. The duration, in hours, that this generated token is valid can be changed with the `password-reset-expiration` config parameter. The default value is 24.

session-timeout

This parameter is a positive integer that specifies how long a user's session should last, in minutes. This session is the same across node classifier, RBAC, and the console. The default value is 60.

failed-attempts-lockout

This parameter is a positive integer that specifies how many failed login attempts are allowed on an account before that account is revoked. The default value is 10.

Note: If you change this value, create a new file or Puppet resets back to 10 when it next runs. Create the file in an RBAC section of `/etc/puppetlabs/console-services/conf.d`.

certificate-whitelist

This parameter is a path for specifying the file that contains the names of hosts that are allowed to use RBAC APIs and other downstream component APIs, such as the Node Classifier and the Activity services. This configuration is for the users who want to script interaction with the RBAC service.

Users must connect to the RBAC service with a client certificate that has been specified in this `certificate-whitelist` file. A successful match of the client certificate and this certificate-whitelist allows access to the RBAC APIs as the `api_user`. By default, this user is an administrator and has all available permissions.

The certificate whitelist contains, at minimum, the certificate for the nodes PE is installed on.

RBAC database configuration

Credential information for the RBAC service is stored in a PostgreSQL database.

The configuration information for that database is found in the 'rbac-database' section of the config.

For example:

```
rbac-database: {
  classname: org.postgresql.Driver
  subprotocol: postgresql
  subname: "//<path-to-host>:5432/perbac"
  user: <username here>
  password: <password here>
}
```

classname

Used by the RBAC service for connecting to the database; this option should always be `org.postgresql.Driver`.

subprotocol

Used by the RBAC service for connecting to the database; this options should always be `postgresql`.

subname

JDBC connection path used by the RBAC service for connecting to the database. This should be set to the hostname and configured port of the PostgreSQL database. `perbac` is the database the RBAC service uses to store credentials.

user

This is the username the RBAC service should use to connect to the PostgreSQL database.

password

This is the password the RBAC service should use to connect to the PostgreSQL database.

RBAC API v2

The role-based access control (RBAC) service enables you to manage users, directory groups, and roles.

Related information

[API index](#) on page 33

APIs allow you to interact with Puppet and Puppet Enterprise (PE) applications from your own code or application integration hooks.

Tokens endpoints

A user's access to PE services can be controlled using authentication tokens. Users can revoke authentication tokens using the `tokens` endpoints.

Related information

[Token-based authentication](#) on page 242

Puppet Enterprise users generate tokens to authenticate their access to certain PE command-line tools and API endpoints. Authentication tokens are tied to the permissions granted to the user through RBAC, and provide the user with the appropriate access to HTTP requests.

DELETE /tokens

Use this endpoint to revoke one or more authentication tokens, ensuring the tokens can no longer be used with RBAC.

Parameters

The tokens must be specified using at least one of the following parameters:

Parameter	Value
<code>revoke_tokens</code>	A list of complete authentication tokens to be revoked.
<code>revoke_tokens_by_usernames</code>	A list of usernames whose tokens are to be revoked.
<code>revoke_tokens_by_labels</code>	A list of the labels of tokens to revoke (only tokens owned by the user making the request can be revoked in this manner).

You can supply parameters either as query parameters or as JSON-encoded in the body. If you include them as query parameters, separate the tokens, labels, or usernames by commas:

```
/tokens?revoke_tokens_by_usernames=<USER NAME>,<USER NAME>
```

If you encoded them in the body, supply them as a JSON array:

```
{ "revoke_tokens_by_usernames": [ "<USER NAME>", "<USER NAME>" ] }
```

If you provide a parameter as both a query parameter and in the JSON body, the values from the two sources are combined. It is not an error to specify the same token using multiple means (such as by providing the entire token to the `revoke_tokens` parameter and also including its label in the value of `revoke_tokens_by_labels`).

In the case of an error, malformed or otherwise input, or bad request data, the endpoint still attempts to revoke as many tokens as possible. This means it's possible to encounter multiple error conditions in a single request, such as if there were malformed usernames supplied in the request *and* a database error occurred when trying to revoke the well-formed usernames.

All operations on this endpoint are idempotent—it is not an error to revoke the same token two or more times.

Any user can revoke any token by supplying the complete token in the `revoke_tokens` query parameter or request body field.

To revoke tokens by username, the user making the request must have the "Users Revoke" permission for that user.

Example JSON body

The following is an example JSON body using each of the available parameters.

```
{ "revoke_tokens" :
  [ "eyJhbGciOiJSUzUxMiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJhZG1pbilIsIm1hdCI6MTQzOTQ5Mzg0NiwiZXh0IjoiMGtAAIp_vBONSQQ2zU5ihWfsDU4mmmfbelRlCzA2TWrsV62DpJhtSc3iMLsSyjcUjsP6I87kjw3XftjcB79kx3aODfSvzf9l35IE7vi97s-9fFcHUFpZboyz60GDyRJVS0whTWpNJMlAfx0UBitggqCYXnE4rr8wKBoeOXoeEQezJ7u8-q0TqeN3ke4azDxdIqfhZ7H10-jDR0C5yeSBGWfX-0KEbp42cGz8lA6rrIHpsaajRWUg9yTheUkT2crh6878orCLgfoBLDh-roBTLeIual6sash-ggpdHqVktFOomEXM6UTJlp1NpuP01rNr9JmLxWhI8WpExH1l_-136D1NJm32kwo-oV6GzXR70xq_N2CwIwObw-XlS5aUUC4KkyPtDmNvnvCln4" ]
  "revoke_tokens_by_labels": [ "Workstation Token", "VPS Token" ],
  "revoke_tokens_by_usernames": [ "<USER NAME>", "<USER NAME>" ] }
```

Response codes

The server uses the following response codes:

Code	Definition
204 No Content	Sent if all operations are successful.
500 Application Error	Sent if there was a database error when trying to revoke tokens.
403	Sent if the user lacks the permission to revoke one of the supplied usernames and no database error occurred.
400 Malformed	<p>Sent if one these conditions are true:</p> <ul style="list-style-type: none"> At least one of the tokens, usernames, or labels is malformed. At least one of the usernames does not exist in the RBAC database. Neither <code>revoke_tokens</code>, <code>revoke_tokens_by_usernames</code>, nor <code>revoke_tokens_by_labels</code> is supplied. There are unrecognized query parameters or fields in the request body, and the user has all necessary permissions and no database error occurred.

All error responses follow the standard JSON error format, meaning they have `kind`, `msg`, and `details` keys.

The `kind` key is `puppetlabs.rbac/database-token-error` if the response is a 500, `permission-denied` if the response is a 403, and `malformed-token-request` if the response is a 400.

The `msg` key contains an English-language description of the problems encountered while processing the request and performing the revocations, ending with either "No tokens were revoked" or "All other tokens were successfully revoked", depending on whether any operations were successful.

The `details` key contains an object with arrays in the `malformed_tokens`, `malformed_usernames`, `malformed_labels`, `nonexistent_usernames`, `permission_denied_usernames`, and `unrecognized_parameters` fields, as well as the Boolean field `other_tokens_revoked`.

The arrays all contain bad input from the request, and the `other_tokens_revoked` field's value indicates whether any of the revocations specified in the request were successful or not.

Example error body

The server returns an error body resembling the following:

```
{
  "kind": "malformed-request",
  "msg": "The following user does not exist: FormerEmployee. All other tokens were successfully revoked.",
  "details": {
    "malformed_tokens": [],
    "malformed_labels": [],
    "malformed_usernames": [],
    "nonexistent_usernames": ["FormerEmployee"],
    "permission_denied_usernames": [],
    "unrecognized_parameters": [],
    "other_tokens_revoked": true
  }
}
```

DELETE /tokens/<token>

Use this endpoint to revoke a single token, ensuring that it can no longer be used with RBAC. Authentication is required.

This endpoint is equivalent to `DELETE /tokens?revoke_tokens={TOKEN}`.

This API can be used only by the admin or API user.

Response codes

The server uses the following response codes:

Code	Definition
204 No Content	Sent if all operations are successful.
400 Malformed	Sent if the token is malformed.

The error response is identical to `DELETE /tokens`.

Example error body

The error response from this endpoint is identical to `DELETE /tokens`.

```
{
  "kind": "malformed-request",
  "msg": "The following token is malformed: notAToken. This can be caused by an error while copying and pasting. No tokens were revoked.",
  "details": {
    "malformed_tokens": ["notAToken"],
    "malformed_labels": [],
    "malformed_usernames": [],
    "nonexistent_usernames": [],
    "permission_denied_usernames": [],
    "unrecognized_parameters": [],
    "other_tokens_revoked": false
  }
}
```

POST /auth/token/authenticate

Use this endpoint to exchange a token for a map representing an RBAC subject and associated token data. This endpoint does not require authentication.

This endpoint accepts a JSON body containing entries for `token` and `update_last_activity?`. The `token` field is a string containing an authentication token. The `update_last_activity?` field is a Boolean data type that indicates whether a successful authentication should update the `last_active` timestamp for the token.

Response codes

The server uses the following response codes:

Code	Definition
200 OK	The subject represented by the token.
400 invalid-token	The provided token was either tampered with or could not be parsed.
403 token-revoked	The provided token has been revoked.
403 token-expired	The token has expired and is no longer valid.
403 token-timed-out	The token has timed out due to inactivity.

Example JSON body

```
{
  "token": "0VZZ6geJQK8zJGKxBdqlatTsMLAsfCQFJuRwxSMNgCr4",
  "update_last_activity?": false
}
```

Example return

```
{
  "description": null,
  "creation": "YYYY-MM-DDT22:24:30Z",
  "email": "franz@kafka.com",
  "is_revoked": false,
  "last_active": "YYYY-MM-DDT22:24:31Z",
  "last_login": "YYYY-MM-DDT22:24:31.340Z",
  "expiration": "YYYY-MM-DDT22:29:30Z",
  "is_remote": false,
  "client": null,
  "login": "franz@kafka.com",
  "is_superuser": false,
  "label": null,
  "id": "c84bae61-f668-4a18-9a4a-5e33a97b716c",
  "role_ids": [1, 2, 3],
  "user_id": "c84bae61-f668-4a18-9a4a-5e33a97b716c",
  "timeout": null,
  "display_name": "Franz Kafka",
  "is_group": false
}
```

Activity service API

The activity service logs changes to role-based access control (RBAC) entities, such as users, directory groups, and user roles.

Related information

[API index](#) on page 33

APIs allow you to interact with Puppet and Puppet Enterprise (PE) applications from your own code or application integration hooks.

Forming activity service API requests

Token-based authentication is required to access the activity service API. You can authenticate requests with user authentication tokens or whitelisted certificates.

By default, the activity service listens on port 4433. All endpoints are relative to the `/activity-api/` path. So, for example, the full URL for the `/v1/events` endpoint on localhost is `https://localhost:4433/activity-api/v1/events`.

Authentication using tokens

Insert a user authentication token variable in an activity service API request.

1. Generate a token: `puppet-access login`
2. Print the token and copy it: `puppet-access show`
3. Save the token as an environment variable: `export TOKEN=<PASTE THE TOKEN HERE>`
4. Include the token variable in your API request:

```
curl -k -X GET https://<HOSTNAME>:<PORT>/activity-api/v1/events?
service_id=classifier -H "X-Authentication:$TOKEN"
```

Authentication using whitelisted certificate

You can also authenticate requests using a certificate listed in RBAC's certificate whitelist, located at `/etc/puppetlabs/console-services/rbac-certificate-whitelist`. Note that if you edit this file, you must reload the `pe-console-services` service (run `sudo service pe-console-services reload`) for your changes to take effect.

Attach the certificate using the command line, as demonstrated in the example curl query below. You must have the whitelisted certificate name (which must match a name in the `/etc/puppetlabs/console-services/rbac-certificate-whitelist` file) and the private key to run the script.

```
curl -X GET https://<HOSTNAME>:<PORT>/activity-api/v1/events?
service_id=classifier \
--cert /etc/puppetlabs/puppet/ssl/certs/<WHITELISTED CERTNAME>.pem \
--key /etc/puppetlabs/puppet/ssl/private_keys/<WHITELISTED CERTNAME>.pem \
--cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem
```

You do not need to use an agent certificate for authentication. You can use `puppet cert generate` to create a new certificate specifically for use with the API.

Related information

[Token-based authentication](#) on page 242

Puppet Enterprise users generate tokens to authenticate their access to certain PE command-line tools and API endpoints. Authentication tokens are tied to the permissions granted to the user through RBAC, and provide the user with the appropriate access to HTTP requests.

Event types reported by the activity service

Activity reporting provides a useful audit trail for actions that change role-based access control (RBAC) entities, such as users, directory groups, and user roles.

Local users

These events are displayed in the console on the **Activity** tab for the affected user.

Event	Description	Example
Creation	A new local user is created. An initial value for each metadata field is reported.	Created with login set to "jean".
Metadata	Any change to the login, display name, or email keys.	Display name set to "Jean Jackson".
Role membership	A user is added to or removed from a role. The display name and user ID of the affected user are displayed.	User Jean Jackson (973c0cee-5ed3-11e4-aa15-123b93f75cba) added to role Operators.

Event	Description	Example
Authentication	A user logs in. The display name and user ID of the affected user are displayed.	User Jean Jackson (973c0cee-5ed3-11e4-aa15-123b93f75cba) logged in.
Password reset token	A token is generated for a user to use when resetting their password. The display name and user ID of the affected user are shown.	A password reset token was generated for user Jean Jackson (973c0cee-5ed3-11e4-aa15-123b93f75cba).
Password changed	A user successfully changes their password with a token.	Password reset for user Jean Jackson (973c0cee-5ed3-11e4-aa15-123b93f75cba).
Revocation	A user is revoked or reinstated.	User revoked.

Remote users

These events are displayed in the console on the **Activity** tab for the affected user.

Event	Description	Example
Role membership	A user is added to or removed from a role. These events are also shown on the page for the role. The display name and user ID of the affected user are displayed.	User Kalo Hill (76483e62-5ed4-11e4-aa15-123b93f75cba) added to role Viewers.
Revocation	A user is revoked or reinstated.	User revoked.

Directory groups

These events are displayed in the console on the **Activity** tab for the affected group.

Event	Description	Example
Importation	A directory group is imported. The initial value for each metadata field is reported (these cannot be updated using the RBAC UI).	Created with display name set to "Engineers".
Role membership	A group is added to or removed from a role. These events are also shown on the page for the role. The group's display name and ID are provided.	Group Engineers (7dee3acc-5ed4-11e4-aa15-123b93f75cba) added to role Operators.

Roles

These events are displayed in the console on the **Activity** tab for the affected role.

Event	Description	Example
Metadata	A role's display name or description changes.	Description set to "Sysadmins with full privileges for node groups."

Event	Description	Example
Members	A group is added to or removed from a role. The display name and ID of the user or group are provided. These events are also displayed on the page for the affected user or group.	User Kalo Hill (76483e62-5ed4-11e4-aa15-123b93f75cba) removed from role Operators.
Permissions	A permission is added to or removed from a role.	Permission users:edit:76483e62-5ed4-11e4-aa15-123b93f75cba added to role Operators.
Delete	A role has been removed.	The Delete event is recorded and available only through the activity service API, not the Activity tab.

Orchestration

These events are displayed in the console on the **Activity** tab for the affected node.

Event	Description	Example
Agent runs	Puppet runs as part of an orchestration job. This includes runs started from the orchestrator or the PE console.	Request Puppet agent run on node.example.com via orchestrator job 12.
Task runs	Tasks run as orchestration jobs set up in the console or on the command line.	Request echo task on neptune.example.com via orchestrator job 9,607

Authentication tokens

These events are displayed in the console on the **Activity** tab on the affected user's page.

Event	Description	Example
Creation	A new token is generated. These events are exposed in the console on the Activity tab for the user who owns the token.	Amari Perez (c84bae61-f668-4a18-9a4a-5e33a97b716c) generated an authentication token.
Direct revocation	A successful token revocation request. These events are exposed in the console on the Activity tab for the user performing the revocation.	Administrator (42bf351c-f9ec-40af-84ad-e976fec7f4bd) revoked an authentication token belonging to Amari Perez (c84bae61-f668-4a18-9a4a-5e33a97b716c), issued at 2016-02-17T21:53:23.000Z and expiring at 2016-02-17T21:58:23.000Z.

Event	Description	Example
Revocation by username	All tokens for a username are revoked. These events are exposed in the console on the Activity tab for the user performing the revocation.	Administrator (42bf351c-f9ec-40af-84ad-e976fec7f4bd) revoked all authentication tokens belonging to Amari Perez (c84bae61-f668-4a18-9a4a-5e33a97b716c).

Directory service settings

These events are not exposed in the console. The activity service API must be used to see these events.

Event	Description	Example
Update settings (except password)	A setting is changed in the directory service settings.	User rdn set to "ou=users".
Update directory service password	The directory service password is changed.	Password updated.

Events endpoints

Use the `events` endpoints to retrieve activity service events.

GET /v1/events

Fetches activity service events. Web session authentication is required.

Request format

The `/v1/events` endpoint supports filtering through query parameters.

service_id

required The ID of the service: `classifier`, `code-manager`, `pe-console`, or `rbac`.

subject_type

The subject who performed the activity: `users`.

subject_id

Comma-separated list of IDs associated with the defined subject type.

object_type

The object affected by the activity: `node`, `node_groups`, `users`, `user_groups`, `roles`, `environment`, or `directory_server_settings`.

object_id

Comma-separated list of IDs associated with the defined object type.

offset

Number of commits to skip before returning events.

limit

Maximum number of events to return. Default is 1000 events.

Examples

```
curl -k -X GET "https://web5.mydomain.edu:4433/activity-api/v1/events?
service_id=classifier&subject_type=users&subject_id=6868e4af-2996-46c6-8e42-1ae873f8a0ba"
-H "X-Authentication:$TOKEN"
```

```
curl -k -X GET "https://web5.mydomain.edu:4433/activity-api/v1/events?
service_id=pe-
console&object_type=node&object_id=emerald-1.platform9.mydomain.edu" -H "X-
Authentication:$TOKEN"
```

Response format

Responses are returned in a structured JSON format. Example return:

```
{
  "commits": [ {
    "object": {
      "id": "b55c209d-e68f-4096-9a2c-5ae52dd2500c", "name":
"web_servers"
    },
    "subject": {
      "id": "6868e4af-2996-46c6-8e42-1ae873f8a0ba", "name":
"kai.evans"
    },
    "timestamp": "2019-03-05T18:52:27Z", "events": [ {
      "message": "Deleted the \"web_servers\" group with id b55c209d-
e68f-4096-9a2c-5ae52dd2500c"
    } ]
  } ], "offset": 0, "limit": 1, "total-rows": 5
}

{
  "commits": [ {
    "object": {
      "id": "emerald-1.platform9.mydomain.edu", "name":
"emerald-1.platform9.mydomain.edu"
    },
    "subject": {
      "id": "dlf4919a-cf65-4c26-8832-8c4b5302b58b", "name": "steve"
    },
    "timestamp": "2019-05-02T22:50:16Z", "events": [ {
      "message": "Scheduled job 3 request for facter_task task on
emerald-1.platform9.mydomain.edu run via orchestrator job 7,759"
    } ]
  } ], "offset": 0, "limit": 1, "total-rows": 2915
}
```

GET /v1/events.csv

Fetches activity service events and returns in a flat CSV format. Token-based authentication is required.

Request format

The `/v1/events.csv` endpoint supports the same parameters as `/v1/events`

Examples

```
curl -k -X GET "https://web5.mydomain.edu:4433/activity-api/v1/events.csv?
service_id=classifier&subject_type=users&subject_id=6868e4af-2996-46c6-8e42-1ae873f8a0ba
-H "X-Authentication:$TOKEN"
```

Response format

Responses are returned in a flat CSV format. Example return:

```
Submit Time,Subject Type,Subject Id,Subject Name,Object Type,Object
Id,Object Name,Type,What,Description,Message
YYYY-MM-DD
18:52:27.76,users,6868e4af-2996-46c6-8e42-1ae873f8a0ba,kai.evens,node_groups,b55c209d-
e68f-4096-9a2c-5ae52dd2500c,web_servers,delete,node_group,delete_node_group,"Deleted
the "web_servers" group with id b55c209d-e68f-4096-9a2c-5ae52dd2500c"
YYYY-MM-DD
18:52:02.391,users,6868e4af-2996-46c6-8e42-1ae873f8a0ba,kai.evens,node_groups,b55c209d-
e68f-4096-9a2c-5ae52dd2500c,web_servers,create,node_group,create_node_group,"Created
the "web_servers" group with id b55c209d-e68f-4096-9a2c-5ae52dd2500c"
YYYY-MM-DD
18:52:02.391,users,6868e4af-2996-46c6-8e42-1ae873f8a0ba,kai.evens,node_groups,b55c209d-
e68f-4096-9a2c-5ae52dd2500c,web_servers,edit,node_group_description,edit_node_group_des
the description to """"
YYYY-MM-DD
18:52:02.391,users,6868e4af-2996-46c6-8e42-1ae873f8a0ba,kai.evens,node_groups,b55c209d-
e68f-4096-9a2c-5ae52dd2500c,web_servers,edit,node_group_environment,edit_node_group_env
the environment to "production""
YYYY-MM-DD
18:52:02.391,users,6868e4af-2996-46c6-8e42-1ae873f8a0ba,kai.evens,node_groups,b55c209d-
e68f-4096-9a2c-5ae52dd2500c,web_servers,edit,node_group_environment_override,edit_node_g
the environment override setting to false
YYYY-MM-DD
18:52:02.391,users,6868e4af-2996-46c6-8e42-1ae873f8a0ba,kai.evens,node_groups,b55c209d-
e68f-4096-9a2c-5ae52dd2500c,web_servers,edit,node_group_parent,edit_node_group_parent,Ch
the parent to ec519937-8681-43d3-8b74-380d65736dba
YYYY-MM-DD
00:41:18.944,users,6868e4af-2996-46c6-8e42-1ae873f8a0ba,kai.evens,node_groups,ec519937-
Orchestrator,edit,node_group_class_parameter,delete_node_group_class_parameter_puppet_e
the "use_application_services" parameter from the
"puppet_enterprise::profile::orchestrator" class"
YYYY-MM-DD
00:41:10.631,users,6868e4af-2996-46c6-8e42-1ae873f8a0ba,kai.evens,node_groups,ec519937-
Orchestrator,edit,node_group_class_parameter,add_node_group_class_parameter_puppet_ent
the "use_application_services" parameter to the
"puppet_enterprise::profile::orchestrator" class"
YYYY-MM-DD
20:41:30.223,users,6868e4af-2996-46c6-8e42-1ae873f8a0ba,kai.evens,node_groups,46e34005-
bc48-4813221e9ffb,PE
Agent,schedule_deploy,node_group,schedule_puppet_agent_on_node_group,Schedule
puppet agent run on nodes in this group to be run at 2019-01-16T08:00:00Z
```

Inspecting your infrastructure

The Puppet Enterprise console offers a variety of tools you can use to monitor the current state of your infrastructure, see the results of planned or unplanned changes to your Puppet code, and investigate problems. These tools are grouped in the **Inspect** section of the console's sidebar.

Monitoring current infrastructure state

When nodes fetch their configurations from the Puppet master, they send back inventory data and a report of their run. This information is summarized on the **Overview** page in the console.

The **Overview** page displays the most recent run status of each of your nodes so you can quickly find issues and diagnose their causes. You can also use this page to gather essential information about your infrastructure at a glance, such as how many nodes your Puppet master is managing, and whether any nodes are unresponsive.

Node run statuses

The **Overview** page displays the run status of each node following the most recent Puppet run. There are 10 possible run statuses.

Nodes run in enforcement mode

❗ With failures

This node's last Puppet run failed, or Puppet encountered an error that prevented it from making changes.

The error is usually tied to a particular resource (such as a file) managed by Puppet on the node. The node as a whole might still be functioning normally. Alternatively, the problem might be caused by a situation on the Puppet master, preventing the node's agent from verifying whether the node is compliant.

🟡 With corrective changes

During the last Puppet run, Puppet found inconsistencies between the last applied catalog and this node's configuration, and corrected those inconsistencies to match the catalog.

Note: Corrective change reporting is available only on agent nodes running PE 2016.4 and later. Agents running earlier versions report all change events as "with intentional changes."

🟢 With intentional changes

During the last Puppet run, changes to the catalog were successfully applied to the node.

✅ Unchanged

This node's last Puppet run was successful, and it was fully compliant. No changes were necessary.

Nodes run in no-op mode

Note: No-op mode reporting is available only on agent nodes running PE 2016.4 and later. Agents running earlier versions report all no-op mode runs as "would be unchanged."

❗ With failures

This node's last Puppet run in no-op mode failed, or Puppet encountered an error that prevented it from simulating changes.

🟡 Would have corrective changes

During the last Puppet run, Puppet found inconsistencies between the last applied catalog and this node's configuration, and would have corrected those inconsistencies to match the catalog.

🟢 Would have intentional changes

During the last Puppet run, catalog changes would have been applied to the node.

✅ Would be unchanged

This node's last Puppet run was successful, and the node was fully compliant. No changes would have been necessary.

Nodes not reporting

Unresponsive

The node hasn't reported to the Puppet master recently. Something might be wrong. The cutoff for considering a node unresponsive defaults to one hour, and can be configured.

Check the run status table to see the timestamp for the last known Puppet run for the node and an indication of whether its last known run was in no-op mode. Correct the problem to resume Puppet runs on the node.

Have no reports

Although Puppet Server is aware of this node's existence, the node has never submitted a Puppet report for one or more of the following reasons: it's a newly commissioned node; it has never come online; or its copy of Puppet is not configured correctly.

Note: Expired or deactivated nodes are displayed on the Overview page for seven days. To extend the amount of time that you can view or search for these nodes, change the `node-ttl` setting in PuppetDB. Changing this setting affects resources and exported resources.

Special categories

In addition to reporting the run status of each node, the **Overview** page provides a secondary count of nodes that fall into special categories.

Intended catalog failed

During the last Puppet run, the intended catalog for this node failed, so Puppet substituted a cached catalog, as per your configuration settings.

This typically occurs when you have compilation errors in your Puppet code. Check the Puppet run's log for details.

This category is shown only if one or more agents fail to retrieve a valid catalog from Puppet Server.

Enforced resources found





During the last Puppet run in no-op mode, one or more resources was enforced, as per your use of the `noop => false` metaparameter setting.

This category is shown only if enforced resources are present on one or more nodes.

How Puppet determines node run statuses

Puppet uses a hierarchical system to determine a single run status for each node. This system gives higher priority to the activity types most likely to cause problems in your deployment, so you can focus on the nodes and events most in need of attention.

During a Puppet run, several activity types might occur on a single node. A node's run status reflects the activity with the highest alert level, regardless of how many events of each type took place during the run. Failure events receive the highest alert level, and no change events receive the lowest.

Run status	Definitely happened	Might also have happened
	Failure	Corrective change, intentional change, no change
	Corrective change	Intentional change, no change
	Intentional change	No change
	No change	

For example, during a Puppet run in enforcement mode, a node with 100 resources receives intentional changes on 30 resources, corrective changes on 10 resources, and no changes on the remaining 60 resources. This node's run status is "with corrective changes."

Node run statuses also prioritize run mode (either enforcement or no-op) over the state of individual resources. This means that a node run in no-op mode is always reported in the **Nodes run in no-op** column, even if some of its resource changes were enforced. Suppose the no-op flags on a node's resources are all set to false. Changes to the resources are enforced, not simulated. Even so, because it is run in no-op mode, the node's run status is "would have intentional changes."

Filtering nodes on the Overview page

You can filter the list of nodes displayed on the **Overview** page by run status and by node fact. If you set a run status filter, and also set a node fact filter, the table takes both filters into account, and shows only those nodes matching both filters.

Clicking **Remove filter** removes all filters currently in effect.

The filters you set are persistent. If you set run status or fact filters on the **Overview** page, they continue to be applied to the table until they're changed or removed, even if you navigate to other pages in the console or log out. The persistent storage is associated with the browser tab, not your user account, and is cleared when you close the tab.

Important: The filter results count and the fact filter matching nodes counts are cached for two minutes after first retrieval. This reduces the total load on PuppetDB and decreases page load time, especially for fact filters with multiple rows. As a result, the displayed counts might be up to two minutes out of date.

Filter by node run status

The status counts section at the top of the **Overview** page shows a summary of the number of nodes with each run status as of the last Puppet run. Filter nodes by run status to quickly focus on nodes with failures or change events.

In the status counts section, select a run status (such as **with corrective changes** or **have no reports**) or a run status category (such as **Nodes run in no-op**).

Filter by node fact

You can create a highly specific list of nodes for further investigation by using the fact filter tool.

For example, you can check that nodes you've updated have successfully changed, or find out the operating systems or IP addresses of a set of failed nodes to better understand the failure. You might also filter by facts to fulfill an auditor's request for information, such as the number of nodes running a particular version of software.

1. Click **Filter by fact value**. In the **Fact** field, select one of the available facts. An empty fact field is not allowed.

Tip: To see the facts and values reported by a node on its most recent run, click the node name in the **Run status** table, then select the node's **Facts** tab.

2. Select an Operator:

Operator	Meaning	Notes
=	is	
!=	is not	
~	matches a regular expression (regex)	Select this operator to use wildcards and other regular expressions if you want to find matching facts without having to specify the exact value.
!~	does not match a regular expression (regex)	
>	greater than	Can be used only with facts that have a numeric value.
>=	greater than or equal to	Can be used only with facts that have a numeric value.
<	less than	Can be used only with facts that have a numeric value.

Operator	Meaning	Notes
<=	less than or equal to	Can be used only with facts that have a numeric value.

3. In the **Value** field, enter a value. Strings are case-sensitive, so make sure you use the correct case.

The filter displays an error if you use an invalid string operator (for example, selecting a numeric value operator such as `>=` and entering a non-numeric string such as `pilsen` as the value) or enter an invalid regular expression.

Note: If you enter an invalid or empty value in the **Value** field, PE takes the following action in order to avoid a filter error:

- Invalid or empty Boolean facts are processed as **false**, and results are retrieved accordingly.
- Invalid or empty numeric facts are processed as **0**, and results are retrieved accordingly.
- Invalid or incomplete regular expressions invalidate the filter, and no results are retrieved.

4. Click **Add**.

5. As needed, repeat these steps to add additional filters. If filtering by more than one node fact, specify either **Nodes must match all rules** or **Nodes can match any rule**.

Filtering nodes in your node list

Filter your node list by node name or by PQL query to more easily inspect them.

Filter your node list by node name

Filter your nodes list by node name to inspect them as a group.

Select **Node name**, type in the word you want to filter by, and click **Submit**.

Filter your nodes by PQL query

Filter your nodes list using a common PQL query.

Filtering your nodes list by PQL query enables you to manage them by specific factors, such as by operating system, report status, or class.

Specify a target by doing one of the following:

- Enter a query that selects the target you want. See the [Puppet Query Language \(PQL\) reference](#) for more information.
- Click **Common queries**. Select one of the queries and replace the defaults in the braces (`{ }`) with values that specify the target you want.

Target	PQL query
All nodes	<code>nodes[certname] { }</code>
Nodes with a specific resource (example: httpd)	<code>resources[certname] { type = "Service" and title = "httpd" }</code>
Nodes with a specific fact and value (example: OS name is)	<code>inventory[certname] { facts.os.name = "<OS>" }</code>
Nodes with a specific report status (example: last run failed)	<code>reports[certname] { latest_report_status = "failed" }</code>
Nodes with a specific class (example:)	<code>resources[certname] { type = "Class" and title = "Apache" }</code>
Nodes assigned to a specific environment (example: production)	<code>nodes[certname] { catalog_environment = "production" }</code>

Target	PQL query
Nodes with a specific version of a resource type (example: OpenSSL is v1.1.0e)	<pre>resources[certname] {type = "Package" and title="openssl" and parameters.ensure = "1.0.1e-51.el7_2.7" }</pre>
Nodes with a specific resource and operating system (example: httpd and)	<pre>inventory[certname] { facts.operatingsystem = "CentOS" and resources { type = "Service" and title = "httpd" } }</pre>

Monitor PE services

Puppet Enterprise includes tools for monitoring the status of core services including the activity, classifier, and RBAC services, Puppet Server, and PuppetDB. You can monitor these services on the command line, and from within the console.

View the Puppet Services status monitor

The **Puppet Services status** monitor provides a visual overview of the current state of key services, and can be used to quickly determine whether an unresponsive or restarting service is causing an issue with your deployment.

1. In the console, click **Overview**.
2. Click **Puppet Services status** to open the monitor.

A checkmark appears next to **Puppet Services status** if all applicable services are accepting requests. In the event that no data is available, a question mark appears next to the link. If one or more services is restarting or not accepting requests, a warning icon appears.

puppet infrastructure status command

The `puppet infrastructure status` command displays errors and alerts from PE services, including the activity, classifier, and RBAC services, Puppet Server, and PuppetDB.

The command reports separately on the master and any compilers or replicas in your environment. You must run the command as root.

Exploring your catalog with the node graph

The node graph provides a graphic representation of any node's configuration, shows you the relationships between classes and resources, and lets you gain greater insight into just how the Puppet master compiles your code and ships it to an agent. Visualizing relationships between resources helps you organize them, creating more reliable deployments.

How the node graph can help you

The node graph displays a node's catalog (as of the last Puppet run) as an interactive visual map. The graph shows the desired state for each resource that PE manages, as well as each resource's status as of the last run, and helps you understand the dependencies between resources. It also allows you to visually identify complexity you might not need, and problematic dependencies that need your attention.

The node graph is an ideal tool for:

- Gaining greater insight into your deployment.
- Visualizing the relationship of resources on your nodes.
- Diagnosing dependency loops and viewing all impacted resources.
- Helping new users to understand relationships among classes and resources in their Puppet catalog.
- Understanding defined resource types as they are deployed across your configuration.
- Understanding the content of modules.

Investigate a change with the node graph

Use the node graph to view the details of a change made to one of your nodes, and see how other resources and classes might be impacted.

Access a node's graph

You can reach a node's graph from either the **Overview** page or the node's detail page. If you're using the node graph to find the cause of a particular run status, use the **Overview** page path. If you already know which node you're interested in, the **Nodes** page path is most efficient.

1. Locate a node you're interested in viewing:
 - On the **Overview** page, filter by run status or fact value to locate the node.
 - On the **Nodes** page, search for the node by name.
2. Find and click the **Node graph** link:
 - On the **Overview** page, the **Node graph** link is located to the right of the node's name.
 - On the **Nodes** page, click the node name. The **View node graph** link is located at the top of the node details page.

Focus on a change

When the node graph opens, you'll see the containment view, which presents the catalog in its entirety. This can be a bit overwhelming, but you can use filters to quickly focus on the change event you're interested in.

1. In the **Filter:** bar, select **status**.
2. Select a run status, such as **Corrective change**.

The **Node graph** details pane at the right of the screen opens and displays a list of resources that match your filter criteria.

3. Select any item on the list, and the graph repositions to focus on that resource.

The **Node graph** details pane now shows additional data on current event status, as well as the object's source file, code line number, relevant tags, and class containment information.

Investigate dependencies

If your resource is part of a dependency chain, a **Dependency view** link is shown in the **Node graph** details pane. This link does not appear for resources without any dependencies.

1. Click **Dependency view**.

The **Node graph** details pane displays a list of the resource's ancestor and descendant dependencies.

2. Hover over any item on the list, and the graph highlights that dependency path.

Note: In the event that a dependency cycle is detected, the node graph issues a cycle warning message.

3. Click **Exit dependency view** to return to the containment view.

Viewing and managing all packages in use

The **Packages** page in the console shows all packages in use across your infrastructure by name, version, and provider, as well as the number of instances of each package version in your infrastructure. Use the **Packages** page to quickly identify which nodes are impacted by packages you know are eligible for maintenance updates, security patches, and license renewals.

Package inventory reporting is available for all nodes with Puppet agent version 1.6.0 or later installed, including systems that are not actively managed by Puppet.

Tip: Packages are gathered from all available providers. The package data reported on the **Packages** page can also be obtained by using the `puppet resource` command to search for package.

Enable package data collection

Package data collection is disabled by default, so the Packages page in the console initially appears blank. In order to view a node's current package inventory, enable package data collection.

You can choose to collect package data on all your nodes, or just a subset. Any node with Puppet agent version 1.6.0 or later installed can report package data, including nodes that are not under active management with Puppet Enterprise.

1. In the console, click **Classification**.
 - If you want to collect package data on all your nodes, click the **PE Agent** node group.
 - If you want to collect package data on a subset of your nodes, click **Add group** and create a new classification node group. Select **PE Agent** as the group's parent name. After the new node group is set up, use the **Rules** tab to dynamically add the relevant nodes.
2. Click **Configuration**. In the **Add new class** field, select **puppet_enterprise::profile::agent** and click **Add class**.
3. In the **puppet_enterprise::profile::agent** class, set the **Parameter** to **package_inventory_enabled** and the **Value** to **true**. Click **Add parameter**, and commit changes.
4. Run Puppet to apply these changes to the nodes in your node group.

Puppet enables package inventory collection on this Puppet run, and begins collecting package data and reporting it on the **Packages** page on each subsequent Puppet run.
5. Run Puppet a second time to begin collecting package data, then click **Packages**.

View and manage package inventory

To view and manage the complete inventory of packages on your systems, use the Packages page in the console.

Before you begin

Make sure you have enabled package data collection for the nodes you wish to view.

Tip: If all the nodes on which a certain package is installed are deactivated, but the nodes' specified `node-purge-ttl` period has not yet elapsed, instances of the package still appear in summary counts on the **Packages** page. To correct this issue, adjust the `node-purge-ttl` setting and run garbage collection.

1. Run Puppet to collect the latest package data from your nodes.
2. In the console, click **Packages** to view your package inventory. To narrow the list of packages, enter the name or partial name of a package in the **Filter by package name** field and click **Apply**.
3. Click any package name or version to enter the detail page for that package.
4. On a package's detail page, use the **Version** selector to locate nodes with a particular package version installed.
5. Use the **Instances** selector to locate nodes where the package is not managed with Puppet, or to view nodes on which a package instance is managed with Puppet.

To quickly find the place in your manifest where a Puppet-managed package is declared, select a code path in the **Instances** selector and click **Copy path**.

6. To modify a package on a group of nodes:
 - If the package is managed with Puppet, select a code path in the **Instances** selector and click **Copy path**, then navigate to and update the manifest.
 - If the package is not managed with Puppet, click **Run > Task** and create a new task.

View package data collection metadata

The `puppet_inventory_metadata` fact reports whether package data collection is enabled on a node, and shows the time spent collecting package data on the node during the last Puppet run.

Before you begin

Make sure you have enabled package data collection for the nodes you wish to view.

1. Click **Classification** and select the node group you created when enabling package data collection.
2. Click **Matching nodes** and select a node from the list.
3. On the node's inventory page, click **Facts** and locate **puppet_inventory_metadata** in the list.

The fact value looks something like:

```
{
  "packages" : {
    "collection_enabled" : true,
    "last_collection_time" : "1.9149s"
  }
}
```

Disable package data collection

If you need to disable package data collection, set **package_inventory_enabled** to `false` and run Puppet twice.

1. Click **Classification** and select the node group you used when enabling package data collection.
2. Click **Configuration**.
3. In the **puppet_enterprise::profile::agent** class, locate **package_inventory_enabled** and click **Edit**.
4. Change the **Value** of **package_inventory_enabled** to `false`, then commit changes.
5. Run Puppet to apply these changes to the nodes in your node group and disable package data collection.

Package data is collected for a final time during this run.

6. Run Puppet a second time to purge package data from the impacted nodes' storage.

Infrastructure reports

Each time Puppet runs on a node, it generates a report that provides information such as when the run took place, any issues encountered during the run, and the activity of resources on the node. These reports are collected on the **Reports** page in the console.

Working with the reports table

The Reports page provides a summary view of key data from each report. Use this page to track recent node activity so you can audit your system and perform root cause analysis over time.

The reports table lists the number of resources on each node in each of the following states:

Correction applied	Number of resources that received a corrective change after Puppet identified resources that were out of sync with the applied catalog.
Failed	Number of resources that failed.
Changed	Number of resources that changed.
Unchanged	Number of resources that remained unchanged.
No-op	Number of resources that would have been changed if not run in no-op mode.
Skipped	Number of resources that were skipped because they depended on resources that failed.

Failed restarts

Number of resources that were supposed to restart but didn't.

For example, if changes to one resource notify another resource to restart, and that resource doesn't restart, a failed restart is reported. It's an indirect failure that occurred in a resource that was otherwise unchanged.

The reports table also offers the following information:

- **No-op mode:** An indicator of whether the node was run in no-op mode.
- **Config retrieval:** Time spent retrieving the catalog for the node (in seconds).
- **Run time:** Time spent applying the catalog on the node (in seconds).

Tip: Report count caching is used to improve console performance. In some cases, caching might cause summary counts of available reports to be displayed inaccurately the first time the page is accessed after a fresh install.

Filtering reports

You can filter the list of reports displayed on the **Reports** page by run status and by node fact. If you set a run status filter, and also set a node fact filter, the table takes both filters into account, and shows only those reports matching both filters.

Clicking **Remove filter** removes all filters currently in effect.

The filters you set are persistent. If you set run status or fact filters on the **Reports** page, they continue to be applied to the table until they're changed or removed, even if you navigate to other pages in the console or log out. The persistent storage is associated with the browser tab, not your user account, and is cleared when you close the tab.

Filter by node run status

Filter reports to quickly focus on nodes with failures or change events by using the **Filter by run status** bar.

1. Select a run status (such as **No-op mode: with failures**). The table updates to reflect your filter selection.
2. To remove the run status filter, select **All run statuses**.

Filter by node fact

You can create a highly specific list of nodes for further investigation by using the fact filter tool.

For example, you can check that nodes you've updated have successfully changed, or find out the operating systems or IP addresses of a set of failed nodes to better understand the failure. You might also filter by facts to fulfill an auditor's request for information, such as the number of nodes running a particular version of software.

1. Click **Filter by fact value**. In the **Fact** field, select one of the available facts. An empty fact field is not allowed.

Tip: To see the facts and values reported by a node on its most recent run, click the node name in the **Run status** table, then select the node's **Facts** tab.

2. Select an Operator:

Operator	Meaning	Notes
=	is	
!=	is not	
~	matches a regular expression (regex)	Select this operator to use wildcards and other regular expressions if you want to find matching facts without having to specify the exact value.
!~	does not match a regular expression (regex)	

Operator	Meaning	Notes
>	greater than	Can be used only with facts that have a numeric value.
>=	greater than or equal to	Can be used only with facts that have a numeric value.
<	less than	Can be used only with facts that have a numeric value.
<=	less than or equal to	Can be used only with facts that have a numeric value.

3. In the **Value** field, enter a value. Strings are case-sensitive, so make sure you use the correct case.

The filter displays an error if you use an invalid string operator (for example, selecting a numeric value operator such as `>=` and entering a non-numeric string such as `pilsen` as the value) or enter an invalid regular expression.

Note: If you enter an invalid or empty value in the **Value** field, PE takes the following action in order to avoid a filter error:

- Invalid or empty Boolean facts are processed as **false**, and results are retrieved accordingly.
- Invalid or empty numeric facts are processed as **0**, and results are retrieved accordingly.
- Invalid or incomplete regular expressions invalidate the filter, and no results are retrieved.

4. Click **Add**.

5. As needed, repeat these steps to add additional filters. If filtering by more than one node fact, specify either **Nodes must match all rules** or **Nodes can match any rule**.

Working with individual reports

To examine a report in greater detail, click **Report time**. This opens a page that provides details for the node's resources in three sections: **Events**, **Log**, and **Metrics**.

Events

The **Events** tab lists the events for each managed resource on the node, its status, whether correction was applied to the resource, and — if it changed — what it changed from and what it changed to. For example, a user or a file might change from absent to present.

To filter resources by event type, click **Filter by event status** and choose an event.

Sort resources by name or events by severity level, ascending or descending, by clicking the **Resource** or **Events** sorting controls.

To download the events data as a `.csv` file, click **Export data**. The filename is `events-<node name>-<timestamp>`.

Log

The **Log** tab lists errors, warnings, and notifications from the node's latest Puppet run.

Each message is assigned one of the following severity levels:

Standard	Caution (yellow)	Warning (red)
debug	warning	err
info	alert	emerg
notice		crit

To read the report chronologically, click the time sorting controls. To read it in order of issue severity, click the severity level sorting controls.

To download the log data as a .csv file, click **Export data**. The filename is `log-<node name>-<timestamp>`.

Metrics

The **Metrics** tab provides a summary of the key data from the node's latest Puppet run.

Metric	Description
Report submitted by:	The certname of the Puppet master that submitted the report to PuppetDB.
Puppet environment	The environment assigned to the node.
Puppet run	<ul style="list-style-type: none"> The time that the Puppet run began The time that the Puppet master submitted the catalog The time that the Puppet run finished The time PuppetDB received the report The duration of the Puppet run The length of time to retrieve the catalog The length of time to apply the resources to the catalog
Catalog application	Information about the catalog application that produces the report: the config version that Puppet uses to match a specific catalog for a node to a specific Puppet run, the catalog UUID that identifies the catalog used to generate a report during a Puppet run, and whether the Puppet run used a cached catalog.
Resources	The total number of resources in the catalog.
Events	A list of event types and the total count for each one.
Top resource types	A list of the top resource types by time, in seconds, it took to be applied.

Analyzing changes across Puppet runs

The **Events** page in the console shows a summary of activity in your infrastructure. You can analyze the details of important changes, and investigate common causes behind related events. You can also examine specific class, node, and resource events, and find out what caused them to fail, change, or run as no-op.

What is an event?

An event occurs whenever PE attempts to modify an individual property of a given resource. Reviewing events lets you see detailed information about what has changed on your system, or what isn't working properly.

During a Puppet run, Puppet compares the current state of each property on each resource to the desired state for that property, as defined by the node's catalog. If Puppet successfully compares the states and the property is already in sync (in other words, if the current state is the desired state), Puppet moves on to the next resource without noting anything. Otherwise, it attempts some action and records an event, which appears in the report it sends to the Puppet master at the end of the run. These reports provide the data presented on the **Events** page in the console.

Event types

There are six types of event that can occur when Puppet reviews each property in your system and attempts to make any needed changes. If a property is already in sync with its catalog, no event is recorded: no news is good news in the world of events.

Event	Description
Failure	A property was out of sync; Puppet tried to make changes, but was unsuccessful.
Corrective change	Puppet found an inconsistency between the last applied catalog and a property's configuration, and corrected the property to match the catalog.
Intentional change	Puppet applied catalog changes to a property.
Corrective no-op	Puppet found an inconsistency between the last applied catalog and a property's configuration, but Puppet was instructed to not make changes on this resource, via either the <code>--noop</code> command-line option, the <code>noop</code> setting, or the <code>noop => true</code> metaparameter. Instead of making a corrective change, Puppet logs a corrective no-op event and reports the change it would have made.
Intentional no-op	Puppet would have applied catalog changes to a property, but Puppet was instructed to not make changes on this resource, via either the <code>--noop</code> command-line option, the <code>noop</code> setting, or the <code>noop => true</code> metaparameter. Instead of making an intentional change, Puppet logs an intentional no-op event and reports the change it would have made.
Skip	<p>A prerequisite for this resource was not met, so Puppet did not compare its current state to the desired state. This prerequisite is either one of the resource's dependencies or a timing limitation set with the <code>schedule</code> metaparameter. The resource might be in sync or out of sync; Puppet doesn't know yet..</p> <p>If the <code>schedule</code> metaparameter is set for a given resource, and the scheduled time hasn't arrived when the run happens, that resource logs a skip event on the Events page. This is true for a user-defined <code>schedule</code>, but does not apply to built-in scheduled tasks that happen weekly, daily, or at other intervals.</p>

Working with the Events page

During times when your deployment is in a state of stability, with no changes being made and everything functioning optimally, the **Events** page reports little activity, and might not seem terribly interesting. But when change occurs—when packages require upgrades, when security concerns threaten, or when systems fail—the **Events** page helps you understand what's happening and where so you can react quickly.

The **Events** page fetches data when loading, and does not refresh—even if there's a Puppet run while you're on the page—until you close or reload the page. This ensures that shifting data won't disrupt an investigation.

You can see how recent the shown data is by checking the timestamp at the top of the page. Reload the page to update the data to the most recent events.

Tip: Keeping time synchronized by running NTP across your deployment helps the **Events** page produce accurate information. NTP is easily managed with PE, and setting it up is an excellent way to learn Puppet workflows.

Monitoring infrastructure with the Events summary pane

The **Events** page displays all events from the latest report of every responsive node in the deployment.

Tip: By default, PE considers a node unresponsive after one hour, but you can configure this setting to meet your needs by adjusting the `puppet_enterprise::console_services::no_longer_reporting_cutoff` parameter.

On the left side of the screen, the **Events** summary pane shows an overview of Puppet activity across your infrastructure. This data can help you rapidly assess the magnitude of any issue.

The **Events** summary pane is split into three categories—the **Classes** summary, **Nodes** summary, and **Resources** summary—to help you investigate how a change or failure event impacts your entire deployment.

Gaining insight with the Events detail pane

Clicking an item in the **Events** summary pane loads its details (and any sub-items) in the **Events** detail pane on the right of the screen. The summary pane on the left always shows the list of items from which the one in the detail pane on the right was chosen, to let you easily view similar items and compare their states.

Click any item in the the **Classes** summary, **Nodes** summary, or **Resources** summary to load more specific info into the detail pane and begin looking for the causes of notable events. Switch between perspectives to find the common threads among a group of failures or corrective changes, and follow them to a root cause.

Analyzing changes and failures

You can use the **Events** page to analyze the root causes of events resulting from a Puppet run. For example, to understand the cause of a failure after a Puppet run, select the class, node, or resource with a failure in the **Events** summary pane, and then review the details of the failure in the **Events** detail pane.

You can view additional details by clicking on the failed item in the in the **Events** detail pane.

Use the **Classes** summary, **Nodes** summary, and **Resources** summary to focus on the information you need. For example, if you're concerned about a failed service, say Apache or MongoDB, you can start by looking into failed resources or classes. If you're experiencing a geographic outage, you might start by drilling into failed node events.

Understanding event display issues

In some special cases, events are not displayed as expected on the **Events** page. These cases are often caused by the way that the console receives data from other parts of Puppet Enterprise, but sometimes are due to the way your Puppet code is interpreted.

Runs that restart PuppetDB are not displayed

If a given Puppet run restarts PuppetDB, Puppet is not able to submit a run report from that run to PuppetDB because PuppetDB is not available. Because the **Events** page relies on data from PuppetDB, and PuppetDB reports are not queued, the **Events** page does not display any events from that run. Note that in such cases, a run report *is* available on the **Reports** page. Having a Puppet run restart PuppetDB is an unlikely scenario, but one that could arise in cases where some change to, say, a parameter in the `puppetdb` class causes the `pe-puppetdb` service to restart.

Runs without a compiled catalog are not displayed

If a run encounters a catastrophic failure where an error prevents a catalog from compiling, the **Events** page does not display any failures. This is because no events occurred.

Simplified display for some resource types

For resource types that take the `ensure` property, such as user or file resource types, the **Events** page displays a single event when the resource is first created. This is because Puppet has changed only one property (`ensure`), which sets all the baseline properties of that resource at the same time. For example, all of the properties of a given user are created when the user is added, just as if the user was added manually. If a later Puppet run changes properties of that user resource, each individual property change is shown as a separate event.

Updated modes display without leading zeros

When the mode attribute for a `file` resource is updated, and numeric notation is used, leading zeros are omitted in the **New Value** field on the **Events** page. For example, 0660 is shown as 660 and 0000 is shown as 0.

Viewing and managing Puppet Server metrics

Puppet Server can provide performance and status metrics to external services for monitoring server health and performance over time.

You can track Puppet Server metrics by using:

- Customizable, networked Graphite and Grafana instances
- A built-in experimental developer dashboard
- Status API endpoints

Note: None of these methods are officially supported. The Grafanadash and Puppet-graphite modules referenced in this document are not Puppet-supported modules; they are mentioned for testing and demonstration purposes only. The developer dashboard is a tech preview. Both the Grafana and developer dashboard methods take advantage of the Status API, including some endpoints that are also a tech preview.

Getting started with Graphite

Puppet Enterprise can export many metrics to Graphite, a third-party monitoring application that stores real-time metrics and provides customizable ways to view them. After Graphite support is enabled, Puppet Server 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 different Puppet Server metrics on demand. However, it requires considerable configuration and additional server resources. For an easier, but more limited, web-based dashboard of Puppet Server metrics built into Puppet Server, use the developer dashboard. To retrieve metrics manually via HTTP, use the Status API.

To use Graphite with Puppet Enterprise, 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

The Grafanadash 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.



CAUTION: The Grafanadash module referenced in this document is not a Puppet-supported module; it is for testing and demonstration purposes only. It is tested against CentOS 7 only. Also, install this module on a dedicated agent only. Do not install it on the Puppet master, because the module makes security policy changes that are inappropriate for a Puppet master:

- 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 module

Install the Grafanadash 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 PE 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 is a dashboard that can interpret and visualize Puppet Server metrics over time, but you must configure it to do so.

Grafana runs as a web dashboard, and the Grafanadash module configures it at port 10000 by default. However, there are no Puppet metrics displayed by default. You must create a metrics dashboard to view Puppet's metrics in Grafana, or edit and import a JSON-based dashboard such as the sample Grafana dashboard that we provide.

1. In a web browser on a computer that can reach the Puppet agent node, navigate to `http://<AGENT'S HOSTNAME>:10000`.
2. Open the `sample_metrics_dashboard.json` file in a text editor on the same computer you're using to access Grafana.
3. Throughout the file, replace our sample setting of `master.example.com` with the hostname of your Puppet master. This value must be used as the `metrics_server_id` setting, as configured below.
4. Save the file.
5. In the Grafana UI, click **search** (the folder icon), then **Import**, then **Browse**.
6. 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. However, these graphs remain empty until you enable Puppet Server's Graphite metrics.

Related information

[Sample Grafana dashboard graphs](#) on page 295

Use the sample Grafana dashboard as your starting point and customize it to suit your needs. You can click on the title of any graph, and then click **edit** to adjust the graphs as you see fit.

Enabling Puppet Server's Graphite support

Use the PE Master node group in the console to configure Puppet Server's metrics output settings.

1. In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Master** group.
2. On the **Configuration** tab, in the `puppet_enterprise::profile::master` class, add these parameters:
 - a) Set `metrics_graphite_enabled` to `true` (default is `false`).
 - b) Set `metrics_server_id` to the Puppet master hostname.
 - c) Set `metrics_graphite_host` to the hostname for the agent node on which you're running Graphite and Grafana.
 - d) Set `metrics_graphite_update_interval_seconds` to a value to set Graphite's update frequency in seconds. This setting is optional, and the default value is 60.
3. Verify that these parameters are set to their default values, unless your Graphite server uses a non-standard port:
 - a) Set `metrics_jmx_enabled` to `true` (default value).
 - b) Set `metrics_graphite_port` to 2003 (default value) or the Graphite port on your Graphite server.
 - c) Set `profiler_enabled` to `true` (default value).
4. Commit changes.

Note: In the Grafana UI, choose an appropriate time window from the drop-down menu.

Note: The `puppet_enterprise::profile::master::metrics_enabled` parameter used in Puppet Enterprise 2016.3 and earlier is no longer necessary and has been deprecated. If you set it, PE notifies you of the setting's deprecation.

Sample Grafana dashboard graphs

Use the sample Grafana dashboard as your starting point and customize it to suit your needs. You can click on the title of any graph, and then click **edit** to adjust the graphs as you see fit.

[Sample Grafana dashboard code](#)

Graph name	Description
Active requests	<p>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.</p>
Request durations	<p>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.</p>
Request ratios	<p>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.</p>
External HTTP Communications	<p>This graph tracks the amount of time it takes Puppet Server to send data and requests for common operations to, and receive responses from, external HTTP services, such as PuppetDB.</p>
File Sync	<p>This graph tracks how long Puppet Server spends on File Sync operations, for both its storage and client services.</p>

Graph name	Description
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, increase the number of JRuby instances to allow it to allocate more JRubies. Otherwise, consider adding additional compilers to distribute the catalog compilation load.
JRuby Timers	<p>This graph tracks several JRuby pool metrics.</p> <ul style="list-style-type: none"> • 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. This occurs while Puppet Server synchronizes code for File Sync. • The lock wait time represents the amount of time that Puppet Server waits to acquire a lock on the pool. <p>These metrics help identify sources of potential JRuby allocation bottlenecks.</p>
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 on the master.

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 `master.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.master.example.com.num-cpus, 'num
cpus')"
            },
            {
                "target": "alias(puppetlabs.master.example.com.http.active-
requests.count, 'active requests')"
            },
            {
                "target": "alias(puppetlabs.master.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

These HTTP and Puppet profiler metrics are available from the Puppet Server and can be added to your metrics reporting.

Graphite metrics properties

Each metric is prefixed with `puppetlabs.<MASTER-HOSTNAME>`; for instance, the Grafana dashboard file refers to the `num-cpus` metric as `puppetlabs.<MASTER-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.<MASTER-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 metrics—so many that past versions of Puppet Enterprise could overwhelm Grafana servers. As of Puppet Enterprise 2016.4, Puppet Server exports only a subset of its available metrics by default. This

set is designed to report the most relevant Puppet Server metrics for administrators monitoring its 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.<MASTER-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.<MASTER-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.<MASTER-HOSTNAME>.compiler.find_facts`: The time spent parsing facts.
 - `puppetlabs.<MASTER-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.<MASTER-HOSTNAME>.compiler.static_compile`: The time spent compiling static catalogs.
 - `puppetlabs.<MASTER-HOSTNAME>.compiler.static_compile_inlining`: The time spent inlining metadata for static catalogs.
 - `puppetlabs.<MASTER-HOSTNAME>.compiler.static_compile_postprocessing`: The time spent post-processing static catalogs.

File sync metrics:

- `puppetlabs.<MASTER-HOSTNAME>.file-sync-client.clone-timer`: The time spent by file sync clients on compilers initially cloning repositories on the master.
- `puppetlabs.<MASTER-HOSTNAME>.file-sync-client.fetch-timer`: The time spent by file sync clients on compilers fetching repository updates from the master.
- `puppetlabs.<MASTER-HOSTNAME>.file-sync-client.sync-clean-check-timer`: The time spent by file sync clients on compilers checking whether the repositories are clean.
- `puppetlabs.<MASTER-HOSTNAME>.file-sync-client.sync-timer`: The time spent by file sync clients on compilers synchronizing code from the private datadir to the live codedir.
- `puppetlabs.<MASTER-HOSTNAME>.file-sync-storage.commit-add-rm-timer`
- `puppetlabs.<MASTER-HOSTNAME>.file-sync-storage.commit-timer`: The time spent committing code on the master into the file sync repository.

Function metrics:

- `puppetlabs.<MASTER-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.<MASTER-HOSTNAME>.functions.<FUNCTION-NAME>.mean`.

HTTP metrics:

- `puppetlabs.<MASTER-HOSTNAME>.http.active-histo`: A histogram of active HTTP requests over time.
- `puppetlabs.<MASTER-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.<MASTER-HOSTNAME>.http.puppet-v3-environment-/*/-requests`: The time Puppet Server has spent handling environment requests, including time spent waiting for an available JRuby instance.

- `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-environment_classes-/*/-requests`: The time spent handling requests to the `environment_classes` API endpoint, which the Node Classifier uses to refresh classes.
- `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-environments-requests`: The time spent handling requests to the `environments` API endpoint requests made by the Orchestrator
- The following metrics measure the time spent handling file-related API endpoints:
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-file_bucket_file-/*/-requests`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-file_content-/*/-requests`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-file_metadata-/*/-requests`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-file_metadatas-/*/-requests`
- `puppetlabs.<MASTER-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.<MASTER-HOSTNAME>.http.puppet-v3-report-/*/-requests`: The time spent handling report requests. A bottleneck here might indicate an issue with PuppetDB.
- `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-static_file_content-/*/-requests`: The time spent handling requests to the `static_file_content` API endpoint used by Direct Puppet with file sync.

JRuby metrics: Puppet Server uses an embedded JRuby interpreter to execute Ruby code. JRuby spawns parallel instances known as JRubies to execute Ruby code, which occurs during most Puppet Server activities. See [Tuning JRuby on Puppet Server](#) for details on adjusting JRuby settings.

- `puppetlabs.<MASTER-HOSTNAME>.jruby.borrow-timer`: The time spent with a borrowed JRuby.
- `puppetlabs.<MASTER-HOSTNAME>.jruby.free-jrubies-histo`: A histogram of free JRubies over time. This metric's average value must be greater than 1; if it isn't, more JRubies or another compiler might be needed to keep up with requests.
- `puppetlabs.<MASTER-HOSTNAME>.jruby.lock-held-timer`: The time spent holding the JRuby lock.
- `puppetlabs.<MASTER-HOSTNAME>.jruby.lock-wait-timer`: The time spent waiting to acquire the JRuby lock.
- `puppetlabs.<MASTER-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.<MASTER-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.<MASTER-HOSTNAME>.puppetdb.catalog.save`
- `puppetlabs.<MASTER-HOSTNAME>.puppetdb.command.submit`
- `puppetlabs.<MASTER-HOSTNAME>.puppetdb.facts.find`
- `puppetlabs.<MASTER-HOSTNAME>.puppetdb.facts.search`
- `puppetlabs.<MASTER-HOSTNAME>.puppetdb.report.process`
- `puppetlabs.<MASTER-HOSTNAME>.puppetdb.resource.search`

Counters only

HTTP metrics:

- `puppetlabs.<MASTER-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.<MASTER-HOSTNAME>.http.puppet-v3-catalog-/*/-percentage`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-environment-/*/-percentage`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-environment_classes-/*/-percentage`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-environments-percentage`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-file_bucket_file-/*/-percentage`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-file_content-/*/-percentage`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-file_metadata-/*/-percentage`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-file_metadatas-/*/-percentage`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-node-/*/-percentage`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-report-/*/-percentage`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-resource_type-/*/-percentage`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-resource_types-/*/-percentage`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-static_file_content-/*/-percentage`
 - `puppetlabs.<MASTER-HOSTNAME>.http.puppet-v3-status-/*/-percentage`
- `puppetlabs.<MASTER-HOSTNAME>.http.total-requests`: The total requests handled by Puppet Server.

JRuby metrics:

- `puppetlabs.<MASTER-HOSTNAME>.jruby.borrow-count`: The number of successfully borrowed JRubies.
- `puppetlabs.<MASTER-HOSTNAME>.jruby.borrow-retry-count`: The number of attempts to borrow a JRuby that must be retried.
- `puppetlabs.<MASTER-HOSTNAME>.jruby.borrow-timeout-count`: The number of attempts to borrow a JRuby that resulted in a timeout.
- `puppetlabs.<MASTER-HOSTNAME>.jruby.request-count`: The number of requested JRubies.
- `puppetlabs.<MASTER-HOSTNAME>.jruby.return-count`: The number of JRubies successfully returned to the pool.
- `puppetlabs.<MASTER-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 compilers.
- `puppetlabs.<MASTER-HOSTNAME>.jruby.num-jrubies`: The total number of JRuby instances on the server, governed by the `max-active-instances` setting. See [Tuning JRuby on Puppet Server](#) for details.

Other metrics

These metrics measure raw resource availability and capacity.

- `puppetlabs.<MASTER-HOSTNAME>.num-cpus`: The number of available CPUs on the server.
- `puppetlabs.<MASTER-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.<MASTER-HOSTNAME>.memory.total.committed`
 - `puppetlabs.<MASTER-HOSTNAME>.memory.total.init`
 - `puppetlabs.<MASTER-HOSTNAME>.memory.total.used`
 - `puppetlabs.<MASTER-HOSTNAME>.memory.total.max`
 - `puppetlabs.<MASTER-HOSTNAME>.memory.heap.committed`
 - `puppetlabs.<MASTER-HOSTNAME>.memory.heap.init`
 - `puppetlabs.<MASTER-HOSTNAME>.memory.heap.used`
 - `puppetlabs.<MASTER-HOSTNAME>.memory.heap.max`
 - `puppetlabs.<MASTER-HOSTNAME>.memory.non-heap.committed`
 - `puppetlabs.<MASTER-HOSTNAME>.memory.non-heap.init`
 - `puppetlabs.<MASTER-HOSTNAME>.memory.non-heap.used`
 - `puppetlabs.<MASTER-HOSTNAME>.memory.non-heap.max`

Related information

[Modifying exported metrics](#) on page 302

In addition to the default metrics, you can also export metrics measuring specific environments and nodes managed by Puppet Server.

Modifying exported metrics

In addition to the default metrics, you can also export metrics measuring specific environments and nodes managed by Puppet Server.

The `$metrics_puppetserver_metrics_allowed` class parameter in the `puppet_enterprise::profile::master` class takes an array of metrics as strings. To export additional metrics, add them to this array.

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.

Omit the `puppetlabs.<MASTER-HOSTNAME>` prefix and field suffixes (such as `.count` or `.mean`) from metrics. Instead, suffix the environment or node name as a field to the metric.

1. For example, to track the compilation time for the production environment, add `compiler.compile.production` to the `metrics-allowed` list.
2. To track only the `my.node.localdomain` node in the production environment, add `compiler.compile.production.my.node.localdomain` to the `metrics-allowed` list.

Status API

The status API allows you to check the health of PE components. It can be useful in automated monitoring of your PE infrastructure, removing unhealthy service instances from a load-balanced pool, checking configuration values, or when troubleshooting problems in PE.

You can check the overall health of `pe-console-services`, as well as the health of the individual services within `pe-console-services`:

- Activity service
- Node classifier
- PuppetDB
- Puppet Server
- Role-based access control (RBAC)

- Code Manager (if configured)

The endpoints provide overview health information in an overall healthy/error/unknown status field, and fine-detail information such as the availability of the database, the health of other required services, or connectivity to the Puppet master.

Related information

[API index](#) on page 33

APIs allow you to interact with Puppet and Puppet Enterprise (PE) applications from your own code or application integration hooks.

Authenticating to the status API

Token-based authentication is not required to access the status API. You can choose to authenticate requests by using whitelisted certificates, or can access the API without authentication via HTTP.

You can authenticate requests using a certificate listed in RBAC's certificate whitelist, located at `/etc/puppetlabs/console-services/rbac-certificate-whitelist`. Note that if you edit this file, you must reload the `pe-console-services` service (run `sudo service pe-console-services reload`) for your changes to take effect.

The status API's endpoints can be served over HTTP, which does not require any authentication. This is disabled by default.

Tip: To use HTTP, locate the **PE Console** node group in the console, and in the `puppet_enterprise::profile::console` class, set `console_services_plaintext_status_enabled` to `true`.

Forming requests to the status API

The HTTPS status endpoints are available on the console services API server, which uses port 4433 by default.

The path prefix is `/status`, so, for example, the URL to get the statuses for all services as JSON is `https://<DNS NAME OF YOUR CONSOLE HOST>:4433/status/v1/services`.

To access that URL using curl commands, run:

```
curl https://<DNS NAME OF YOUR CONSOLE HOST>:4433/status/v1/services \
  --cert /etc/puppetlabs/puppet/ssl/certs/<WHITELISTED CERTNAME>.pem \
  --key /etc/puppetlabs/puppet/ssl/private_keys/<WHITELISTED CERTNAME>.pem \
  --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem
```

If enabled, the HTTP status endpoints are available on port 8123.

Tip: To change the port, locate the **PE Console** node group in the console, and in the `puppet_enterprise::profile::console` class, set the `console_services_plaintext_status_port` parameter to your desired port number.

JSON endpoints

These two endpoints provide machine-consumable information about running services. They are intended for scripting and integration with other services.

GET /status/v1/services

Use the `/services` endpoint to retrieve the statuses of all PE services.

The content type for this endpoint is `application/json; charset=utf-8`.

Query parameters

The request accepts the following parameters:

Parameter	Value
level	How thorough of a check to run. Set to <code>critical</code> , <code>debug</code> , or <code>info</code> (default).
timeout	Specified in seconds; defaults to 30.

Response format

The response is a JSON object that lists details about the services, using the following keys:

Key	Definition
service_version	Package version of the JAR file containing a given service.
service_status_version	The version of the API used to report the status of the service.
detail_level	Can be <code>critical</code> , <code>debug</code> , or <code>info</code> .
state	Can be <code>running</code> , <code>error</code> , or <code>unknown</code> .
status	An object with the service's status details. Usually only relevant for <code>error</code> and <code>unknown</code> states.
active_alerts	An array of objects containing <code>severity</code> and a message about your replication from pglogical if you have replication enabled; otherwise, it's an empty array.

For example:

```
{ "rbac-service": { "service_version": "1.8.11-SNAPSHOT",
  "service_status_version": 1,
  "detail_level": "info",
  "state": "running",
  "status": {
    "activity_up": true,
    "db_up": true,
    "db_pool": { "state": "ready" },
    "replication": { "mode": "none", "status": "none" }
  },
  "active_alerts": [],
  "service_name": "rbac-service"
}

"classifier-service": { "service_version": "1.8.11-SNAPSHOT",
  "service_status_version": 1,
  "detail_level": "info",
  "state": "running",
  "status": {
    "activity_up": true,
    "db_up": true,
    "db_pool": { "state": "ready" },
    "replication": { "mode": "none", "status": "none" }
  },
  "active_alerts": [],
  "service_name": "classifier-service"
}
```


Response codes

The server uses the following response codes:

- 200 if and only if all services report a status of `running`
- 503 if any service's status is `unknown` or `error`
- 400 if a `level` parameter is set but is invalid (not `critical`, `debug`, or `info`)

GET /status/v1/services/<SERVICE NAME>

Use the `/services/<SERVICE NAME>` endpoint to retrieve the status of a particular PE service.

The content type for this endpoint is `application/json; charset=utf-8`.

Query parameters

The request accepts the following parameters:

Parameter	Value
<code>level</code>	How thorough of a check to run. Set to <code>critical</code> , <code>debug</code> , or <code>info</code> (default).
<code>timeout</code>	Specified in seconds; defaults to 30.

Response format

The response is a JSON object that lists details about the service, using the following keys:

Key	Definition
<code>service_version</code>	Package version of the JAR file containing a given service.
<code>service_status_version</code>	The version of the API used to report the status of the service.
<code>detail_level</code>	Can be <code>critical</code> , <code>debug</code> , or <code>info</code> .
<code>state</code>	Can be <code>running</code> , <code>error</code> , or <code>unknown</code> .
<code>status</code>	An object with the service's status details. Usually only relevant for <code>error</code> and <code>unknown</code> states.
<code>active_alerts</code>	An array of objects containing <code>severity</code> and a message about your replication from pglogical if you have replication enabled; otherwise, it's an empty array.

For example:

```
{ "rbac-service": { "service_version": "1.8.11-SNAPSHOT",
  "service_status_version": 1,
  "detail_level": "info",
  "state": "running",
  "status": {
    "activity_up": true,
    "db_up": true,
    "db_pool": { "state": "ready" },
    "replication": { "mode": "none", "status": "none" }
  },
  "active_alerts": [],
}
```

Response codes

The server uses the following response codes:

- 200 if and only if all services report a status of `running`
- 503 if any service's status is `unknown` or `error`
- 400 if a `level` parameter is set but is invalid (not `critical`, `debug`, or `info`)
- 404 if no service named `<SERVICE NAME>` is found

Activity service plaintext endpoints

The activity service plaintext endpoints are designed for load balancers that don't support any kind of JSON parsing or parameter setting. They return simple string bodies (either the state of the service in question or a simple error message) and a status code relevant to the status result.

GET /status/v1/simple

The `/status/v1/simple` returns a status that reflects all services the status service knows about.

The content type for this endpoint is `text/plain; charset=utf-8`.

Query parameters

No parameters are supported. Defaults to using the `critical` status level.

Response codes

The server uses the following response codes:

- 200 if and only if all services report a status of `running`
- 503 if any service's status is `unknown` or `error`

Possible responses

The endpoint returns a status that reflects all services it knows about. It decides on what status to report using the following logic:

- `running` if and only if all services are `running`
- `error` if any service reports an `error`
- `unknown` if any service reports an `unknown` and no services report an `error`

GET /status/v1/simple/<SERVICE NAME>

The `/status/v1/simple/<SERVICE NAME>` endpoint returns the plaintext status of the specified service, such as `rbac-service` or `classifier-service`.

The content type for this endpoints is `text/plain; charset=utf-8`.

Query parameters

No parameters are supported. Defaults to using the `critical` status level.

Response codes

The server uses the following response codes:

- 200 if and only if all services report a status of `running`
- 503 if any service's status is `unknown` or `error`
- 404 if no service named `<SERVICE NAME>` is found

Possible responses

The endpoint returns a status that reflects all services it knows about. It decides on what status to report using the following logic:

- `running` if and only if all services are running
- `error` if any service reports an error
- `unknown` if any service reports an unknown and no services report an error
- `not found: <SERVICENAME>` if any service can't be found

Metrics endpoints

Puppet Server is capable of tracking advanced metrics to give you additional insight into its performance and health.

The HTTPS metrics endpoints are available on port 8140 of the master server:

```
curl -k https://<DNS NAME OF YOUR MASTER>:8140/status/v1/services?
level=debug`
```

Note: These API endpoints are a tech preview. The metrics described here are returned only when passing the `level=debug` URL parameter, and the structure of the returned data might change in future versions.

These metrics fall into three categories:

- JRuby metrics (`/status/v1/services/pe-jruby-metrics`)
- HTTP route metrics (`/status/v1/services/pe-master`)
- Catalog compilation profiler metrics (`/status/v1/services/pe-puppet-profiler`)

All of these metrics reflect data for the lifetime of the current Puppet Server process and reset whenever the service is restarted. Any time-related metrics report milliseconds unless otherwise noted.

Like the standard status endpoints, the metrics endpoints return machine-consumable information about running services. This JSON response includes the same keys returned by a standard status endpoint request (see JSON endpoints). Each endpoint also returns additional keys in an `experimental` section.

GET /status/v1/services/pe-jruby-metrics

The `/status/v1/services/pe-jruby-metrics` endpoint returns JSON containing information about the JRuby pools from which Puppet Server fulfills agent requests.

You must query it at port 8140 and append the `level=debug` URL parameter.

Query parameters

No parameters are supported. Defaults to using the `critical` status level.

Response codes

The server uses the following response codes:

- `200` if and only if all services report a status of `running`
- `503` if any service's status is `unknown` or `error`

Response keys

The metrics are returned in two subsections of the `experimental` section: `jruby-pool-lock-status` and `metrics`.

The response's `experimental/jruby-pool-lock-status` section contains the following keys:

Key	Definition
<code>current-state</code>	The state of the JRuby pool lock, which should be either <code>:not-in-use</code> (unlocked), <code>:requested</code> (waiting for lock), or <code>:acquired</code> (locked).
<code>last-change-time</code>	The date and time of the last <code>current-state</code> update, formatted as an ISO 8601 combined date and time in UTC.

The response's `experimental/metrics` section contains the following keys:

Key	Definition
<code>average-borrow-time</code>	The average amount of time a JRuby instance spends handling requests, calculated by dividing the total duration in milliseconds of the <code>borrowed-instances</code> value by the <code>borrow-count</code> value.
<code>average-free-jrubies</code>	The average number of JRuby instances that are not in use over the Puppet Server process's lifetime.
<code>average-lock-held-time</code>	The average time the JRuby pool held a lock, starting when the value of <code>jruby-pool-lock-status/current-state</code> changed to <code>:acquired</code> . This time mostly represents file sync syncing code into the live codedir, and is calculated by dividing the total length of time that Puppet Server held the lock by the value of <code>num-pool-locks</code> .
<code>average-lock-wait-time</code>	The average time Puppet Server spent waiting to lock the JRuby pool, starting when the value of <code>jruby-pool-lock-status/current-state</code> changed to <code>:requested</code> . This time mostly represents how long Puppet Server takes to fulfill agent requests, and is calculated by dividing the total length of time that Puppet Server waits for locks by the value of <code>num-pool-locks</code> .
<code>average-requested-jrubies</code>	The average number of requests waiting on an available JRuby instance over the Puppet Server process's lifetime.
<code>average-wait-time</code>	The average time Puppet Server spends waiting to reserve an instance from the JRuby pool, calculated by dividing the total duration in milliseconds of <code>requested-instances</code> by the <code>requested-count</code> value.
<code>borrow-count</code>	The total number of JRuby instances that have been used.
<code>borrow-retry-count</code>	The total number of times that a borrow attempt failed and was retried, such as when the JRuby pool is flushed while a borrow attempt is pending.
<code>borrow-timeout-count</code>	The number of requests that were not served because they timed out while waiting for a JRuby instance.

Key	Definition
<code>borrowed-instances</code>	<p>A list of the JRuby instances currently in use, each reporting:</p> <ul style="list-style-type: none"> <code>duration-millis</code>: The length of time that the instance has been running. <code>reason/request</code>: A hash of details about the request being served. <ul style="list-style-type: none"> <code>request-method</code>: The HTTP request method, such as POST, GET, PUT, or DELETE. <code>route-id</code>: The route being served. For routing metrics, see the HTTP metrics endpoint. <code>uri</code>: The request's full URI. <code>time</code>: The time (in milliseconds since the Unix epoch) when the JRuby instance was borrowed.
<code>num-free-jrubies</code>	The number of JRuby instances in the pool that are ready to be used.
<code>num-jrubies</code>	The total number of JRuby instances.
<code>num-pool-locks</code>	The total number of times the JRuby pools have been locked.
<code>requested-count</code>	The number of JRuby instances borrowed, waiting, or that have timed out.
<code>requested-instances</code>	<p>A list of the requests waiting to be served, each reporting:</p> <ul style="list-style-type: none"> <code>duration-millis</code>: The length of time the request has waited. <code>reason/request</code>: A hash of details about the waiting request. <ul style="list-style-type: none"> <code>request-method</code>: The HTTP request method, such as POST, GET, PUT, or DELETE. <code>route-id</code>: The route being served. For routing metrics, see the HTTP metrics endpoint. <code>uri</code>: The request's full URI. <code>time</code>: The time (in milliseconds since the Unix epoch) when Puppet Server received the request.
<code>return-count</code>	The total number of JRuby instances that have been used.

For example:

```
"pe-jruby-metrics": {
  "detail_level": "debug",
  "service_status_version": 1,
  "service_version": "2.2.22",
  "state": "running",
  "status": {
    "experimental": {
      "jruby-pool-lock-status": {
        "current-state": ":not-in-use",
        "last-change-time": "2015-12-03T18:59:12.157Z"
```

```

    },
    "metrics": {
      "average-borrow-time": 292,
      "average-free-jrubies": 0.4716243097301104,
      "average-lock-held-time": 1451,
      "average-lock-wait-time": 0,
      "average-requested-jrubies": 0.21324752542875958,
      "average-wait-time": 156,
      "borrow-count": 639,
      "borrow-retry-count": 0,
      "borrow-timeout-count": 0,
      "borrowed-instances": [
        {
          "duration-millis": 3972,
          "reason": {
            "request": {
              "request-method": "post",
              "route-id": "puppet-v3-catalog-/*/",
              "uri": "/puppet/v3/catalog/"
            }
          }
        }
      ],
      "time": 1448478371406
    },
    {
      "num-free-jrubies": 0,
      "num-jrubies": 1,
      "num-pool-locks": 2849,
      "requested-count": 640,
      "requested-instances": [
        {
          "duration-millis": 3663,
          "reason": {
            "request": {
              "request-method": "put",
              "route-id": "puppet-v3-report-/*/",
              "uri": "/puppet/v3/report/"
            }
          }
        }
      ],
      "time": 1448478371715
    },
    {
      "return-count": 638
    }
  ]
}

```

GET /status/v1/services/pe-master

The `/status/v1/services/pe-master` endpoint returns JSON containing information about the routes that agents use to connect to this server.

You must query it at port 8140 and append the `level=debug` URL parameter.

Query parameters

No parameters are supported. Defaults to using the `critical` status level.

Response codes

The server uses the following response codes:

- 200 if and only if all services report a status of `running`
- 503 if any service's status is `unknown` or `error`

Response keys

The response's `experimental/http-metrics` section contains a list of routes, each containing the following keys:

Key	Definition
<code>aggregate</code>	The total time Puppet Server spent processing requests for this route.
<code>count</code>	The total number of requests Puppet Server processed for this route.
<code>mean</code>	The average time Puppet Server spent on each request for this route, calculated by dividing the aggregate value by the count.
<code>route-id</code>	The route being served. The request returns a route with the special <code>route-id</code> of <code>"total"</code> , which represents the aggregate data for all requests along all routes.

Routes for newer versions of Puppet Enterprise and newer agents are prefixed with `puppet-v3`, while Puppet Enterprise 3 agents' routes are not. For example, a PE 2017.3 `route-id` might be `puppet-v3-report-/*`, while the equivalent PE 3 agent's `route-id` is `:environment-report-/*`.

For example:

```
"pe-master": {
  {...},
  "status": {
    "experimental": {
      "http-metrics": [
        {
          "aggregate": 70668,
          "count": 234,
          "mean": 302,
          "route-id": "total"
        },
        {
          "aggregate": 28613,
          "count": 13,
          "mean": 2201,
          "route-id": "puppet-v3-catalog-*/"
        },
        {...}
      ]
    }
  }
}
```

GET /status/v1/services/pe-puppet-profiler

The `/status/v1/services/pe-puppet-profiler` endpoint returns JSON containing statistics about catalog compilation. You can use this data to discover which functions or resources are consuming the most resources or are most frequently used.

You must query it at port 8140 and append the `level=debug` URL parameter.

The Puppet Server profiler is enabled by default, but if it has been disabled, this endpoint's metrics are not available. Instead, the endpoint returns the same keys returned by a standard status endpoint request and an empty `status` key.

Query parameters

No parameters are supported. Defaults to using the `critical` status level.

Response codes

The server uses the following response codes:

- 200 if and only if all services report a status of `running`
- 503 if any service's status is `unknown` or `error`

Response keys

If the profiler is enabled, the response returns two subsections in the `experimental` section:

- `experimental/function-metrics`, containing statistics about functions evaluated by Puppet Server when compiling catalogs.
- `experimental/resource-metrics`, containing statistics about resources declared in manifests compiled by Puppet Server.

Each function measured in the `function-metrics` section also has a **function** key containing the function's name, and each resource measured in the `resource-metrics` section has a **resource** key containing the resource's name.

The two sections otherwise share these keys:

Key	Definition
<code>aggregate</code>	The total time spent handling this function call or resource during catalog compilation.
<code>count</code>	The number of times Puppet Server has called the function or instantiated the resource during catalog compilation.
<code>mean</code>	The average time spent handling this function call or resource during catalog compilation, calculated by dividing the <code>aggregate</code> value by the <code>count</code> .

For example:

```
"pe-puppet-profiler": {
  {...},
  "status": {
    "experimental": {
      "function-metrics": [
        {
          "aggregate": 1628,
          "count": 407,
          "function": "include",
          "mean": 4
        },
        {...},
      ]
      "resource-metrics": [
        {
          "aggregate": 3535,
          "count": 5,
          "mean": 707,
          "resource": "Class[Puppet_enterprise::Profile::Console]"
        },
        {...},
      ]
    }
  }
}
```



```

    }
  }
}

```

The metrics API

Puppet Enterprise includes an optional, enabled-by-default web endpoint for Java Management Extension (JMX) metrics, namely managed beans (MBeans).

These endpoints include:

- GET /metrics/v1/mbeans
- POST /metrics/v1/mbeans
- GET /metrics/v1/mbeans/<name>

Note: These API endpoints are a tech preview. The metrics described here are returned only when passing the `level=debug` URL parameter, and the structure of the returned data might change in future versions. To disable this endpoint, set `puppet_enterprise::master::puppetserver::metrics_webservice_enabled: false` in Hiera.

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 the 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.

For 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 contains 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"
}
```

Managing nodes

Common node management tasks include adding and removing nodes from your deployment, grouping and classifying nodes, and running Puppet on nodes. You can also deploy code to nodes using an environment-based testing workflow or the roles and profiles method.

Adding and removing agent nodes

After you install a Puppet agent on a node, accept its certificate signing request and begin managing it with Puppet Enterprise (PE). Or remove nodes that you no longer need.

Managing certificate signing requests

When you install a Puppet agent on a node, the agent automatically submits a certificate signing request (CSR) to the master. You must accept this request to bring before the node under PE management can be added your deployment. This allows Puppet to run on the node and enforce your configuration, which in turn adds node information to PuppetDB and makes the node available throughout the console.

You can approve certificate requests from the PE console or the command line. If DNS altnames are set up for agent nodes, you must approve the CSRs on use the command line interface .

Note: Specific user permissions are required to manage certificate requests:

- To accept or reject CSRs in the console or on the command line, you need the permission **Certificate requests: Accept and reject**.
- To manage certificate requests in the console, you also need the permission **Console: View**.

Related information

[Installing agents](#) on page 147

You can install Puppet Enterprise agents on *nix, Windows, and macOS.

Managing certificate signing requests in the console

Open certificate signing requests appear in the console on the **Unsigned certs** page. Accept or reject submitted requests individually or in a batch.

- To manage requests individually, click **Accept** or **Reject**.
- To manage the entire list of requests, click **Accept All** or **Reject All**. Nodes are processed in batches. If you close the browser window or navigate to another website while processing is in progress, only the current batch is processed.

The node appears in the PE console after the next Puppet run. To make a node available immediately after you approve the request, run Puppet on demand.

Related information

[Running Puppet on demand](#) on page 434

The orchestrator gives you the ability to set up jobs in the console or on the command line to trigger on-demand Puppet runs.

Managing certificate signing requests on the command line

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

To view pending node requests on the command line:

```
$ sudo puppetserver ca list
```

To sign a pending request:

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

To sign pending requests for nodes with DNS altnames:

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

Remove agent nodes

If you no longer wish to manage an agent node, you can remove it and make its license available for another node.

Removing a node:

- Deactivates the node in PuppetDB.
- Deletes the Puppet master's information cache for the node.
- Makes the license available for another node.
- Makes the hostname available for another node.

Note: Removing a node doesn't uninstall the agent from the node.

1. On the agent node, stop the agent service.

- Agent versions 4.0 or later: `service puppet stop`
- Agent versions earlier than 4.0: `service pe-puppet stop`

Note: To see which version of Puppet is installed on a node, run `puppet --version`.

2. On the master, purge the node: `puppet node purge <CERTNAME>`

The node's certificate is revoked, the certificate revocation list (CRL) is updated, and the node is deactivated in PuppetDB and removed from the console. The license is now available for another node. The node can't check in or re-register with PuppetDB on the next Puppet run.

3. If you have compilers, run Puppet on them: `puppet agent -t`

The updated CRL is managed by Puppet and distributed to compilers.

4. Optional: If the node you're removing was pinned to any node groups, you must manually unpin it from individual node groups or from all node groups using the `unpin-from-all` command endpoint.

Related information

[Uninstall infrastructure nodes](#) on page 177

The `puppet-enterprise-uninstaller` script is installed on the master. In order to uninstall, you must run the uninstaller on each infrastructure node.

[POST /v1/commands/unpin-from-all](#) on page 392

Use the `/v1/commands/unpin-from-all` to unpin specified nodes from all groups they're pinned to. Nodes that are dynamically classified using rules aren't affected.

Uninstall *nix agents

The *nix agent package includes an uninstall script, which you can use when you're ready to retire a node.

1. On the agent node, run the uninstall script: `run /opt/puppetlabs/bin/puppet-enterprise-uninstaller`
2. Follow prompts to uninstall.
3. (Optional) If you plan to reinstall on the node at a later date, remove the agent certificate for the agent from the master: `puppetserver ca clean <AGENT CERT NAME>`

Uninstall Windows agents

To uninstall the agent from a Windows node, use the Windows **Add or Remove Programs** interface, or uninstall from the command line.

Uninstalling the agent removes the Puppet program directory, the agent service, and all related registry keys. The data directory remains intact, including all SSL keys. To completely remove Puppet from the system, manually delete the data directory.

1. Use the Windows **Add or Remove Programs** interface to remove the agent.

Alternatively, you can uninstall from the command line if you have the original .msi file or know the product code of the installed MSI, for example: `msiexec /qn /norestart /x [puppet.msi|<PRODUCT_CODE>]`

2. (Optional) If you plan to reinstall on the node at a later date, remove the agent certificate for the agent from the master: `puppetserver ca clean <AGENT CERT NAME>`

Adding and removing agentless nodes

Using the inventory, you can manage nodes, including devices such as network switches or firewalls, without installing the Puppet agent on them. The inventory stores node and device information securely.

The inventory connects to agentless nodes through SSH or WinRM remote connections. The inventory uses transport definitions from installed device modules to connect to devices that can't have an agent installed on them.

After you add credentials to the inventory, authorized users can run tasks on these nodes and devices without re-entering credentials. On the **Tasks** page, these nodes and devices appear in the same list of targets as those that have agents installed.

Add agentless nodes to the inventory

Add nodes that don't have the Puppet agent installed to the inventory so you can run tasks on them.

Before you begin

Add classes to the **PE Master** node group for each agent platform used in your environment. For example, `pe_repo::platform::el_7_x86_64`.

Make sure you have the permission **Nodes: Add and delete connection information from inventory service**.

1. In the console, click **Inventory**.

2. Select the node type **Nodes without Puppet agents**.
3. Select a transport method.
 - **SSH** for *nix targets
 - **WinRM** for Windows targets
4. Enter target host names and the credentials required to access them. If you use an SSH key, include begin and end tags.
5. Optional: Select additional target options. For example, to add a target port number, select **Target Port** from the drop-down list, enter the number, and click **Add**. For details, see [Transport configuration options](#) on page 317, below.
6. Click **Add nodes**.

After the nodes have been added to the inventory, they are added to PuppetDB, and you can view them from the **Nodes** page. Nodes in the inventory can be added to an inventory node list when you set up a job to run tasks. To review each inventory node's connection options, or to remove the node from inventory, go to the **Connections** tab on the node's details page.

Transport configuration options

A list of transport configuration options for SSH and WinRM transports.

Target options for SSH transport

Option	Definition
Target port	Connection port. Default is 22.
Connection time-out in seconds	The length of time PE should wait when establishing connections.
Run as another user	After login, the user name to use for running commands.
Temporary directory	The directory to use when uploading temporary files to the target.
Sudo password	Password to use when changing users via <code>run-as</code> .
Process request as tty	Enable text terminal allocation.

Target options for WinRM transport

Option	Definition
Target port	Connection port. Default is 5986, or 5985 if <code>ssl : false</code>
Connection time-out in seconds	The length of time PE should wait when establishing connections.
Temporary directory	The directory to use when uploading temporary files to the target.
Acceptable file extension	List of file extensions that are accepted for scripts or tasks. Scripts with these file extensions rely on the target node's file type association to run. For example, if Python is installed on the system, a <code>.py</code> script should run with <code>python.exe</code> . The extensions <code>.ps1</code> , <code>.rb</code> , and <code>.pp</code> are always allowed and run via hard-coded executables.

Add devices to the inventory

If you have installed modules for device transports in your production environment, you can add connections to those devices to your inventory. This lets you manage network devices such as switches and firewalls, and run Puppet and task jobs on them, just like other agentless nodes in your infrastructure.

Before you begin

Make sure you have the permission **Nodes: Add and delete connection information from inventory service**.

The connection details differ for each type of device transport, as defined in the module; see the device module's README for details.



CAUTION: This initial implementation of network device configuration management is limited in how many device connections it can handle. Managing more than 100 devices can cause performance issues on the master. We plan to gather feedback from this release and improve the scaling and performance capabilities. In the meantime, avoid impacting master performance by limiting the number of network device connections you make to 100.

1. In the console, click **Inventory**.
2. Select the node type **Network devices**.
3. Select a device type from the list of device transports that you have installed as modules in your production environment.
4. Enter the device certname and other connection details, as defined in the transport module. Mandatory fields are marked with an asterisk. See the module README file if you need more details or examples specific to the transport.
5. Click **Add node**.

After devices have been added to the inventory, they are added to PuppetDB, and you can view them from the **Nodes** page. Devices in the inventory can be added to an inventory node list when you set up a job to run tasks. To review each inventory device's connection options, or to remove the device from inventory, go to the **Connections** tab on the device's node details page.

Remove agentless nodes and devices from the inventory

Remove an agentless node or device connection from the inventory from the **Connections** tab on its details page.

Before you begin

Make sure you have the permission **Nodes: Add and delete connection information from inventory service**.

1. On the **Overview** or **Nodes** page, find the node or device whose connection you want to remove, and click its name to open its details page.
2. Click **Connections**.
3. Click **Remove connection**. The exact name of the link varies depending on the connection type: **Remove SSH Connection**, **Remove WinRM connection**, or similar.
4. Confirm that you want to remove the connection.

When you remove a node or device connection from the inventory, PuppetDB marks it as expired after the standard node time-to-live (`node-ttl`) and then purges the node when it reaches its node-purge time-to-live limit (`node-purge-ttl`). At this point the node no longer appears in the console, and the node's license is available for use.

Tip: For more information about `node-ttl` and `node-purge-ttl` settings, see the PE docs for [database settings](#).

Related information

[Node inventory API](#) on page 410

These are the endpoints for the node inventory v1 API.

How nodes are counted

Your *node count* is the number of nodes in your inventory. Your license limits you to a certain number of active nodes before you hit your *bursting limit*. If you hit your bursting limit on four days during a month, you must purchase a license for more nodes or remove some nodes from your inventory.

Note: *Node* in this context includes agent nodes, agentless nodes, masters, compilers, nodes running in `noop` mode, and nodes that have been purged but had prior activity within the same calendar month.

Nodes included in the node count

The following nodes are included in your node count:

- Nodes with a report in PuppetDB during the calendar month.
- Nodes that have executed a Puppet run, task, or plan in the orchestrator, even if they do not have a report during the calendar month.

Nodes not included in the node count

The following nodes are not included in your node count:

- Nodes that are in the inventory service but are not used with Puppet runs, tasks, or plans.
- Nodes that have been purged and have no reports or activity within the calendar month.

Reaching the bursting limit

When you go above your node license's count limit, you enter what is called the *bursting limit*. The bursting limit lets you to exceed the number of nodes allowed under your license and enter a new threshold without extra charge. You are allowed to use your bursting limit on four consecutive or non-consecutive days per calendar month. Any higher usage beyond the four days will require you to either purge nodes or buy a license for more nodes.

The amount of time within your bursting limit does not matter for it to be counted as one day. For example, assuming you are licensed to use 1000 nodes and your bursting limit is 2000 nodes:

- If you use 1200 nodes for two hours one day, you have three days left to use your bursting limit that month.
- If you use 1900 nodes for 23 hours one day, you have three days left to use your bursting limit that month.
- If you use 1500 nodes for one hour each day for four days, you must contact your Puppet representative to buy a license for more nodes or purge some of your nodes until the next calendar month.

When nodes are counted

PE tracks node counts daily from 12:00 midnight UTC to 12:00 midnight UTC.

The same time is used for the calendar month. For example, the month of September would include activity from 12:00 midnight UTC 01 September until 12:00 midnight UTC 01 October. After that point, the bursting limit restarts with a fresh four days in October.

Viewing your node count

View your daily node count in the console by navigating to the **License** page and scrolling to the **Calendar month usage** section. This section also contains information about your subscription expiration date and license warnings, such as your license being expired or out of compliance.

To see daily node usage information on the command line, use the [Puppet orchestrator API: usage endpoint](#) on page 548.

Removing nodes

If you have unused nodes cluttering your inventory and are concerned about reaching your limit, read about removing them in [Adding and removing agent nodes](#) on page 314 and [Adding and removing agentless nodes](#) on page 316.

Related information

[Purchasing and installing a license key](#) on page 145

Your license must support the number of nodes that you want to manage with Puppet Enterprise.

Running Puppet on nodes

Puppet automatically attempts to run on each of your nodes every 30 minutes. To trigger a Puppet run outside of the default 30-minute interval, you can manually run Puppet.

In a Puppet run, the master and agent nodes perform the following actions:

1. The agent node sends facts to the master and requests a catalog.
2. The master compiles and returns the agent's catalog.
3. The agent applies the catalog by checking each resource the catalog describes. If it finds any resources that are not in the desired state, it makes the necessary changes.

Note: Puppet run behavior differs slightly if static catalogs are enabled.

Related information

[Static catalogs in Puppet Enterprise](#) on page 216

A catalog is a document that describes the desired state for each resource that Puppet manages on a node. A master typically compiles a catalog from manifests of Puppet code. A static catalog is a specific type of Puppet catalog that includes metadata that specifies the desired state of any file resources containing `source` attributes pointing to `puppet:///locations` on a node.

Running Puppet with the orchestrator

The Puppet orchestrator is a set of interactive tools used to deploy configuration changes when and how you want them. You can use the orchestrator to run Puppet from the console, command line, or API.

You can use the orchestrator to enforce change based on a:

- selection of nodes – from the console or the command line:

```
puppet job run --nodes <COMMA-SEPARATED LIST OF NODE NAMES>
```

- PQL nodes query – from the console or the command line, for example:

```
puppet job run --query 'nodes[certname] { facts {name = "operatingsystem"
and value = "Debian" } }'
```

- application or application instance - from the command line.

If you're putting together your own tools for running Puppet or want to enable CI workflows across your infrastructure, use the orchestrator API.

Related information

[Running Puppet on demand from the console](#) on page 434

When you set up a job to run Puppet from the console, the orchestrator creates a job ID to track the job, shows you all nodes included in the job, and runs Puppet on those nodes in the appropriate order. Puppet compiles a new catalog for each node included in the job.

Running Puppet with SSH

Run Puppet with SSH from an agent node.

SSH into the node and run `puppet agent --test` or `puppet agent -t`.

Running Puppet from the console

In the console, you can run Puppet from the node detail page for nodes that have an agent connection.

Run options include:

- **No-op** – Simulates changes without actually enforcing a new catalog. Nodes with `noop = true` in their `puppet.conf` files always run in no-op mode.
- **Debug** – Prints all messages available for use in debugging.
- **Trace** – Prints stack traces on some errors.
- **Evaltrace** – Shows a breakdown of the time taken for each step in the run.

When the run completes, the console displays the node's run status.

Note: The **Run Puppet** button is not available if an agent does not have an active websocket session with the PCP broker, or if the node's connection method is SSH, WinRM, or a network device transport.

Related information

[Node run statuses](#) on page 281

The **Overview** page displays the run status of each node following the most recent Puppet run. There are 10 possible run statuses.

Activity logging on console Puppet runs

When you initiate a Puppet run from the console, the Activity service logs the activity.

You can view activity on a single node by selecting the node, then clicking the **Activity** tab.

Alternatively, you can use the Activity Service API to retrieve activity information.

Related information

[Activity service API](#) on page 274

The activity service logs changes to role-based access control (RBAC) entities, such as users, directory groups, and user roles.

Troubleshooting Puppet run failures

Puppet creates a **View Report** link for most failed runs, which you can use to access events and logs.

This table shows some errors that can occur when you attempt to run Puppet, and suggestions for troubleshooting.

Error	Possible Cause
Changes could not be applied	Conflicting classes are a common cause. Check the log to get more detail.
Noop, changes could not be applied	Conflicting classes are a common cause. Check the log to get more detail.
Run already in progress	Occurs when a run is triggered in the command line or by another user, and you click Run .
Run request times out	Occurs if you click Run and the agent isn't available.
Report request times out	Occurs when the report is not successfully stored in PuppetDB after the run completes.
Invalid response (such as a 500 error)	Your Puppet code might be have incorrect formatting.
Button is disabled and a run is not allowed.	The user has permission, but the agent is not responding.

Grouping and classifying nodes

Configure nodes by assigning classes, parameters, and variables to them. This is called *classification*.

The main steps involved in classifying nodes are:

1. Create *node groups*.
2. Add nodes to groups, either manually or dynamically, with *rules*.
3. Assign classes to node groups.

Nodes can match the rules of many node groups. They receive classes, class parameters, and variables from all the node groups that they match.

How node group inheritance works

Node groups exist in a hierarchy of parent and child relationships. Nodes inherit classes, class parameters and variables, and rules from all ancestor groups.

- **Classes** – If an ancestor node group has a class, all descendent node groups also have the class.
- **Class parameters and variables** – Descendent node groups inherit class parameters and variables from ancestors unless a different value is set for the parameter or variable in the descendent node group.
- **Rules** – A node group can only match nodes that all of its ancestors also match. Specifying rules in a child node group is a way of narrowing down the nodes in the parent node group to apply classes to a specific subset of nodes.

Because nodes can match multiple node groups from separate hierarchical lineages, it's possible for two equal node groups to contribute conflicting values for variables and class parameters. Conflicting values cause a Puppet run on an agent to fail.

Tip: In the console, you can see how node groups are related on the **Classification** page, which displays a hierarchical view of node groups. From the command line, you can use the group children endpoint to review group lineage.

Related information

[GET /v1/group-children/:id](#) on page 397

Use the `/v1/group-children/:id` endpoint to retrieve a specified group and its descendents.

Best practices for classifying node groups

To organize node groups, start with the high-level functional groups that reflect the business requirements of your organization, and work down to smaller segments within those groups.

For example, if a large portion of your infrastructure consists of web servers, create a node group called `web servers` and add any classes that need to be applied to all web servers.

Next, identify subsets of web servers that have common characteristics but differ from other subsets. For example, you might have production web servers and development web servers. So, create a `dev web` child node group under the `web servers` node group. Nodes that match the `dev web` node group get all of the classes in the parent node group in addition to the classes assigned to the `dev web` node group.

Create node groups

You can create node groups to assign either an environment or classification.

- **Environment node groups** assign environments to nodes, such as test, development, or production.
- **Classification node groups** assign classification data to nodes, including classes, parameters, and variables.

Create environment node groups

Create custom environment node groups so that you can target deployment of Puppet code.

1. In the console, click **Classification**, and click **Add group**.

2. Specify options for the new node group and then click **Add**.

- **Parent name** – Select the top-level environment node group in your hierarchy. If you're using default environment node groups, this might be **Production environment** or **All environments**. Every environment node group you add must be a descendant of the top-level environment node group.
- **Group name** – Enter a name that describes the role of this environment node group, for example, Test environment.
- **Environment** – Select the environment that you want to assign to nodes that match this node group.
- **Environment group** – Select this option.

You can now add nodes to your environment node group to control which environment each node belongs to.

Create classification node groups

Create classification node groups to assign classification data to nodes.

1. In the console, click **Classification**, and click **Add group**.

2. Specify options for the new node group and then click **Add**.

- **Parent name** – Select the name of the classification node group that you want to set as the parent to this node group. Classification node groups inherit classes, parameters, and variables from their parent node group. By default, the parent node group is the **All Nodes** node group.
- **Group name** – Enter a name that describes the role of this classification node group, for example, Web Servers.
- **Environment** – Specify an environment to limit the classes and parameters available for selection in this node group.

Note: Specifying an environment in a classification node group does not assign an environment to any nodes, as it does in an environment node group.
- **Environment group** – Do not select this option.

You can now add nodes to your classification node group dynamically or statically.

Add nodes to a node group

There are two ways to add nodes to a node group.

- Individually pin nodes to the node group (static)
- Create rules that match node facts (dynamic)

Statically add nodes to a node group

If you have a node that needs to be in a node group regardless of the rules specified for that node group, you can pin the node to the node group.

A pinned node remains in the node group until you manually remove it. Adding a pinned node essentially creates the rule `<the certname of your node> = <the certname>`, and includes this rule along with the other fact-based rules.

1. In the console, click **Classification**, and then find the node group that you want to pin a node to and select it.
2. On the **Rules** tab, in the pinned nodes section, enter the certname of the node.
3. Click **Pin node**, and then commit changes.

Dynamically add nodes to a node group

Rules are the most powerful and scalable way to include nodes in a node group. You can create rules in a node group that are used to match node facts.

When nodes match the rules in a node group, they're classified with all of the classification data (classes, parameters, and variables) for the node group.

When nodes no longer match the rules of a node group, the classification data for that node group no longer applies to the node.

1. In the console, click **Classification**, and then find the node group that you want to add the rule to and select it.
2. On the **Rules** tab, specify rules for the fact, then click **Add rule**.
3. (Optional) Repeat step 2 as needed to add more rules.

Tip: If the node group includes multiple rules, be sure to specify whether **Nodes must match all rules** or **Nodes can match any rule**.

4. Commit changes.

Writing node group rules

To dynamically assign nodes to a group, you must specify rules based on node facts. Use this reference to fill out the **Rules** tab for node groups.

Option	Definition
Fact	<p>Specifies the fact used to match nodes.</p> <p>Select from the list of known facts, or enter part of a string to view fuzzy matches.</p> <p>To use structured or trusted facts, select the initial value from the dropdown list, then type the rest of the fact.</p> <ul style="list-style-type: none"> • To descend into a hash, use dots (".") to designate path segments, such as <code>os.release.major</code> or <code>trusted.certname</code>. • To specify an item in an array, surround the numerical index of the item in square brackets, such as <code>processors.models[0]</code> or <code>mountpoints./.options[0]</code>. • To identify path segments that contain dots or UTF-8 characters, surround the segment with single or double quotes, such as <code>trusted.extensions."1.3.6.1.4.1.34380.1.2.1"</code>. • To use trusted extension short names, append the short name after a second dot, such as <code>trusted.extensions.pp_role</code>. <p>Tip: Structured and trusted facts don't provide type-ahead suggestions beyond the top-level name key, and the facts aren't verified when entered. After adding a rule for a structured or trusted fact, review the number of matching nodes to verify that the fact was entered correctly.</p>

Option	Definition
Operator	<p>Describes the relationship between the fact and value.</p> <p>Operators include:</p> <ul style="list-style-type: none"> • = — is • != — is not • ~ — matches regex • !~ — does not match regex • > — greater than • >= — greater than or equal to • < — less than • <= — less than or equal to <p>The numeric operators >, >=, <, and <= can be used only with facts that have a numeric value.</p> <p>To match highly specific node facts, use ~ or !~ with a regular expression for Value.</p>
Value	Specifies the value associated with the fact.

Using structured and trusted facts for node group rules

Structured facts group a set of related facts, whereas trusted facts are a specific type of structured fact.

Structured facts group a set of related facts in the form of a hash or array. For example, the structured fact `os` includes multiple independent facts about the operating system, including architecture, family, and release. In the console, when you view facts about a node, you can differentiate structured facts because they're surrounded by curly braces.

Trusted facts are a specific type of structured fact where the facts are immutable and extracted from a node's certificate. Because they can't be changed or overridden, trusted facts enhance security by verifying a node's identity before sending sensitive data in its catalog.

You can use structured and trusted facts in the console to dynamically add nodes to groups.

Note: If you're using trusted facts to specify certificate extensions, in order for nodes to match to rules correctly, you must use short names for Puppet [registered IDs](#) and numeric IDs for [private extensions](#). Numeric IDs are required for private extensions whether or not you specify a short name in the `custom_trusted_oid_mapping.yaml` file.

Declare classes

Classes are the blocks of Puppet code used to configure nodes and assign resources to them.

Before you begin

The class that you want to apply must exist in an installed module. You can download modules from the Forge or create your own module.

1. In the console, click **Classification**, and then find the node group that you want to add the class to and select it.
2. On the **Configuration** tab, in the **Add new class** field, select the class to add.

The **Add new class** field suggests classes that the master knows about and that are available in the environment set for the node group.

3. Click **Add class** and then commit changes.

Note: Classes don't appear in the class list until they're retrieved from the master and the environment cache is refreshed. By default, both of these actions occur every three minutes. To override the default refresh period and force the node classifier to retrieve the classes from the master immediately, click the **Refresh** button.

Enable data editing in the console

The ability to edit configuration data in the console is enabled by default in new installations. If you upgrade from an earlier version and didn't previously have configuration data enabled, you must manually enable classifier configuration data, because enabling requires edits to your `hiera.yaml` file.

On your master, edit `/etc/puppetlabs/puppet/hiera.yaml` to add:

```
hierarchy:
- name: "Classifier Configuration Data"
  data_hash: classifier_data
```

Place any additional hierarchy entries, such as `hiera-yaml` or `hiera-eyaml` under the same hierarchy key, preferably below the `Classifier Configuration Data` entry.

Note: If you enable data editing in the console, add both **Set environment** and **Edit configuration data** to groups that set environment or modify class parameters in order for users to make changes.

If your environment is configured for high availability, you must also update `hiera.yaml` on your replica.

Define data used by node groups

The console offers multiple ways to specify data used in your manifests.

- **Configuration data** — Specify values through automatic parameter lookup.
- **Parameters** — Specify resource-style values used by a declared class.
- **Variables** — Specify values to make available in Puppet code as top-scope variables.

Set configuration data

Configuration data set in the console is used for automatic parameter lookup, the same way that Hieradata is used. Console configuration data takes precedence over Hieradata, but you can combine data from both sources to configure nodes.

Tip: In most cases, setting configuration data in the console is the more scalable and consistent method, but there are some cases where the console is preferable. Use the console to set configuration data if:

- You want to override `data`. Data set in the console overrides `data` when configured as recommended.
 - You want to give someone access to set or change data and they don't have the skill set to do it in `.`
 - You simply prefer the console user interface.
1. In the console, click **Classification**, then find the node group that you want to add configuration data to and select it.
 2. On the **Configuration** tab in the **Data** section, specify a **Class** and select a **Parameter** to add.

You can select from existing classes and parameters in the node group's environment, or you can specify free-form values. Classes aren't validated, but any class you specify must be present in the node's catalog at runtime in order for the parameter value to be applied.

When you select a parameter, the **Value** field is automatically populated with the inherited or default value.

3. Optional: Change the default parameter **Value**.

Related information

[Enable data editing in the console](#) on page 198

The ability to edit configuration data in the console is enabled by default in new installations. If you upgrade from an earlier version and didn't previously have configuration data enabled, you must manually enable classifier configuration data, because enabling requires edits to your `hiera.yaml` file.

Set parameters

Parameters are declared resource-style, which means they can be used to override other data; however, this override capability can introduce class conflicts and declaration errors that cause Puppet runs to fail.

1. In the console, click **Classification**, and then find the node group that you want to add a parameter to and select it.

2. On the **Configuration** tab, in the **Classes** section, select the class you want to modify and the **Parameter** to add.

The **Parameter** drop-down list shows all of the parameters that are available for that class in the node group's environment. When you select a parameter, the **Value** field is automatically populated with the inherited or default value.

3. (Optional) Change the default **Value**.

Set variables

Variables set in the console become top-scope variables available to all Puppet manifests.

1. In the console, click **Classification**, and then find the node group that you want to set a variable for and select it.
2. On the **Variables** tab, enter options for the variable:
 - **Key** – Enter the name of the variable.
 - **Value** – Enter the value that you want to assign to the variable.
3. Click **Add variable**, and then commit changes.

Tips for specifying parameter and variable values

Parameters and variables can be structured as JSON. If they can't be parsed as JSON, they're treated as strings.

Parameters and variables can be specified using these data types and syntax:

- Strings (for example, "centos")
 - Variable-style syntax, which interpolates the result of referencing a fact (for example, "I live at \$ipaddress.")
 - Expression-style syntax, which interpolates the result of evaluating the embedded expression (for example, \${\$os"release"})

Note: Strings must be double-quoted, because single quotes aren't valid JSON.

Tip: To enter a value in the console that contains a literal dollar sign, like a password hash — for example, \$1\$nnkkFwEc\$safFMXYaUVfKrDV4FLCm0/ — escape each dollar sign with a backslash to disable interpolation.

- Booleans (for example, true or false)
- Numbers (for example, 123)
- Hashes (for example, { "a" : 1 })

Note: Hashes must use colons rather than hash rockets.

- Arrays (for example, ["1" , "2.3"])

Variable-style syntax

Variable-style syntax uses a dollar sign (\$) followed by a Puppet fact name.

Example: "I live at \$ipaddress"

Variable-style syntax is interpolated as the value of the fact. For example, \$ipaddress resolves to the value of the ipaddress fact.

Important: The endpoint variable \$pe_node_groups cannot be interpolated when used as a classifier in class variable values.

Indexing cannot be used in variable-style syntax because the indices are treated as part of the string literal. For example, given the following fact: processors => { "count" => 4, "physicalcount" => 1 }, if you use variable-style syntax to specify \$processors[count], the value of the processors fact is interpolated but it is followed by a literal "[count]". After interpolation, this example becomes { "count" => 4, "physicalcount" => 1 }[count].

Note: Do not use the :: top-level scope indication because the console is not aware of Puppet variable scope.

Expression-style syntax

Use expression-style syntax when you need to index into a fact (`${os[release]}`), refer to trusted facts (`"My name is ${trusted[certname]}"`), or delimit fact names from strings (`"My ${os} release"`).

The following is an example of using expression-style syntax to access the full release number of an operating system:

```
${os"release"}
```

Expression-style syntax uses the following elements in order:

- an initial dollar sign and curly brace (`${}`)
- a legal Puppet fact name preceded by an optional dollar sign
- any number of index expressions (the quotations around indices are optional but are required if the index string contains spaces or square brackets)
- a closing curly brace (`}`)

Indices in expression-style syntax can be used to access individual fields of structured facts, or to refer to trusted facts. Use strings in an index if you want to access the keys of a hashmap. If you want to access a particular item or character in an array or string based on the order in which it is listed, you can use an integer (zero-indexed).

Examples of legal expression-style interpolation:

- `${os}`
- `${$os}`
- `${$os[release]}`
- `${$os['release']}`
- `${$os["release"]}`
- `${$os[2]}` (accesses the value of the third (zero-indexed) key-value pair in the `os` hash)
- `${$osrelease}` (accesses the value of the third key-value pair in the `release` hash)

In the console, an index can be only simple string literals or decimal integer literals. An index cannot include variables or operations (such as string concatenation or integer arithmetic).

Examples of illegal expression-style interpolation:

- `${$:os}`
- `{os[release]}`
- `${os[0xff]}`
- `${os[6/3]}`
- `${os[$family + release]}`
- `${os + release}`

Trusted facts

Trusted facts are considered to be keys of a hashmap called `trusted`. This means that all trusted facts must be interpolated using expression-style syntax. For example, the `certname` trusted fact would be expressed like this: `"My name is ${trusted[certname]}"`. Any trusted facts that are themselves structured facts can have further index expressions to access individual fields of that trusted fact.

Note: Regular expressions, resource references, and other keywords (such as `'undef'`) are not supported.

View nodes in a node group

To view all nodes that currently match the rules specified for a node group:

1. In the console, click **Classification**, and then find the node group that you want to view and select it.
2. Click **Matching nodes**.

You see the number of nodes that match the node group's rules, along with a list of the names of matching nodes. This is based on the facts collected during the node's last Puppet run. The matching nodes list is updated as rules are added, deleted, and edited. Nodes must match rules in ancestor node groups as well as the rules of the current node group in order to be considered a matching node.

Making changes to node groups

You can edit or remove node groups, remove nodes or classes from node groups, and edit or remove parameters and variables.

Edit or remove node groups

You can change the name, description, parent, environment, or environment group setting for node groups, or you can delete node groups that have no children.

1. In the console, click **Classification**, and select a node group.
2. At the top right of the page, select an option.
 - **Edit node group metadata** — Enables edit mode so you can modify the node group's metadata as needed.
 - **Remove node group** — Removes the node group. You're prompted to confirm the removal.
3. Commit changes.

Remove nodes from a node group

To remove dynamically-assigned nodes from a node group, edit or delete the applicable rule. To remove statically-assigned nodes from a node group, unpin them from the group.

Note: When a node no longer matches the rules of a node group, it is no longer classified with the classes assigned in that node group. However, the resources that were installed by those classes are not removed from the node. For example, if a node group has the `apache` class that installs the Apache package on matching nodes, the Apache package is not removed from the node even when the node no longer matches the node group rules.

1. In the console, click **Classification**, and select a node group.
2. On the **Rules** tab, select the option appropriate for the type of node.
 - **Dynamically-assigned nodes** — In the **Fact** table, click **Remove** to remove an individual rule, or click **Remove all rules** to delete all rules for the node group.
 - **Statically-assigned nodes** — In the **Certname** table, click **Unpin** to unpin an individual node, or click **Unpin all pinned nodes** to unpin all nodes from the node group.

Tip: To unpin a node from all groups it's pinned to, use the `unpin-from-all` command endpoint.

3. Commit changes.

Related information

[POST /v1/commands/unpin-from-all](#) on page 392

Use the `/v1/commands/unpin-from-all` to unpin specified nodes from all groups they're pinned to. Nodes that are dynamically classified using rules aren't affected.

Remove classes from a node group

Make changes to your node group by removing classes.

Note: If a class appears in a node group list but is crossed out, the class has been deleted from Puppet.

1. In the console, click **Classification**, and select a node group.
2. On the **Configuration** tab in the **Classes** section, click **Remove this class** to remove an individual class, or click **Remove all classes** to remove all classes from the node group.
3. Commit changes.

Edit or remove parameters

Make changes to your node group by editing or deleting the parameters of a class.

1. In the console, click **Classification**, and select a node group.
2. On the **Configuration** tab in the **Classes** section, for the class and parameter that you want to edit, select an option.
 - **Edit** — Enables edit mode so you can modify the parameter as needed.
 - **Remove** — Removes the parameter.
3. Commit changes.

Edit or remove variables

Make changes to your node group by editing or removing variables.

1. In the console, click **Classification**, and select a node group.
2. On the **Variables** tab, select an option.
 - **Edit** — Enables edit mode so you can modify the variable as needed.
 - **Remove** — Removes the variable.
 - **Remove all variables** — Removes all variables from the node group.
3. Commit changes.

Environment-based testing

An environment-based testing workflow is an effective approach for testing new code before pushing it to production.

Before testing and promoting data using an environment-based workflow, you must have configured:

- A test environment that's a child of the production environment
- Classification node groups that include nodes assigned dynamically or statically

Test and promote a parameter

Test and promote a parameter when using an environment-based testing workflow.

1. Create a classification node group with the test environment that is a child of a classification group that uses the production environment.
2. In the child group, set a test parameter. The test parameter overrides the value set by the parent group using the production environment.
3. If you're satisfied with your test results, manually change the parameter in the parent group.
4. (Optional) Delete the child test group.

Test and promote a class

Test and promote a class when using an environment-based testing workflow.

1. Create a classification node group with the test environment that is a child of a classification group that uses the production environment.

The node classifier validates your parameters against the test environment.

2. If you're satisfied with your test results, change the environment for the node group from test to production.

Testing code with canary nodes using alternate environments

Puppet Enterprise allows you to centrally manage which nodes are in which environments.

In most cases the environments are long-lived, such as development, testing, and production, and nodes don't move between these environments after their initial environment has been set.

When an agent node matches the rules specified in environment node groups, the agent is classified in that environment regardless of any environments specified in the agent's own `puppet.conf` file. Agents can't override this server-specified environment. That's the desired behavior in most cases.

A notable exception is when you want to test new Puppet code before deployment, and you have a code promotion workflow based on environments. In this case, you can specify that certain nodes are allowed to use an agent-specified environment. You map the agent-specified environment to a feature branch in your version control system. This override enables you to quickly test the code in your feature branch without permanently changing the environment that the node is in.

To apply an agent-specified environment for more than one run, specify the environment in the node's `puppet.conf` file. Doing this also sets the `agent_specified_environment` fact to `true`. The node will continue to get the agent-specified environment until you remove the environment from its `puppet.conf` file, or change the rules in the testing environment group.

Preconfigured node groups

Puppet Enterprise includes preconfigured node groups that are used to manage your configuration.

All Nodes node group

This node group is at the top of the hierarchy tree. All other node groups stem from this node group.

Classes

No default classes. Avoid adding classes to this node group.

Matching nodes

All nodes.

Notes

You can't modify the preconfigured rule that matches all nodes.

Infrastructure node groups

Infrastructure node groups are used to manage PE.

Important: Don't make changes to infrastructure node groups other than pinning new nodes for documented functions, like creating compilers. If you want to add custom classifications to infrastructure nodes, create new child groups and apply classification there.

PE Infrastructure node group

This node group is the parent to all other infrastructure node groups.

The **PE Infrastructure** node group contains data such as the hostnames and ports of various services and database info (except for passwords).

It's very important to correctly configure the `puppet_enterprise` class in this node group. The parameters set in this class affect the behavior of all other preconfigured node groups that use classes starting with `puppet_enterprise::profile`. Incorrect configuration of this class could potentially cause a service outage.



CAUTION: Never remove the **PE Infrastructure** node group. Removing the **PE Infrastructure** node group disrupts communication between all of your **PE Infrastructure** nodes.

Classes

`puppet_enterprise` — sets the default parameters for child node groups

Matching nodes

Nodes are not pinned to this node group. The **PE Infrastructure** node group is the parent to other infrastructure node groups, such as **PE Master**, and is used only to set classification that all child node groups inherit. Never pin nodes directly to this node group.

These are the parameters for the `puppet_enterprise` class, where `<YOUR HOST>` is your master certname. You can find the certname with `puppet config print certname`.

Parameter	Value
<code>database_host</code>	"<YOUR HOST> "
<code>puppetdb_host</code>	"<YOUR HOST> "
<code>database_port</code>	"<YOUR PORT NUMBER> " Required only if you changed the port number from the default 5432.
<code>database_ssl</code>	true if you're using the PE-installed PostgreSQL, and false if you're using your own PostgreSQL.
<code>puppet_master_host</code>	"<YOUR HOST> "
<code>certificate_authority_host</code>	"<YOUR HOST> "
<code>console_port</code>	"<YOUR PORT NUMBER> " Required only if you changed the port number from the default 443.)
<code>puppetdb_database_name</code>	"pe-puppetdb"
<code>puppetdb_database_user</code>	"pe-puppetdb"
<code>puppetdb_port</code>	"<YOUR PORT NUMBER> " Required only if you changed the port number from the default 8081.)
<code>console_host</code>	"<YOUR HOST> "
<code>pcp_broker_host</code>	"<YOUR HOST> "

PE Certificate Authority node group

This node group is used to manage the certificate authority.

Classes

`puppet_enterprise::profile::certificate_authority` — manages the certificate authority on the first master node

Matching nodes

On a new install, the master is pinned to this node group.

Notes

Don't add additional nodes to this node group.

PE Master node group

This node group is used to manage masters and add compilers.

Classes

- `puppet_enterprise::profile::master` — manages the Puppet master service

Matching nodes

On a new install, the master is pinned to this node group.

PE Compiler node group

This node group is a subset of the **PE Master** node group used to manage compilers running the PuppetDB service.

Classes

- `puppet_enterprise::profile::master` — manages the Puppet master service
- `puppet_enterprise::profile::puppetdb` — manages the PuppetDB service

Matching nodes

Compilers running the PuppetDB service are added to this node group.

Notes

Don't add additional nodes to this node group.

PE Orchestrator node group

This node group is used to manage the application orchestration service.

Classes

`puppet_enterprise::profile::orchestrator` — manages the application orchestration service

Matching nodes

On a new install, the master is pinned to this node group.

Notes

Don't add additional nodes to this node group.

PE PuppetDB node group

This node group is used to manage nodes running the PuppetDB service. If the node is also serving as a compiler, it's instead classified in the **PE Compiler** node group.

Classes

`puppet_enterprise::profile::puppetdb` — manages the PuppetDB service

Matching nodes

PuppetDB nodes that aren't functioning as compilers are pinned to this node group.

Notes

Don't add additional nodes to this node group.

PE Console node group

This node group is used to manage the console.

Classes

- `puppet_enterprise::profile::console` — manages the console, node classifier, and RBAC
- `puppet_enterprise::license` — manages the PE license file for the status indicator

Matching nodes

On a new install, the console server node is pinned to this node group.

Notes

Don't add additional nodes to this node group.

PE Agent node group

This group is used to manage the configuration of agents.

Classes

`puppet_enterprise::profile::agent` — manages your agent configuration

Matching nodes

All managed nodes are pinned to this node group by default.

PE Infrastructure Agent node group

This node group is a subset of the **PE Agent** node group used to manage infrastructure-specific overrides.

Classes

`puppet_enterprise::profile::agent` — manages your agent configuration

Matching nodes

All nodes used to run your Puppet infrastructure and managed by the PE installer are pinned to this node group by default, including the master, PuppetDB, console, and compilers.

Notes

You might want to manually pin to this group any additional nodes used to run your infrastructure, such as compiler load balancer nodes. Pinning a compiler load balancer node to this group allows it to receive its catalog from the master, rather than the compiler, which helps ensure availability.

PE Database node group

This node group is used to manage the PostgreSQL service.

Classes

- `puppet_enterprise::profile::database` — manages the PE-PostgreSQL service

Matching nodes

The node specified as `puppet_enterprise::database_host` is pinned to this group. By default, the database host is the PuppetDB server node.

Notes

Don't add additional nodes to this node group.

Environment node groups

Environment node groups are used only to set environments. They cannot contain any classification.

Preconfigured environment node groups differ depending on your version of PE, and you can customize environment groups as needed for your ecosystem.

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

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.

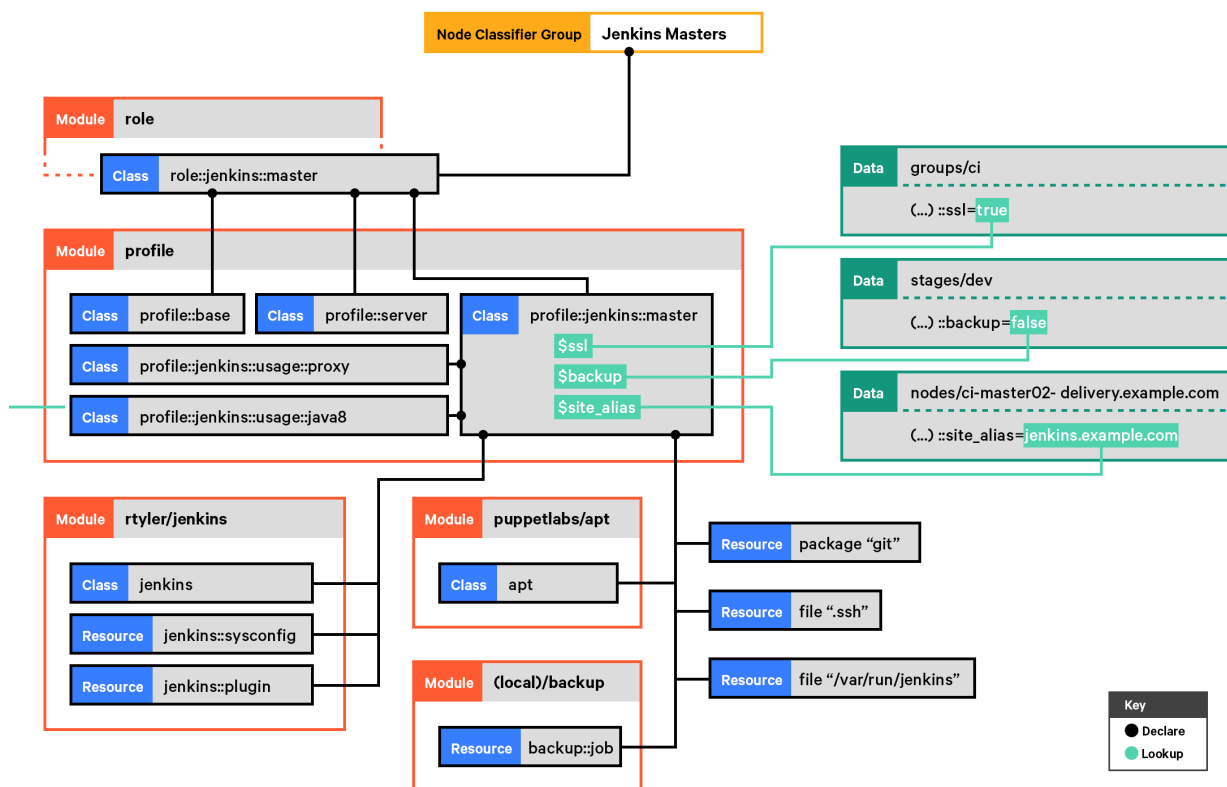
Configuring roles and profiles

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.



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 `rtyler/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 `rtyler/jenkins` module.

Jenkins requires Java, and the `rtyler` 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 [rtyler/jenkins](#), [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 336

There are rules for writing profile classes.

[Methods for data lookup](#) on page 336

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.

Tip: In most cases, setting configuration data in `data` is the more scalable and consistent method, but there are some cases where the console is preferable. Use the console to set configuration data if:

- You want to override `data`. Data set in the console overrides `data` when configured as recommended.
- You want to give someone access to set or change data and they don't have the skill set to do it in `data`.

- You simply prefer the console user interface.

```
# /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 336

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 336

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 -
+   Dhudson.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 336

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.

Node classifier API v1

These are the endpoints for the node classifier v1 API.

Tip: In addition to these endpoints, you can use the status API to check the health of the node classifier service.

Related information

[API index](#) on page 33

APIs allow you to interact with Puppet and Puppet Enterprise (PE) applications from your own code or application integration hooks.

Forming node classifier requests

Requests to the node classifier API must be well-formed HTTP(S) requests.

By default, the node classifier service listens on port 4433 and all endpoints are relative to the `/classifier-api/` path. For example, the full URL for the `/v1/groups` endpoint on localhost would be `https://localhost:4433/classifier-api/v1/groups`.

If needed, you can change the port the classifier API listens on.

Related information

[Configuring and tuning the console](#) on page 195

After installing Puppet Enterprise, you can change product settings to customize the console's behavior, adjust to your team's needs, and improve performance. Many settings can be configured in the console itself.

Authenticating to the node classifier API

You need to authenticate requests to the node classifier API. You can do this using RBAC authentication tokens or with the RBAC certificate whitelist.

Authentication token

You can make requests to the node classifier API using RBAC authentication tokens.

For detailed information about authentication tokens, see [token-based authentication](#).

In this example, we are using the `/groups` endpoint of the node classifier API to get a list of all groups that exist in the node classifier, along with their associated metadata. The example assumes that you have already generated a token and saved it as an environment variable using `export TOKEN=<PASTE THE TOKEN HERE>`.

```
curl -k -X GET https://<HOSTNAME>:<PORT>/classifier-api/v1/groups -H "X-Authentication:$TOKEN"
```

The example above uses the `X-Authentication` header to supply the token information. In some cases, such as GitHub webhooks, you might need to supply the token in a token parameter. To supply the token in a token parameter, you would specify the request like this:

```
curl -k -X GET "https://<HOSTNAME>:<PORT>/classifier-api/v1/groups?token=$TOKEN"
```

Note: Supplying the token as a token parameter is not as secure as using the `X-Authentication` method.

Whitelisted certificate

You can also authenticate requests using a certificate listed in RBAC's certificate whitelist. The RBAC whitelist is located at `/etc/puppetlabs/console-services/rbac-certificate-whitelist`. If you edit this file, you must reload the `pe-console-services` service for your changes to take effect (`sudo service pe-console-services reload`). You can attach the certificate using the command line as demonstrated in the example cURL query below. You must have the whitelist certificate name and the private key to run the script.

The following query returns a list of all groups that exist in the node classifier, along with their associated metadata. This query shows how to attach the whitelist certificate to authenticate the node classifier API.

In this query, the "whitelisted certname" needs to match a name in the file, `/etc/puppetlabs/console-services/rbac-certificate-whitelist`.

```
curl -X GET https://<HOSTNAME>:<PORT>/classifier-api/v1/groups \
  --cert /etc/puppetlabs/puppet/ssl/certs/<WHITELISTED CERTNAME>.pem \
  --key /etc/puppetlabs/puppet/ssl/private_keys/<WHITELISTED CERTNAME>.pem \
  --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem -H "Content-Type: application/json"
```

You do not need to use an agent certificate for authentication. You can use `puppet cert generate` to create a new certificate specifically for use with the API.

Related information

[Token-based authentication](#) on page 242

Puppet Enterprise users generate tokens to authenticate their access to certain PE command-line tools and API endpoints. Authentication tokens are tied to the permissions granted to the user through RBAC, and provide the user with the appropriate access to HTTP requests.

Using pagination parameters

If you have a large number of groups, classes, nodes, node check-ins, or environments, then sending a GET request through the classifier API could return an excessively large number of items.

To limit the number of items returned, you can use the `limit` and `offset` parameters.

- `limit`: The value of the `limit` parameter limits how many items are returned in a response. The value must be a non-negative integer. If you specify a value other than a non-negative integer, you get a 400 Bad Request error.
- `offset`: The value of the `offset` parameter specifies the number of item that are skipped. For example, if you specify an offset of 20 with a limit of 10, as shown in the example below, the first 20 items are skipped and you get back item 21 through to item 30. The value must be a non-negative integer. If you specify a value other than a non-negative integer, you get a 400 Bad Request error.

The following example shows a request using the `limit` and `offset` parameters.

```
curl https://<DNS NAME OF CONSOLE>:4433/classifier-api/v1/groups?
limit=10&offset=20 \
  --cert /etc/puppetlabs/puppet/ssl/certs/<WHITELISTED CERTNAME>.pem \
  --key /etc/puppetlabs/puppet/ssl/private_keys/<WHITELISTED CERTNAME>.pem \
  --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem \
  -H "Content-Type: application/json"
```

Groups endpoint

The `groups` endpoint is used to create, read, update, and delete groups.

A group belongs to an environment, applies classes (possibly with parameters), and matches nodes based on rules. Because groups are so central to the classification process, this endpoint is where most of the action is.

GET /v1/groups

Use the `/v1/groups` endpoint to retrieve a list of all node groups in the node classifier.

Query parameters

The request accepts these parameters.

Parameter	Value
<code>inherited</code>	If set to any value besides 0 or <code>false</code> , the node group includes the classes, class parameters, configuration data, and variables that it inherits from its ancestors.

Response format

The response is a JSON array of node group objects, using these keys:

Key	Definition
<code>name</code>	The name of the node group (a string).

Key	Definition
<code>id</code>	The node group's ID, which is a string containing a type-4 (random) UUID. The regular expression used to validate node group UUIDs is <code>[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}</code> .
<code>description</code>	An optional key containing an arbitrary string describing the node group.
<code>environment</code>	The name of the node group's environment (a string), which indirectly defines what classes are available for the node group to set, and is the environment that nodes classified into this node group run under.
<code>environment_trumps</code>	This is a boolean that changes the behavior of classifications that this node group is involved in. The default behavior is to return a classification-conflict error if the node groups that a node is classified into do not all have the same environment. If this flag is set, then this node group's environment overrides the environments of the other node groups (provided that none of them also have this flag set), with no attempt made to validate that the other node groups' classes and class parameters exist in this node group's environment.
<code>parent</code>	The ID of the node group's parent (a string). A node group is not permitted to be its own parent, unless it is the All Nodes group, which is the root of the hierarchy. Note that the root group always has the lowest-possible random UUID, 00000000-0000-4000-8000-000000000000.
<code>rule</code>	A boolean condition on node properties. When a node's properties satisfy this condition, it's classified into the node group.
<code>classes</code>	An object that defines both the classes consumed by nodes in this node group and any non-default values for their parameters. The keys of the object are the class names, and the values are objects describing the parameters. The parameter objects' keys are parameter names, and the values are what the node group sets for that parameter (always a string).
<code>config_data</code>	An object similar to the <code>classes</code> object that specifies parameters that are applied to classes if the class is assigned in the classifier or in puppet code. The keys of the object are the class names, and the values are objects describing the parameters. The parameter objects' keys are parameter names, and the values are what the group sets for that parameter (always a string). This feature is enabled/disabled via the <code>classifier::allow-config-data</code> setting. When set to false, this key is stripped from the payload.

Key	Definition
deleted	An object similar to the <code>classes</code> object that shows which classes and class parameters set by the node group have since been deleted. If none of the node group's classes or parameters have been deleted, this key is not present, so checking the presence of this key is an easy way to check whether the node group has references that need updating. The keys of this object are class names, and the values are further objects. These secondary objects always contain the <code>puppetlabs.classifier/deleted</code> key, whose value is a Boolean indicating whether the entire class has been deleted. The other keys of these objects are parameter names, and the other values are objects that always contain two keys: <code>puppetlabs.classifier/deleted</code> , mapping to a boolean indicating whether the specific class parameter has been deleted; and <code>value</code> , mapping to the string value set by the node group for this parameter (the value is duplicated for convenience, as it also appears in the <code>classes</code> object).
variables	An object that defines the values of any top-level variables set by the node group. The object is a mapping between variable names and their values (which can be any JSON value).
last_edited	The most recent time at which a user committed changes to a node group. This is a time stamp in ISO 8601 format, <code>YYYY-MM-DDTHH:MM:SSZ</code> .
serial_number	A number assigned to a node group. This number is incremented each time changes to a group are committed. <code>serial_number</code> is used to prevent conflicts when multiple users make changes to the same node group at the same time.

This example shows a node group object:

```
{
  "name": "Webservers",
  "id": "fc500c43-5065-469b-91fc-37ed0e500e81",
  "last_edited": "2018-02-20T02:36:17.776Z",
  "serial_number": 16,
  "environment": "production",
  "description": "This group captures configuration relevant to all web-
facing production webservers, regardless of location.",
  "parent": "00000000-0000-4000-8000-000000000000",
  "rule": ["and", ["~", ["trusted", "certname"], "www"],
    [">=", ["fact", "total_ram"], "512"]],
  "classes": {
    "apache": {
      "serveradmin": "bofh@travaglia.net",
      "keepalive_timeout": "5"
    }
  },
  "config_data": {
```

```

    "puppet_enterprise::profile::console": {"certname":
"console.example.com"},
    "puppet_enterprise::profile::puppetdb": {"listen_address": "0.0.0.0"}
  },
  "variables": {
    "ntp_servers": ["0.us.pool.ntp.org", "1.us.pool.ntp.org",
"2.us.pool.ntp.org"]
  }
}

```

This example shows a node group object that refers to some classes and parameters that have been deleted:

```

{
  "name": "Spaceship",
  "id": "fc500c43-5065-469b-91fc-37ed0e500e81",
  "last_edited": "2018-03-13T21:37:03.608Z",
  "serial_number": 42,
  "environment": "space",
  "parent": "00000000-0000-4000-8000-000000000000",
  "rule": ["=", ["fact", "is_spaceship"], "true"],
  "classes": {
    "payload": {
      "type": "cubesat",
      "count": "8",
      "mass": "10.64"
    },
    "rocket": {
      "stages": "3"
    }
  },
  "deleted": {
    "payload": {"puppetlabs.classifier/deleted": true},
    "rocket": {
      "puppetlabs.classifier/deleted": false,
      "stages": {
        "puppetlabs.classifier/deleted": true,
        "value": "3"
      }
    }
  },
  "variables": {}
}

```

The entire payload class has been deleted, because its deleted parameters object's `puppetlabs.classifier/deleted` key maps to `true`, in contrast to the `rocket` class, which has had only its `stages` parameter deleted.

Rule condition grammar

Nodes can be classified into groups using rules. This example shows how rule conditions must be structured:

```

condition  : [ {bool} {condition}+ ] | [ "not" {condition} ] |
{operation}
    bool    : "and" | "or"
    operation : [ {operator} {fact-path} {value} ]
    operator  : "=" | "~" | ">" | ">=" | "<" | "<="
    fact-path : {field-name} | [ {path-type} {field-name} {path-
component}+ ]
    path-type : "trusted" | "fact"
    path-component : field-name | number
    field-name : string

```

For the regex operator "~", the value is interpreted as a Java regular expression. Literal backslashes must be used to escape regex characters in order to match those characters in the fact value.

For the numeric comparison operators (">", ">=", "<", and "<="), the fact value (which is always a string) is coerced to a number (either integral or floating-point). If the value can't be coerced to a number, the numeric operation evaluates to false.

For the fact path, the rule can be either a string representing a top level field (the only current meaningful value here would be "name" representing the node name) or a list of strings and indices that represent looking up a field in a nested data structure. When passing a list of strings or indices, the first and second entries in the list must be strings. Subsequent entries can be indices.

Regular facts start with "fact" (for example, ["fact", "architecture"]) and trusted facts start with "trusted" (for example, ["trusted", "certname"]).

Error responses

`serial_number`

If you commit a node group with a `serial_number` that an API call has previously assigned, the service returns a 409 Conflict response.

POST /v1/groups

Use the `/v1/groups` endpoint to create a new node group without specifying its ID. When you use this endpoint, the node classifier service randomly generates an ID.

Request format

The request body must be a JSON object describing the node group to be created. The request uses these keys:

Key	Definition
<code>name</code>	The name of the node group (a string).
<code>environment</code>	The name of the node group's environment. This key is optional; if it's not provided, the default environment (<code>production</code>) is used.
<code>environment_trumps</code>	Whether this node group's environment should override those of other node groups at classification-time. This key is optional; if it's not provided, the default value of <code>false</code> is used.
<code>description</code>	A string describing the node group. This key is optional; if it's not provided, the node group has no description and this key doesn't appear in responses.
<code>parent</code>	The ID of the node group's parent (required).
<code>rule</code>	The condition that must be satisfied for a node to be classified into this node group
<code>variables</code>	An object that defines the names and values of any top-level variables set by the node group. The keys of the object are the variable names, and the corresponding value is that variable's value, which can be any sort of JSON value. The <code>variables</code> key is optional, and if a node group does not define any top-level variables then it may be omitted.

Key	Definition
<code>classes</code>	An object that defines the classes to be used by nodes in the node group. The <code>classes</code> key is required, and at minimum should be an empty object (<code>{}</code>). The <code>classes</code> key also contains the parameters for each class. Some classes have required parameters. This is a two-level object; that is, the keys of the object are class names (strings), and each key's value is another object that defines class parameter values. This innermost object maps between class parameter names and their values. The keys are the parameter names (strings), and each value is the parameter's value, which can be any kind of JSON value. The <code>classes</code> key is <i>not</i> optional; if it is missing, the service returns a 400Bad Request response.
<code>config_data</code>	An optional object that defines the class parameters to be used by nodes in the group. Its structure is the same as the <code>classes</code> object. If you use a <code>config_data</code> key but provide only the class, like <code>"config_data": { "qux": { } }</code> , no configuration data is stored. Note: This feature is enabled with the <code>classifier::allow-config-data</code> setting. When set to false, the presence of this key in the payload results in a 400 response.

Response format

If the node group was successfully created, the service returns a 303 See Other response, with the path to retrieve the created node group in the "location" header of the response.

Error responses

Responses and keys returned for create group requests depend on the type of error.

schema-violation

If any of the required keys are missing or the values of any of the defined keys do not match the required type, the service returns a 400 Bad Request response using these keys:

Key	Definition
<code>kind</code>	"schema-violation"
<code>details</code>	An object that contains three keys: <ul style="list-style-type: none"> <code>submitted</code> — Describes the submitted object. <code>schema</code> — Describes the schema that object is expected to conform to. <code>error</code> — Describes how the submitted object failed to conform to the schema.

malformed-request

If the request's body could not be parsed as JSON, the service returns a 400 Bad Request response using these keys:

Key	Definition
kind	"malformed-request"
details	<p>An object that contains two keys:</p> <ul style="list-style-type: none"> • <code>body</code> — Holds the request body that was received. • <code>error</code> — Describes how the submitted object failed to conform to the schema.

uniqueness-violation

If your attempt to create the node group violates uniqueness constraints (such as the constraint that each node group name must be unique within its environment), the service returns a 422 `Unprocessable Entity` response using these keys:

Key	Definition
kind	"uniqueness-violation"
msg	Describes which fields of the node group caused the constraint to be violated, along with their values.
details	<p>An object that contains two keys:</p> <ul style="list-style-type: none"> • <code>conflict</code> — An object whose keys are the fields of the node group that violated the constraint and whose values are the corresponding field values. • <code>constraintName</code> — The name of the database constraint that was violated.

missing-referents

If classes or class parameters defined by the node group, or inherited by the node group from its parent, do not exist in the submitted node group's environment, the service returns a 422 `Unprocessable Entity` response. In both cases the response object uses these keys:

Key	Definition
kind	"missing-referents"
msg	Describes the error and lists the missing classes or parameters.
details	<p>An array of objects, where each object describes a single missing referent, and has the following keys:</p> <ul style="list-style-type: none"> • <code>kind</code> — "missing-class" or "missing-parameter", depending on whether the entire class doesn't exist, or the class just doesn't have the parameter. • <code>missing</code> — The name of the missing class or class parameter. • <code>environment</code> — The environment that the class or parameter is missing from; that is, the environment of the node group where the error was encountered. • <code>group</code> — The name of the node group where the error was encountered. Due to inheritance, this might not be the group where the parameter was defined. • <code>defined_by</code> — The name of the node group that defines the class or parameter.

missing-parent

If the parent of the node group does not exist, the service returns a 422 `Unprocessable Entity` response. The response object uses these keys:

Key	Definition
kind	"missing-parent"
msg	Shows the parent UUID that did not exist.
details	The full submitted node group.

inheritance-cycle

If the request causes an inheritance cycle, the service returns a 422 `Unprocessable Entity` response. The response object uses these keys:

Key	Definition
kind	"inheritance-cycle"
details	An array of node group objects that includes each node group involved in the cycle
msg	A shortened description of the cycle, including a list of the node group names with each followed by its parent until the first node group is repeated.

GET /v1/groups/<id>

Use the `/v1/groups/\<id>` endpoint to retrieve a node group with the given ID.

Response format

The response is a JSON array of node group objects, using these keys:

Key	Definition
name	The name of the node group (a string).
id	The node group's ID, which is a string containing a type-4 (random) UUID. The regular expression used to validate node group UUIDs is <code>[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}</code> .
description	An optional key containing an arbitrary string describing the node group.
environment	The name of the node group's environment (a string), which indirectly defines what classes are available for the node group to set, and is the environment that nodes classified into this node group run under.

Key	Definition
<code>environment_trumps</code>	This is a boolean that changes the behavior of classifications that this node group is involved in. The default behavior is to return a classification-conflict error if the node groups that a node is classified into do not all have the same environment. If this flag is set, then this node group's environment overrides the environments of the other node groups (provided that none of them also have this flag set), with no attempt made to validate that the other node groups' classes and class parameters exist in this node group's environment.
<code>parent</code>	The ID of the node group's parent (a string). A node group is not permitted to be its own parent, unless it is the All Nodes group, which is the root of the hierarchy. Note that the root group always has the lowest-possible random UUID, 00000000-0000-4000-8000-000000000000.
<code>rule</code>	A boolean condition on node properties. When a node's properties satisfy this condition, it's classified into the node group.
<code>classes</code>	An object that defines both the classes consumed by nodes in this node group and any non-default values for their parameters. The keys of the object are the class names, and the values are objects describing the parameters. The parameter objects' keys are parameter names, and the values are what the node group sets for that parameter (always a string).
<code>deleted</code>	An object similar to the <code>classes</code> object that shows which classes and class parameters set by the node group have since been deleted. If none of the node group's classes or parameters have been deleted, this key is not present, so checking the presence of this key is an easy way to check whether the node group has references that need updating. The keys of this object are class names, and the values are further objects. These secondary objects always contain the <code>puppetlabs.classifier/deleted</code> key, whose value is a Boolean indicating whether the entire class has been deleted. The other keys of these objects are parameter names, and the other values are objects that always contain two keys: <code>puppetlabs.classifier/deleted</code> , mapping to a boolean indicating whether the specific class parameter has been deleted; and <code>value</code> , mapping to the string value set by the node group for this parameter (the value is duplicated for convenience, as it also appears in the <code>classes</code> object).

Key	Definition
<code>variables</code>	An object that defines the values of any top-level variables set by the node group. The object is a mapping between variable names and their values (which can be any JSON value).
<code>last_edited</code>	The most recent time at which a user committed changes to a node group. This is a time stamp in ISO 8601 format, <code>YYYY-MM-DDTHH:MM:SSZ</code> .
<code>serial_number</code>	A number assigned to a node group. This number is incremented each time changes to a group are committed. <code>serial_number</code> is used to prevent conflicts when multiple users make changes to the same node group at the same time.

Error responses

If the node group with the given ID cannot be found, the service returns `404 Not Found` and `malformed-uuid` responses. The body includes a generic 404 error response as described in the errors documentation.

`serial_number`

If you commit a node group with a `serial_number` that an API call has previously assigned, the service returns a `409 Conflict` response.

Related information

[Node classifier errors](#) on page 406

Familiarize yourself with error responses to make working the node classifier service API easier.

PUT /v1/groups/<id>

Use the `/v1/groups/<id>` to create a node group with the given ID.



CAUTION: Any existing node group with the given ID is overwritten.

It is possible to overwrite an existing node group with a new node group definition that contains deleted classes or parameters.

Request format

The request body must be a JSON object describing the node group to be created. The request uses these keys:

Key	Definition
<code>name</code>	The name of the node group (a string).
<code>environment</code>	The name of the node group's environment. This key is optional; if it's not provided, the default environment (<code>production</code>) is used.
<code>environment_trumps</code>	Whether this node group's environment should override those of other node groups at classification-time. This key is optional; if it's not provided, the default value of <code>false</code> is used.

Key	Definition
<code>description</code>	A string describing the node group. This key is optional; if it's not provided, the node group has no description and this key doesn't appear in responses.
<code>parent</code>	The ID of the node group's parent (required).
<code>rule</code>	The condition that must be satisfied for a node to be classified into this node group
<code>variables</code>	An object that defines the names and values of any top-level variables set by the node group. The keys of the object are the variable names, and the corresponding value is that variable's value, which can be any sort of JSON value. The <code>variables</code> key is optional, and if a node group does not define any top-level variables then it may be omitted.
<code>classes</code>	An object that defines the classes to be used by nodes in the node group. The <code>classes</code> key is required, and at minimum should be an empty object (<code>{}</code>). The <code>classes</code> key also contains the parameters for each class. Some classes have required parameters. This is a two-level object; that is, the keys of the object are class names (strings), and each key's value is another object that defines class parameter values. This innermost object maps between class parameter names and their values. The keys are the parameter names (strings), and each value is the parameter's value, which can be any kind of JSON value. The <code>classes</code> key is <i>not</i> optional; if it is missing, the service returns a 400Bad Request response.

Response format

If the node group is successfully created, the service returns a 201 `Created` response, with the node group object (in JSON) as the body. If the node group already exists and is identical to the submitted node group, then the service takes no action and returns a 200 `OK` response, again with the node group object as the body.

Error responses

If the requested node group object contains the `id` key, and its value differs from the UUID specified in the request's path, the service returns a 400 `Bad Request` response.

The response object uses these keys:

Key	Definition
<code>kind</code>	"conflicting-ids"
<code>details</code>	<p>An object that contains two keys:</p> <ul style="list-style-type: none"> <code>submitted</code> — Contains the ID submitted in the request body. <code>fromUrl</code> — Contains the ID taken from the request URL.

In addition, this operation can produce the general `malformed-error` response and any response that could also be generated by the POST group creation endpoint.

POST /v1/groups/<id>

Use the `/v1/groups/<id>` endpoint to update the name, environment, parent, rule, classes, class parameters, configuration data, and variables of the node group with the given ID by submitting a node group delta.

Request format

The request body must be JSON object describing the delta to be applied to the node group.

The `classes`, `config_data`, `variables`, and `rule` keys of the delta are merged with the node group, and then any keys of the resulting object that have a null value are deleted. This allows you to remove classes, class parameters, configuration data, variables, or the rule from the node group by setting them to null in the delta.

If the delta has a `rule` key that's set to a new value or nil, it's updated wholesale or removed from the group accordingly.

The `name`, `environment`, `description`, and `parent` keys, if present in the delta, replace the old values wholesale with their values.

The `serial_number` key is optional. If you update a node group and provide the `serial_number` in the payload, and the `serial_number` is not the current one for that group, the service returns a 409 Conflict response. To bypass this check, omit the `serial_number`.

Note that the root group's rule cannot be edited; any attempts to do so raise a 422 `Unprocessable Entity` response.

In the following examples, a delta is merged with a node group to update the group.

Node group:

```
{
  "name": "Webservers",
  "id": "58463036-0efa-4365-b367-b5401c0711d3",
  "environment": "staging",
  "parent": "00000000-0000-4000-8000-000000000000",
  "rule": [ "~", [ "trusted", "certname" ], "www" ],
  "classes": {
    "apache": {
      "serveradmin": "bofh@travaglia.net",
      "keepalive_timeout": 5
    },
    "ssl": {
      "keystore": "/etc/ssl/keystore"
    }
  },
  "variables": {
    "ntp_servers": [ "0.us.pool.ntp.org", "1.us.pool.ntp.org",
      "2.us.pool.ntp.org" ]
  }
}
```

Delta:

```
{
  "name": "Production Webservers",
  "id": "58463036-0efa-4365-b367-b5401c0711d3",
  "environment": "production",
  "parent": "01522c99-627c-4a07-b28e-a25dd563d756",
  "classes": {
    "apache": {
      "serveradmin": "roy@reynholm.co.uk",
```

```

    "keepalive_timeout": null
  },
  "ssl": null
},
"variables": {
  "dns_servers": ["dns.reynholm.co.uk"]
}
}

```

Updated group:

```

{
  "name": "Production Webservers",
  "id": "58463036-0efa-4365-b367-b5401c0711d3",
  "environment": "production",
  "parent": "01522c99-627c-4a07-b28e-a25dd563d756",
  "rule": ["~", ["trusted", "certname"], "www"],
  "classes": {
    "apache": {
      "serveradmin": "roy@reynholm.co.uk"
    }
  },
  "variables": {
    "ntp_servers": ["0.us.pool.ntp.org", "1.us.pool.ntp.org",
    "2.us.pool.ntp.org"],
    "dns_servers": ["dns.reynholm.co.uk"]
  }
}

```

Note that the `ssl` class was deleted because its entire object was mapped to null, whereas for the `apache` class only the `keepalive_timeout` parameter was deleted.

Deleted classes and class parameters

If the node group definition contains classes and parameters that have been deleted it is still possible to update the node group with those parameters and classes. Updates that don't increase the number of errors associated with a node group are allowed.

Error responses

The response object uses these keys:

Key	Definition
kind	"conflicting-ids"
details	<p>An object that contains two keys:</p> <ul style="list-style-type: none"> <code>submitted</code> — Contains the ID submitted in the request body. <code>fromUrl</code> — Contains the ID taken from the request URL.

If the requested node group object contains the `id` key, and its value differs from the UUID specified in the request's path, the service returns a `400 Bad Request` response.

In addition, this operation can produce the general `malformed-error` response and any response that could also be generated by the POST group creation endpoint.

422 responses to POST requests can include errors that were caused by the node group's children, but a node group being created with a PUT request cannot have any children.

DELETE /v1/groups/<id>

Use the `/v1/groups/\<id\>` endpoint to delete the node group with the given ID.

Response format

If the delete operation is successful, the sever returns a `204 No Content` response.

Error responses

In addition to the general `malformed-uuid` response, if the node group with the given ID does not exist, the service returns a `404 Not Found` response, as described in the errors documentation.

children-present

The service returns a `422 Unprocessable Entity` and rejects the delete request if the node group that is being deleted has children.

The response object uses these keys:

Key	Definition
<code>kind</code>	"children-present"
<code>msg</code>	Explains why the delete was rejected and names the children.
<code>details</code>	Contains the node group in question along with all of its children.

Related information

[Node classifier errors](#) on page 406

Familiarize yourself with error responses to make working the node classifier service API easier.

POST /v1/groups/<id>/pin

Use the `/v1/groups/<id>/pin` endpoint to pin nodes to the group with the given ID.

Request format

You can provide the names of the nodes to pin in two ways.

- As the value of the `nodes` query parameter. For multiple nodes, use a comma-separated list format.

For example:

```
POST /v1/groups/58463036-0efa-4365-b367-b5401c0711d3/pin?nodes=foo%2Cbar%2Cbaz
```

This request pins the nodes `foo`, `bar`, and `baz` to the group.

- In the body of a request. In the `nodes` field of a JSON object, specify the node name as the value. For multiple nodes, use a JSON array.

For example:

```
{"nodes": ["foo", "bar", "baz"]}
```

This request pins the nodes `foo`, `bar`, and `baz` to the group.

It's easier to use the query parameter method. However, if you want to affect a large number of nodes at once, the query string might get truncated. Strings are truncated if they exceed 8,000 characters. In such cases, use the second method. The request body in the second method is allowed to be many megabytes in size.

Response format

If the pin is successful, the service returns a `204 No Content` response with an empty body.

Error responses

This endpoint uses the following error responses.

If you don't supply the `nodes` query parameter or a request body, the service returns a `400 Malformed Request` response, using these keys:

Key	Definition
<code>kind</code>	"missing-parameters"
<code>msg</code>	Explains the missing <code>nodes</code> query parameter.

If you supply a request body that is not valid JSON, the service returns a `400 Malformed Request` response. The response object uses these keys:

Key	Definition
<code>kind</code>	"malformed-request"
<code>details</code>	An object with a <code>body</code> key containing the body as received by the service, and an <code>error</code> field containing a string describing the error encountered when trying to parse the request's body.

If the request's body is valid JSON, but the payload is not an object with just the `nodes` field and no other fields, the service returns a `400 Malformed Request` response. The response object uses these keys:

Key	Definition
<code>kind</code>	"schema-violation"
<code>msg</code>	Describes the difference between what was submitted and the required format.

POST /v1/groups/<id>/unpin

Use the `/v1/groups/<id>/unpin` endpoint to unpin nodes from the group with the given ID.

Note: Nodes not actually pinned to the group can be specified without resulting in an error.

Request format

You can provide the names of the nodes to pin in two ways.

- As the value of the `nodes` query parameter. For multiple nodes, use a comma-separated list format.

For example:

```
POST /v1/groups/58463036-0efa-4365-b367-b5401c0711d3/unpin?nodes=foo%2Cbar%2Cbaz
```

This request unpins the nodes `foo`, `bar`, and `baz` from the group. If any of the specified nodes were not pinned to the group, they are ignored.

- In the body of a request. In the `nodes` field of a JSON object, specify the node name as the value. For multiple nodes, use a JSON array.

For example:

```
{"nodes": ["foo", "bar", "baz"]}
```

This request unpins the nodes `foo`, `bar`, and `baz` from the group. If any of the specified nodes were not pinned to the group, they are ignored.

It's easier to use the query parameter method. However, if you want to affect a large number of nodes at once, the query string might get truncated. Strings are truncated if they exceed 8,000 characters. In such cases, use the second method. The request body in the second method is allowed to be many megabytes in size.

Response format

If the unpin is successful, the service returns a 204 `No Content` response with an empty body.

Error responses

If you don't supply the `nodes` query parameter or a request body, the service returns a 400 `Malformed Request` response. The response object uses these keys:

Key	Definition
<code>kind</code>	"missing-parameters"
<code>msg</code>	Explains the missing <code>nodes</code> query parameter.

If a request body is supplied but it is not valid JSON, the service returns a 400 `Malformed Request` response. The response object uses these keys:

Key	Definition
<code>kind</code>	"malformed-request"
<code>details</code>	An object with a <code>body</code> key containing the body as received by the service, and an <code>error</code> field containing a string describing the error encountered when trying to parse the request's body.

If the request's body is valid JSON, but the payload is not an object with just the `nodes` field and no other fields, the service returns a 400 `Malformed Request` response. The response object uses these keys:

Key	Definition
<code>kind</code>	"schema-violation"
<code>msg</code>	Describes the difference between what was submitted and the required format.

GET /v1/groups/:id/rules

Use `/v1/groups/:id/rules` to resolve the rules for the requested group and then translate those rules to work with the nodes and inventory endpoints in PuppetDB.

Response format

A successful response includes these keys:

rule

The rules for the group in classifier format.

rule_with_inherited

The inherited rules (including the rules for this group) in classifier format

translated

An object containing two children, each of the inherited rules translated into a different format.

nodes_query_format

The optimized translated inherited group in the format that works with the nodes endpoint in PuppetDB.

inventory_query_format

The optimized translated inherited group in the format that works with the inventory endpoint in PuppetDB.

```
{
  "rule": [
    "=",
    [
      "fact",
      "is_spaceship"
    ],
    "true"
  ],
  "rule_with_inherited": [
    "and",
    [
      "=",
      [
        "fact",
        "is_spaceship"
      ],
      "true"
    ],
    [
      "~",
      "name",
      ".*"
    ]
  ],
  "translated": {
    "nodes_query_format": [
      "or",
      [
        "=",
        [
          "fact",
          "is_spaceship"
        ],
        "true"
      ],
      [
        "=",
        [
          "fact",
          "is_spaceship",
          true
        ]
      ]
    ],
    "inventory_query_format": [
      "or",
      [
        "=",
        "facts.is_spaceship",
        "true"
      ],
      [
        "=",
        "facts.is_spaceship",
        true
      ]
    ]
  ]
}
```

```
}
```

Error responses

In addition to the general `malformed-uuid` error response, if the group with the given ID cannot be found, a 404 Not Found response is returned.

Related information

[Error response description](#) on page 406

Errors from the node classifier service are JSON responses.

Groups endpoint examples

Use example requests to better understand how to work with groups in the node classifier API.

These requests assume the following configuration:

- The Puppet master is running on `puppetlabs-nc.example.vm` with access to certificates and whitelisting to enable URL requests.
- Port 4433 is open.

Create a group called My Nodes

This request uses `POST /v1/groups` to create a group called *My Nodes*.

```
curl -X POST -H 'Content-Type: application/json' \
  --cert /etc/puppetlabs/puppet/ssl/certs/puppetlabs-nc.example.vm.pem \
  --key /etc/puppetlabs/puppet/ssl/private_keys/puppetlabs-nc.example.vm.pem \
  --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem \
  -d '{ "name": "My Nodes",
        "parent": "00000000-0000-4000-8000-000000000000",
        "environment": "production",
        "classes": {}
      }' \
  https://puppetlabs-nc.example.vm:4433/classifier-api/v1/groups
```

Related information

[POST /v1/groups](#) on page 365

Use the `/v1/groups` endpoint to create a new node group without specifying its ID. When you use this endpoint, the node classifier service randomly generates an ID.

Get the group ID of My Nodes

This request uses the groups endpoint to get details about groups.

```
curl 'https://puppetlabs-nc.example.vm:4433/classifier-api/v1/groups' \
  -H "Content-Type: application/json" \
  --cert /etc/puppetlabs/puppet/ssl/certs/puppetlabs-nc.example.vm.pem \
  --key /etc/puppetlabs/puppet/ssl/private_keys/puppetlabs-nc.example.vm.pem \
  --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem
```

The response is a JSON file containing details about all groups, including the *My Nodes* group ID.

```
{
  "environment_trumps": false,
  "parent": "00000000-0000-4000-8000-000000000000",
  "name": "My Nodes",
  "variables": {},
  "id": "085e2797-32f3-4920-9412-8e9decf4ef65",
  "environment": "production",
  "classes": {}
}
```

```
}
```

Related information

[Groups endpoint](#) on page 361

The groups endpoint is used to create, read, update, and delete groups.

Pin a node to the My Nodes group

This request uses POST `/v1/groups/<id>` to pin the node *example-to-pin.example.vm* to *My Groups* using the group ID retrieved in the previous step.

```
curl -X POST -H 'Content-Type: application/json' \
  --cert /etc/puppetlabs/puppet/ssl/certs/puppetlabs-nc.example.vm.pem \
  --key /etc/puppetlabs/puppet/ssl/private_keys/puppetlabs-nc.example.vm.pem \
  --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem \
  -d '{ "nodes": ["example-to-pin.example.vm"] }' \
  https://puppetlabs-nc.example.vm:4433/classifier-api/v1/groups/085e2797-32f3-4920-9412-8e9decf4ef65/pin
```

Related information

[POST /v1/groups/<id>](#) on page 372

Use the `/v1/groups/<id>` endpoint to update the name, environment, parent, rule, classes, class parameters, configuration data, and variables of the node group with the given ID by submitting a node group delta.

Pin a second node to the My Nodes group

This request uses POST `/v1/groups/<id>` to pin a second node *example-to-pin-2.example.vm* to *My Groups*.

Note: You must supply all the nodes to pin to the group. For example, this request includes both *example-to-pin.example.vm* and *example-to-pin-2.example.vm*.

```
curl -X POST -H 'Content-Type: application/json' \
  --cert /etc/puppetlabs/puppet/ssl/certs/puppetlabs-nc.example.vm.pem \
  --key /etc/puppetlabs/puppet/ssl/private_keys/puppetlabs-nc.example.vm.pem \
  --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem \
  -d '{ "nodes": ["example-to-pin-2.example.vm"] }' \
  https://puppetlabs-nc.example.vm:4433/classifier-api/v1/groups/085e2797-32f3-4920-9412-8e9decf4ef65/pin
```

Related information

[POST /v1/groups/<id>](#) on page 372

Use the `/v1/groups/<id>` endpoint to update the name, environment, parent, rule, classes, class parameters, configuration data, and variables of the node group with the given ID by submitting a node group delta.

Unpin a node from the My Nodes group

This request uses POST `/v1/groups/<id>` to unpin the node *example-to-unpin.example.vm* from *My Groups*.

```
curl -X POST -H 'Content-Type: application/json' \
  --cert /etc/puppetlabs/puppet/ssl/certs/puppetlabs-nc.example.vm.pem \
  --key /etc/puppetlabs/puppet/ssl/private_keys/puppetlabs-nc.example.vm.pem \
  --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem \
  -d '{ "nodes": ["example-to-unpin.example.vm"] }' \
  https://puppetlabs-nc.example.vm:4433/classifier-api/v1/groups/085e2797-32f3-4920-9412-8e9decf4ef65/unpin
```

Add a class and parameters to the My Nodes group

This request uses POST `/v1/groups/<id>` to specify the *apache* class and parameters for *My Groups*.

```
curl -X POST -H 'Content-Type: application/json' \
  --cert /etc/puppetlabs/puppet/ssl/certs/puppetlabs-nc.example.vm.pem.pem \
  --key /etc/puppetlabs/puppet/ssl/private_keys/puppetlabs-nc.example.vm.pem.pem \
  --cacert /etc/puppetlabs/puppet/ssl/certs/ca.pem \
  -d '{ "classes": { "apache": { "serveradmin": "roy@reynholm.co.uk", "keepalive_timeout": null } } }' \
  https://puppetlabs-nc.example.vm.pem:4433/classifier-api/v1/groups/085e2797-32f3-4920-9412-8e9decf4ef65
```

Related information

[POST /v1/groups/<id>](#) on page 372

Use the `/v1/groups/<id>` endpoint to update the name, environment, parent, rule, classes, class parameters, configuration data, and variables of the node group with the given ID by submitting a node group delta.

Classes endpoint

Use the `classes` endpoints to retrieve lists of classes, including classes with specific environments. The output from this endpoint is especially useful for creating new node groups, which usually contain a reference to one or more classes.

The node classifier gets its information about classes from Puppet, so don't use this endpoint to create, update, or delete classes.

GET /v1/classes

Use the `/v1/classes` endpoint to retrieve a list of all classes known to the node classifier.

Note: All other operations on classes require using the environment-specific endpoints.

All `/classes` endpoints return the classes *currently known* to the node classifier, which retrieves them periodically from the master. To force an update, use the `update_classes` endpoint. To determine when classes were last retrieved from the master, use the `last_class_update` endpoint.

Response format

The response is a JSON array of class objects. Each class object contains these keys:

Key	Definition
<code>name</code>	The name of the class (a string).
<code>environment</code>	The name of the environment that this class exists in. Note that the same class can exist in different environments and can have different parameters in each environment.
<code>parameters</code>	An object describing the parameters and default parameter values for the class. The keys of this object are the parameter names (strings). Each value is the default value for the associated parameter as a string, boolean, number, structured value, or null. If the value is null, the parameter is required.

This is an example of a class object:

```
{
  "name": "apache",
```

```

    "environment": "production",
    "parameters": {
        "default_mods": true,
        "default_vhost": true,
        ...
    }
}

```

GET /v1/environments/<environment>/classes

Use the `/v1/environments/<environment>/classes` to retrieve a list of all classes known to the node classifier within the given environment.

Response format

The response is a JSON array of class objects. Each class object contains these keys:

Key	Definition
name	The name of the class (a string).
environment	The name of the environment that this class exists in. Note that the same class can exist in different environments and can have different parameters in each environment.
parameters	An object describing the parameters and default parameter values for the class. The keys of this object are the parameter names (strings). Each value is the default value for the associated parameter as a string, boolean, number, structured value, or null. If the value is null, the parameter is required.

Error responses

This request does not produce error responses.

GET /v1/environments/<environment>/classes/<name>

Use the `/v1/environments/<environment>/classes/<name>` endpoint to retrieve the class with the given name in the given environment.

Response format

If the class exists, the response is a JSON array of class objects. Each class object contains these keys:

Key	Definition
name	The name of the class (a string).
environment	The name of the environment that this class exists in. Note that the same class can exist in different environments and can have different parameters in each environment.

Key	Definition
parameters	An object describing the parameters and default parameter values for the class. The keys of this object are the parameter names (strings). Each value is the default value for the associated parameter as a string, boolean, number, structured value, or null. If the value is null, the parameter is required.

Error responses

If the class with the given name cannot be found, the service returns a 404 `Not Found` response with an empty body.

Classification endpoint

The `classification` endpoint takes a node name and a set of facts, and returns information about how that node is classified. The output can help you test your classification rules.

POST /v1/classified/nodes/<name>

Use the `/v1/classified/nodes/\<name\>` endpoint to retrieve the classification information for the node with the given name and facts as supplied in the body of the request.

Request format

The request body can contain a JSON object describing the facts and trusted facts of the node to be classified. The object can have these keys:

Key	Definition
fact	The regular, non-trusted facts of the node. The value of this key is a further object, whose keys are fact names, and whose values are the fact values. Fact values can be a string, number, boolean, array, or object.
trusted	The trusted facts of the node. The values of this key are subject to the same restrictions as those on the value of the <code>fact</code> key.

Response format

The response is a JSON object describing the node post-classification, using these keys:

Key	Definition
name	The name of the node (a string).
groups	An array of the group-ids (strings) that this node was classified into.
environment	The name of the environment that this node uses, which is taken from the node groups the node was classified into.
classes	An object where the keys are class names and the values are objects that map parameter names to values.
parameters	An object where the keys are top-level variable names and the values are the values assigned to those variables.

This is an example of a response from this endpoint:

```
{
  "name": "foo.example.com",
  "groups": [ "9c0c7d07-a199-48b7-9999-3cdf7654e0bf", "96d1a058-225d-48e2-
a1a8-80819d31751d" ],
  "environment": "staging",
  "parameters": {},
  "classes": {
    "apache": {
      "keepalive_timeout": 30,
      "log_level": "notice"
    }
  }
}
```

Error responses

If the node is classified into multiple node groups that define conflicting classifications for the node, the service returns a 500 Server Error response.

The body of this response contains the usual JSON error object described in the errors documentation.

The `kind` key of the error is "classification-conflict", the `msg` describes generally why this happens, and the `details` key contains an object that describes the specific conflicts encountered.

The details object can have these keys:

Key	Definition
environment	Maps directly to an array of value detail objects (described below).
variables	Contains an object with a key for each conflicting variable, whose values are an array of value detail objects.
classes	Contains an object with a key for each class that had conflicting parameter definitions, whose values are further objects that describe the conflicts for that class's parameters.

A value details object describes one of the conflicting values defined for the environment, a variable, or a class parameter. Each object contains these keys:

Key	Definition
value	The defined value, which is a string for environment and class parameters, but for a variable can be any JSON value.
from	The node group that the node was classified into that caused this value to be added to the node's classification. This group cannot define the value, because it can be inherited from an ancestor of this group.
defined_by	The node group that actually defined this value. This is often the <code>from</code> group, but could instead be an ancestor of that group.

This example shows a classification conflict error object with node groups truncated for clarity:

```
{
  "kind": "classification-conflict",
  "msg": "The node was classified into multiple unrelated node groups that
  defined conflicting class parameters or top-level variables. See `details`
  for a list of the specific conflicts.",
  "details": {
    "classes": {
      "songColors": {
        "blue": [
          {
            "value": "Blue Suede Shoes",
            "from": {
              "name": "Elvis Presley",
              "classes": {},
              "rule": ["=", "nodename", "the-node"],
              ...
            },
            "defined_by": {
              "name": "Carl Perkins",
              "classes": {"songColors": {"blue": "Blue Suede Shoes"}},
              "rule": ["not", ["=", "nodename", "the-node"]],
              ...
            }
          },
          {
            "value": "Since You've Been Gone",
            "from": {
              "name": "Aretha Franklin",
              "classes": {"songColors": {"blue": "Since You've Been Gone"}},
              ...
            },
            "defined_by": {
              "name": "Aretha Franklin",
              "classes": {"songColors": {"blue": "Since You've Been Gone"}},
              ...
            }
          }
        ]
      }
    }
  }
}
```

In this example, the conflicting "Blue Suede Shoes" value was included in the classification because the node matched the "Elvis Presley" node group (because that is the value of the "from" key), but that node group doesn't define the "Blue Suede Shoes" value. That value is defined by the "Carl Perkins" node group, which is an ancestor of the "Elvis Presley" node group, causing the latter to inherit the value from the former. The other conflicting value, "Since You've Been Gone", is defined by the same node group that the node matched.

Related information

[Node classifier errors](#) on page 406

Familiarize yourself with error responses to make working the node classifier service API easier.

POST /v1/classified/nodes/<name>/explanation

Use the /v1/classified/nodes/<name>/explanation endpoint to retrieve an explanation of how a node is classified by submitting its facts.

Request format

The body of the request must be a JSON object describing the node's facts. This object uses these keys:

Key	Definition
fact	Describes the regular (non-trusted) facts of the node. Its value must be a further object whose keys are fact names, and whose values are the corresponding fact values. Structured facts can be included here; structured fact values are further objects or arrays.
trusted	Optional key that describes the trusted facts of the node. Its value has exactly the same format as the <code>fact</code> key's value.
name	The node's name from the request's URL is merged into this object under this key.

This is an example of a valid request body:

```
{
  "fact": {
    "ear-tips": "pointed",
    "eyebrow pitch": "40",
    "hair": "dark",
    "resting bpm": "120",
    "blood oxygen transporter": "hemocyanin",
    "anterior tricuspid": "2",
    "appendices": "1",
    "spunk": "10"
  }
}
```

Response format

The response is a JSON object that describes how the node would be classified.

- If the node would be successfully classified, this object contains the final classification.
- If the classification would fail due to conflicts, this object contains a description of the conflicts.

This response is intended to provide insight into the entire classification process, so that if a node isn't classified as expected, you can trace the deviation.

Classification proceeds in this order:

1. All node group rules are tested on the node's facts and name, and groups that don't match the node are culled, leaving the matching groups.
2. Inheritance relations are used to further cull the matching groups, by removing any matching node group that has a descendant that is also a matching node group. Those node groups that are left over are *leaf groups*.
3. Each leaf group is transformed into its inherited classification by adding all the inherited values from its ancestors.
4. All of the inherited classifications and individual node classifications are inspected for conflicts. A conflict occurs whenever two inherited classifications define different values for the environment, a class parameter, or a top-level variable.
5. Any individual node classification, including classes, class parameters, configuration data, and variables, is added.
6. Individual node classification is applied to the group classification, forming the final classification, which is then returned to the client.

The JSON object returned by this endpoint uses these keys:

Key	Definition
<code>match_explanations</code>	Corresponds to step 1 of classification, finding the matching node groups. This key's value is an explanation object just like those found in node check-ins, which maps between a matching group's ID and an explained condition object that demonstrates why the node matched that group's rule.
<code>leaf_groups</code>	Corresponds to step 2 of classification, finding the leaves. This key's value is an array of the leaf groups (that is, those groups that are not related to any of the other matching groups).
<code>inherited_classifications</code>	Corresponds to step 3 of classification, adding inherited values. This key's value is an object mapping from a leaf group's ID to the classification values provided by that group (after inheritance).
<code>conflicts</code>	Corresponds to step 4 of classification. This key is present only if there are conflicts between the inherited classifications. Its value is similar to a classification object, but wherever there was a conflict there's an array of conflict details instead of a single classification value. Each of these details is an object with three keys: <code>value</code> , <code>from</code> , and <code>defined_by</code> . The <code>value</code> key is a conflicting value, the <code>from</code> key is the group whose inherited classification provided this value, and the <code>defined_by</code> key is the group that actually defined the value (which can be an ancestor of the <code>from</code> group).
<code>individual_classification</code>	Corresponds to step 5 of classification. Its value includes classes, class parameters, configuration data, and variables applied directly during this stage.
<code>final_classification</code>	Corresponds to step 6 of classification, present only if there are no conflicts between the inherited classifications. Its value is the result of merging all individual node classification and group classification.
<code>node_as_received</code>	The submitted node object as received by the service, after adding the name and, if not supplied by the client, an empty <code>trusted</code> object. This key does not correspond to any of the classification steps.
<code>classification_source</code>	An annotated version of the classification that has the environment and every class parameter and variable replaced with an "annotated value" object. This key does not correspond to any of the classification steps.

This example shows a response the endpoint could return in the case of a successful classification:

```
{
  "node_as_received": {
    "name": "Tuvok",
    "trusted": {},
    "fact": {
      "ear-tips": "pointed",
      "eyebrow pitch": "30",
      "blood oxygen transporter": "hemocyanin",
      "anterior tricuspid": "2",

```

```

    "hair": "dark",
    "resting bpm": "200",
    "appendices": "0",
    "spunk": "0"
  }
},
"match_explanations": {
  "000000000-0000-4000-8000-0000000000000": {
    "value": true,
    "form": ["~", {"path": "name", "value": "Tuvok"}, ".*"]
  },
  "8aeeb640-8dca-4b99-9c40-3b75de6579c2": {
    "value": true,
    "form": ["and",
      {
        "value": true,
        "form": [">=", {"path": ["fact", "eyebrow pitch"], "value":
"30"}, "25"]
      },
      {
        "value": true,
        "form": ["=", {"path": ["fact", "ear-tips"], "value":
"pointed"}, "pointed"]
      },
      {
        "value": true,
        "form": ["=", {"path": ["fact", "hair"], "value": "dark"},
"dark"]
      },
      {
        "value": true,
        "form": [">=", {"path": ["fact", "resting bpm"], "value":
"200"}, "100"]
      },
      {
        "value": true,
        "form": ["=",
          {
            "path": ["fact", "blood oxygen transporter"],
            "value": "hemocyanin"
          },
          "hemocyanin"
        ]
      }
    ]
  }
},
"leaf_groups": {
  "8aeeb640-8dca-4b99-9c40-3b75de6579c2": {
    "name": "Vulcans",
    "id": "8aeeb640-8dca-4b99-9c40-3b75de6579c2",
    "parent": "000000000-0000-4000-8000-0000000000000",
    "rule": ["and", [">=", ["fact", "eyebrow pitch"], "25"],
      ["=", ["fact", "ear-tips"], "pointed"],
      ["=", ["fact", "hair"], "dark"],
      [">=", ["fact", "resting bpm"], "100"],
      ["=", ["fact", "blood oxygen transporter"],
"hemocyanin"]
    ],
    "environment": "alpha-quadrant",
    "variables": {},
    "classes": {
      "emotion": {"importance": "ignored"},
      "logic": {"importance": "primary"}
    }
  }
}

```

```

    },
    "config_data": {
      "USS::Voyager": {"designation": "subsequent"}
    }
  },
  "inherited_classifications": {
    "8aeeb640-8dca-4b99-9c40-3b75de6579c2": {
      "environment": "alpha-quadrant",
      "variables": {},
      "classes": {
        "logic": {"importance": "primary"},
        "emotion": {"importance": "ignored"}
      },
      "config_data": {
        "USS::Enterprise": {"designation": "original"},
        "USS::Voyager": {"designation": "subsequent"}
      }
    }
  },
  "individual_classification": {
    "classes": {
      "emotion": {
        "importance": "secondary"
      }
    },
    "variables": {
      "full_name": "S'chn T'gai Spock"
    }
  },
  "final_classification": {
    "environment": "alpha-quadrant",
    "variables": {
      "full_name": "S'chn T'gai Spock"
    },
    "classes": {
      "logic": {"importance": "primary"},
      "emotion": {"importance": "secondary"}
    },
    "config_data": {
      "USS::Enterprise": {"designation": "original"},
      "USS::Voyager": {"designation": "subsequent"}
    }
  },
  "classification_sources": {
    "environment": {
      "value": "alpha-quadrant",
      "sources": ["8aeeb640-8dca-4b99-9c40-3b75de6579c2"]
    },
    "variables": {},
    "classes": {
      "emotion": {
        "puppetlabs.classifier/sources":
["8aeeb640-8dca-4b99-9c40-3b75de6579c2"],
        "importance": {
          "value": "secondary",
          "sources": ["node"]
        }
      },
      "logic": {
        "puppetlabs.classifier/sources":
["8aeeb640-8dca-4b99-9c40-3b75de6579c2"],
        "importance": {
          "value": "primary",

```

```

    "sources": [ "8aeeb640-8dca-4b99-9c40-3b75de6579c2" ]
  },
  "config_data": {
    "USS::Enterprise": {
      "designation": {
        "value": "original",
        "sources": [ "00000000-0000-4000-8000-000000000000" ]
      }
    },
    "USS::Voyager": {
      "designation": {
        "value": "subsequent",
        "sources": [ "8aeeb640-8dca-4b99-9c40-3b75de6579c2" ]
      }
    }
  }
}
}
}
}
}

```

This example shows a response that resulted from conflicts:

```

{
  "node_as_received": {
    "name": "Spock",
    "trusted": {},
    "fact": {
      "ear-tips": "pointed",
      "eyebrow pitch": "40",
      "blood oxygen transporter": "hemocyanin",
      "anterior tricuspid": "2",
      "hair": "dark",
      "resting bpm": "120",
      "appendices": "1",
      "spunk": "10"
    }
  },
  "match_explanations": {
    "00000000-0000-4000-8000-000000000000": {
      "value": true,
      "form": [ "~", { "path": "name", "value": "Spock" }, ".*" ]
    },
    "a130f715-c929-448b-82cd-fe21d3f83b58": {
      "value": true,
      "form": [ ">=", { "path": [ "fact", "spunk" ], "value": "10" }, "5" ]
    },
    "8aeeb640-8dca-4b99-9c40-3b75de6579c2": {
      "value": true,
      "form": [ "and",
        {
          "value": true,
          "form": [ ">=", { "path": [ "fact", "eyebrow pitch" ], "value":
"30" }, "25" ]
        },
        {
          "value": true,
          "form": [ "=", { "path": [ "fact", "ear-tips" ], "value":
"pointed" }, "pointed" ]
        }
      ],
      "value": true,

```

```

        "form": ["=", {"path": ["fact", "hair"], "value": "dark"}],
"dark"]
    },
    {
        "value": true,
        "form": [">=", {"path": ["fact", "resting bpm"], "value":
"200"}, "100"]
    },
    {
        "value": true,
        "form": ["=",
            {
                "path": ["fact", "blood oxygen transporter"],
                "value": "hemocyanin"
            },
            "hemocyanin"
        ]
    }
}
]
},
"leaf_groups": {
    "a130f715-c929-448b-82cd-fe21d3f83b58": {
        "name": "Humans",
        "id": "a130f715-c929-448b-82cd-fe21d3f83b58",
        "parent": "00000000-0000-4000-8000-000000000000",
        "rule": [">=", ["fact", "spunk"], "5"],
        "environment": "alpha-quadrant",
        "variables": {},
        "classes": {
            "emotion": {"importance": "primary"},
            "logic": {"importance": "secondary"}
        }
    },
    "8aeeb640-8dca-4b99-9c40-3b75de6579c2": {
        "name": "Vulcans",
        "id": "8aeeb640-8dca-4b99-9c40-3b75de6579c2",
        "parent": "00000000-0000-4000-8000-000000000000",
        "rule": ["and", [">=", ["fact", "eyebrow pitch"],
"25"],
            ["=", ["fact", "ear-tips"], "pointed"],
            ["=", ["fact", "hair"], "dark"],
            [">=", ["fact", "resting bpm"], "100"],
            ["=", ["fact", "blood oxygen
transporter"], "hemocyanin"]
        ],
        "environment": "alpha-quadrant",
        "variables": {},
        "classes": {
            "emotion": {"importance": "ignored"},
            "logic": {"importance": "primary"}
        }
    }
},
"inherited_classifications": {
    "a130f715-c929-448b-82cd-fe21d3f83b58": {
        "environment": "alpha-quadrant",
        "variables": {},
        "classes": {
            "logic": {"importance": "secondary"},
            "emotion": {"importance": "primary"}
        }
    },
    "8aeeb640-8dca-4b99-9c40-3b75de6579c2": {

```

```

    "environment": "alpha-quadrant",
    "variables": {},
    "classes": {
      "logic": {"importance": "primary"},
      "emotion": {"importance": "ignored"}
    }
  },
  "conflicts": {
    "classes": {
      "logic": {
        "importance": [
          {
            "value": "secondary",
            "from": {
              "name": "Humans",
              "id": "a130f715-c929-448b-82cd-fe21d3f83b58",
              ...
            },
            "defined_by": {
              "name": "Humans",
              "id": "a130f715-c929-448b-82cd-fe21d3f83b58",
              ...
            }
          },
          {
            "value": "primary",
            "from": {
              "name": "Vulcans",
              "id": "8aeeb640-8dca-4b99-9c40-3b75de6579c2",
              ...
            },
            "defined_by": {
              "name": "Vulcans",
              "id": "8aeeb640-8dca-4b99-9c40-3b75de6579c2",
              ...
            }
          }
        ]
      },
      "emotion": {
        "importance": [
          {
            "value": "ignored",
            "from": {
              "name": "Vulcans",
              "id": "8aeeb640-8dca-4b99-9c40-3b75de6579c2",
              ...
            },
            "defined_by": {
              "name": "Vulcans",
              "id": "8aeeb640-8dca-4b99-9c40-3b75de6579c2",
              ...
            }
          },
          {
            "value": "primary",
            "from": {
              "name": "Humans",
              "id": "a130f715-c929-448b-82cd-fe21d3f83b58",
              ...
            },
            "defined_by": {
              "name": "Humans",

```

```

        "id": "a130f715-c929-448b-82cd-fe21d3f83b58",
        ...
      }
    }
  ]
},
"individual_classification": {
  "classes": {
    "emotion": {
      "importance": "secondary"
    }
  },
  "variables": {
    "full_name": "S'chn T'gai Spock"
  }
}
}

```

Commands endpoint

Use the commands endpoint to unpin specified nodes from all groups they're pinned to.

POST /v1/commands/unpin-from-all

Use the /v1/commands/unpin-from-all to unpin specified nodes from all groups they're pinned to. Nodes that are dynamically classified using rules aren't affected.

Nodes are unpinned from only those groups for which you have view and edit permissions. Because group permissions are applied hierarchically, you must have **Create, edit, and delete child groups** or **Edit child group rules** permissions for the parent groups of the groups you want to unpin the node from.

Note: As number of groups increases, response time can increase significantly with the unpin-from-all endpoint.

Request format

The request body must be a JSON object describing the nodes to unpin, using the following key:

Key	Definition
nodes	The certname of the nodes (required).

For example:

```
{ "nodes": [ "foo", "bar" ] }
```

Response format

If unpinning is successful, the service returns a list of nodes with the groups they were unpinned from. If a node wasn't pinned to any groups, it's not included in the response.

```

{ "nodes": [ { "name": "foo",
  "groups": [ { "id": "8310b045-c244-4008-88d0-b49573c84d2d",
    "name": "Webservers",
    "environment": "production" },
    { "id": "84b19b51-6db5-4897-9409-a4a3a94b7f09",
    "name": "Test",
    "environment": "test" } ] },
  { "name": "bar",
    "groups": [ { "id": "84b19b51-6db5-4897-9409-a4a3a94b7f09",

```



```
"name": "Test",
"environment": "test" ] ] ] }
```

Assuming token is set as an environment variable, the example below unpins "host1.example" and "host2.example":

```
curl -k -H "X-Authentication:$TOKEN" \
-H "Content-Type: application/json" \
https://<DNS NAME OF CONSOLE>:4433/classifier-api/v1/commands/unpin-
from-all -X POST \
-d '{"nodes": ["host1.example", "host2.example"]}'
```

The classifier responds with information about each group it is removed from:

```
{ "nodes":
  [ { "name": "host1.example", "groups": [ { "id": "2d83d860-19b4-4f7b-8b70-
e5ee4d8646db", "name": "test", "environment": "production" } ] },
    { "name": "host2.example", "groups": [ { "id": "2d83d860-19b4-4f7b-8b70-
e5ee4d8646db", "name": "test", "environment": "production" } ] } ] }
```

Environments endpoint

Use the `environments` endpoint to retrieve information about environments in the node classifier. The output tells you either which environments are available or whether a named environment exists. The output can be helpful when creating new node groups, which must be associated with an environment. The node classifier gets its information about environments from Puppet, so do not use this endpoint to create, update, or delete them.

GET /v1/environments

Use the `/v1/environments` endpoint to retrieve a list of all environments known to the node classifier.

Response format

The response is a JSON array of environment objects, using the following keys:

Key	Definition
<code>name</code>	The name of the environment (a string).
<code>sync_succeeded</code>	Whether the environment synched successfully during the last class synchronization (a Boolean).

Error responses

No error responses specific to this request are expected.

GET /v1/environments/<name>

Use the `/v1/environments/<name>` endpoint to retrieve the environment with the given name. The main use of this endpoint is to check if an environment actually exists.

Response format

If the environment exists, the service returns a 200 response with an environment object in JSON format.

Error responses

If the environment with the given name cannot be found, the service returns a 404: Not Found response with an empty body.

PUT /v1/environments/<name>

Use the `/v1/environments/\<name\>` endpoint to create a new environment with the given name.

Request format

No further information is required in the request besides the name portion of the URL.

Response format

If the environment is created successfully, the service returns a `201: Created` response whose body is the environment object in JSON format.

Error responses

No error responses specific to this operation are expected.

Nodes check-in history endpoint

Use the `nodes` endpoint to retrieve historical information about nodes that have checked into the node classifier.

Enable check-in storage to use this endpoint

Node check-in storage is disabled by default because it can place excessive loads on larger deployments. You must enable node check-in storage before using the check-in history endpoint. If node check-in storage is not enabled, the endpoint returns an empty array.

To enable node check-in storage, set the `classifier_node_check_in_storage` parameter in the `puppet_enterprise::profile::console` class to `true`.

Related information

[Set configuration data](#) on page 326

Configuration data set in the console is used for automatic parameter lookup, the same way that Hieradata is used. Console configuration data takes precedence over Hieradata, but you can combine data from both sources to configure nodes.

GET /v1/nodes

Use the `/v1/nodes` endpoint to retrieve a list of all nodes that have checked in with the node classifier, each with their check-in history.

Query Parameters**limit**

Controls the maximum number of nodes returned. `limit=10` returns 10 nodes.

offset

Specifies how many nodes to skip before the first returned node. `offset=20` skips the first 20 nodes.



CAUTION: In deployments with large numbers of nodes, a large or unspecified `limit` might cause the console-services process to run out of memory and crash.

Response format

The response is a JSON array of node objects. Each node object contains these two keys:

Key	Definition
<code>name</code>	The name of the node according to Puppet (a string).
<code>check_ins</code>	An array of check-in objects (described below).

Each check-in object describes a single check-in of the node. The check-in objects have the following keys:

Key	Definition
time	The time of the check-in as a string in ISO 8601 format (with timezone).
explanation	An object mapping between IDs of groups that the node was classified into and explained condition objects that describe why the node matched this group's rule.
transaction_uuid	A uuid representing a particular Puppet transaction that is submitted by Puppet at classification time. This makes it possible to identify the check-in involved in generating a specific catalog and report.

The explained condition objects are the node group's rule condition marked up with the node's value and the result of evaluation. Each form in the rule (that is, each array in the JSON representation of the rule condition) is replaced with an object that has two keys:

Key	Definition
value	A Boolean that is the result of evaluating this form. At the top level, this is the result of the entire rule condition, but because each sub-condition is marked up with its value, you can use this to understand, say, which parts of an <code>or</code> condition were true.
form	The condition form, with all sub-forms as further explained condition objects.

Besides the condition markup, the comparison operations of the rule condition have their first argument (the fact path) replaced with an object that has both the fact path and the value that was found in the node at that path.

The following example shows the format of an explained condition.

Start with a node group with the following rule:

```
[ "and", [ ">=", [ "fact", "pressure hulls" ], "1" ],
          [ "=", [ "fact", "warp cores" ], "0" ],
          [ ">=", [ "fact", "docking ports" ], "10" ] ]
```

The following node checks into the classifier:

```
{
  "name": "Deep Space 9",
  "fact": {
    "pressure hulls": "10",
    "docking ports": "18",
    "docking pylons": "3",
    "warp cores": "0",
    "bars": "1"
  }
}
```

When the node checks in for classification, it matches the above rule, so that check-in's explanation object has an entry for the node group that the rule came from. The value of this entry is this explained condition object:

```
{
  "value": true,
  "form": [
    "and",
    {
```

```

      "value": true,
      "form": [ ">=", { "path": ["fact", "pressure hulls"], "value": "3"} ],
    "1" ]
  },
  {
    "value": true,
    "form": [ "=", { "path": ["fact", "warp cores"], "value": "0"} ], "0" ]
  },
  {
    "value": true,
    "form": [ ">" { "path": ["fact", "docking ports"], "value": "18"} ], "9" ]
  }
]
}

```

GET /v1/nodes/<node>

Use the /v1/nodes/<node> endpoint to retrieve the check-in history for only the specified node.

Response format

The response is one node object as described above in the GET /v1/nodes documentation, for the specified node. The following example shows a node object:

```

{
  "name": "Deep Space 9",
  "check_ins": [
    {
      "time": "2369-01-04T03:00:00Z",
      "explanation": {
        "53029cf7-2070-4539-87f5-9fc754a0f041": {
          "value": true,
          "form": [
            "and",
            {
              "value": true,
              "form": [ ">=", { "path": ["fact", "pressure hulls"], "value":
"3"} ], "1" ]
            },
            {
              "value": true,
              "form": [ "=", { "path": ["fact", "warp cores"], "value": "0"} ],
"0" ]
            },
            {
              "value": true,
              "form": [ ">" { "path": ["fact", "docking ports"], "value":
"18"} ], "9" ]
            }
          ]
        }
      }
    ],
    "transaction_uuid": "d3653a4a-4ebe-426e-a04d-dbebec00e97f"
  }
}

```

Error responses

If the specified node has not checked in, the service returns a 404: Not Found response, with the usual JSON error response in its body.

Group children endpoint

Use the group children endpoint to retrieve a specified group and its descendents.

GET /v1/group-children/:id

Use the `/v1/group-children/:id` endpoint to retrieve a specified group and its descendents.

Request format

The request body must be a JSON object specifying a group and an optional depth indicating how many levels of descendents to return.

- `depth`: (optional) an integer greater than or equal to 0 that limits the depth of trees returned. Zero means return the group with no children.

For example:

```
GET /v1/group-children/00000000-0000-4000-8000-000000000000?depth=2
```

Response format

The response is a JSON array of group objects, using the following keys:

Key	Definition
<code>name</code>	The name of the node group (a string).
<code>id</code>	The node group's ID, which is a string containing a type-4 (random) UUID.
<code>description</code>	An optional key containing an arbitrary string describing the node group.
<code>environment</code>	The name of the node group's environment (a string), which indirectly defines what classes are available for the node group to set, and is the environment that nodes classified into this node group run under.
<code>environment_trumps</code>	This is a boolean that changes the behavior of classifications that this node group is involved in. The default behavior is to return a classification-conflict error if the node groups that a node is classified into do not all have the same environment. If this flag is set, then this node group's environment overrides the environments of the other node groups (provided that none of them also have this flag set), with no attempt made to validate that the other node groups' classes and class parameters exist in this node group's environment.
<code>parent</code>	The ID of the node group's parent (a string). A node group is not permitted to be its own parent, unless it is the All Nodes group (which is the root of the hierarchy). Note that the root group always has the lowest-possible random UUID, 00000000-0000-4000-8000-000000000000.
<code>rule</code>	A boolean condition on node properties. When a node's properties satisfy this condition, it's classified into the node group. See Rule condition grammar for more information on how this condition must be structured.

Key	Definition
<code>classes</code>	An object that defines both the classes consumed by nodes in this node group and any non-default values for their parameters. The keys of the object are the class names, and the values are objects describing the parameters. The parameter objects' keys are parameter names, and the values are what the node group sets for that parameter (always a string).
<code>deleted</code>	An object similar the <code>classes</code> object that shows which classes and class parameters set by the node group have since been deleted from Puppet. If none of the node group's classes or parameters have been deleted, this key is not present, so checking the presence of this key is an easy way to check whether the node group has references that need updating. The keys of this object are class names, and the values are further objects. These secondary objects always contain the <code>puppetlabs.classifier/deleted</code> key, whose value is a Boolean indicating whether the entire class has been deleted from Puppet. The other keys of these objects are parameter names, and the other values are objects that always contain two keys: <code>puppetlabs.classifier/deleted</code> , mapping to a boolean indicating whether the specific class parameter has been deleted from Puppet; and <code>value</code> , mapping to the string value set by the node group for this parameter (the value is duplicated for convenience, as it also appears in the <code>classes</code> object).
<code>variables</code>	An object that defines the values of any top-level variables set by the node group. The object is a mapping between variable names and their values (which can be any JSON value).
<code>children</code>	A JSON array of the group's immediate children. Children of children are included to the optionally-specified depth.
<code>immediate_child_count</code>	The number of immediate children of the group. Child count reflects the number of children that exist in the classifier, not the number that are returned in the request, which can vary based on permissions and query parameters.

The following is an example response from a query of the root node group with two children, each with three children. The user has permission to view only `child-1` and `grandchild-5`, which limits the response.

```
[
  {
    "name": "child-1",
    "id": "652227cd-af24-4fd8-96d4-b9b55ca28efb",
    "parent": "000000000-0000-4000-8000-000000000000",
    "environment_trumps": false,
    "rule": ["and", ["=", ["fact", "foo"], "bar"], ["not", ["<",
["fact", "uptime_days"], "31"]]]],
    "variables": {},
    "environment": "test",
```

```

    "classes": {},
    "children": [
      {
        "name": "grandchild-1",
        "id": "a3d976ad-51d3-4a29-af57-09990f3a2481",
        "parent": "652227cd-af24-4fd8-96d4-b9b55ca28efb",
        "environment_trumps": false,
        "rule": ["and", ["=", ["fact", "foo"], "bar"], ["or", ["~",
"name", "db"], ["<", ["fact", "processorcount"], "9"], ["=", ["fact",
"operatingsystem"], "Ubuntu"]]]],
        "variables": {},
        "environment": "test",
        "classes": {},
        "children": [],
        "immediate_child_count": 0
      },
      {
        "name": "grandchild-2",
        "id": "71905c11-5295-41cf-a143-31b278cfc859",
        "parent": "652227cd-af24-4fd8-96d4-b9b55ca28efb",
        "environment_trumps": false,
        "rule": ["and", ["=", ["fact", "foo"], "bar"], ["not", ["~",
["fact", "kernel"], "SunOS"]]]],
        "variables": {},
        "environment": "test",
        "classes": {},
        "children": [],
        "immediate_child_count": 0
      }
    ],
    "immediate_child_count": 2
  },
  {
    "name": "grandchild-5",
    "id": "0bb94f26-2955-4adc-8460-f5ce244d5118",
    "parent": "0960f75e-cdd0-4966-96f6-5e60948a7217",
    "environment_trumps": false,
    "rule": ["and", ["=", ["fact", "foo"], "bar"], ["and", ["<",
["fact", "processorcount"], "16"], [">=", ["fact", "kernelmajversion"],
"2"]]]],
    "variables": {},
    "environment": "test",
    "classes": {},
    "children": [],
    "immediate_child_count": 0
  }
]

```

Permissions

The response returned varies based on your permissions.

Permissions	Response
View the specified group only	An array containing the group and its descendents, ending at the optional depth
View descendents of the specified group, but not the group itself	An array starting at the roots of every tree you have permission to view and ending at the optional depth
View neither the specified group nor its descendents	An empty array

Error responses

Responses and keys returned for group requests depend on the type of error.

malformed-uuid

If the requested ID is not a valid UUID, the service returns a 400: Bad Request response using the following keys:

Key	Definition
kind	"malformed-uuid"
details	The malformed UUID as received by the server.

malformed-number or illegal-count

If the value of the `depth` parameter is not an integer, or is a negative integer, the service returns a 400: Bad Request response using one of the following keys:

Key	Definition
kind	"malformed-number" or "illegal-count"

Rules endpoint

Use the rules endpoint to translate a group's rule condition into PuppetDB query syntax.

POST /v1/rules/translate

Translate a group's rule condition into PuppetDB query syntax.

Request format

The request's body contains a rule condition as it would appear in the `rule` field of a group object.

The endpoint supports an optional query parameter `format`, which defaults to `nodes`. If specified as `format=inventory`, it allows you to get the classifier rules in a compatible [dot notation](#) format instead of the standard [PuppetDB AST](#).

Response format

The response is a PuppetDB query string that can be used with PuppetDB nodes endpoint in order to see which nodes would satisfy the rule condition (that is, which nodes would be classified into a group with that rule).

Error responses

Rules that use structured or trusted facts cannot be converted into PuppetDB queries, because PuppetDB does not yet support structured or trusted facts. If the rule cannot be translated into a PuppetDB query, the server returns a 422 Unprocessable Entity response containing the usual JSON error object. The error object has a `kind` of "untranslatable-rule", a `msg` that describes why the rule cannot be translated, and contains the received rule in `details`.

If the request does not contain a valid rule, the server returns a 400 Bad Request response with the usual JSON error object. If the rule was not valid JSON, the error's `kind` is "malformed-request", the `msg` states that the request's body could not be parsed as JSON, and the `details` contain the request's body as received by the server.

If the rule does not conform to the rule grammar, the `kind` key is "schema-violation", and the `details` key is an object with `submitted`, `schema`, and `error` keys which respectively describe the submitted object, the schema that object is expected to conform to, and how the submitted object failed to conform to the schema.

Related information

[GET /v1/groups](#) on page 361

Use the `/v1/groups` endpoint to retrieve a list of all node groups in the node classifier.

Import hierarchy endpoint

Use the import hierarchy endpoint to delete all existing node groups from the node classifier service and replace them with the node groups in the body of the submitted request.

POST /v1/import-hierarchy

Delete *all* existing node groups from the node classifier service and replace them with the node groups in the body of the submitted request.

The request's body must contain an array of node groups that form a valid and complete node group hierarchy. Valid means that the hierarchy does not contain any cycles, and complete means that every node group in the hierarchy is reachable from the root.

Request format

The request body must be a JSON array of node group objects as described in the `groups` endpoint documentation. All fields of the node group objects must be defined; no default values are supplied by the service.

Note that the output of the group collection endpoint, `/v1/groups`, is valid input for this endpoint.

Response format

If the submitted node groups form a complete and valid hierarchy, and the replacement operation is successful, a 204 No Content response with an empty body is returned.

Error responses

If any of the node groups in the array are malformed, a 400 Bad Request response is returned. The response contains the usual JSON error payload. The `kind` key is "schema-violation"; the `msg` key contains a short description of the problems with the malformed node groups; and the `details` key contains an object with three keys:

- `submitted`: an array of only the malformed node groups found in the submitted request.
- `error`: an array of structured descriptions of how the node group at the corresponding index in the submitted array failed to meet the schema.
- `schema`: the structured schema for node group objects.

If the hierarchy formed by the node groups contains a cycle, then a 422 Unprocessable Entity response is returned. The response contains the usual JSON error payload, where the `kind` key is "inheritance-cycle", the `msg` key contains the names of the node groups in the cycle, and the `details` key contains an array of the complete node group objects in the cycle.

If the hierarchy formed by the node groups contains node groups that are unreachable from the root, then a 422 Unprocessable Entity response is returned. The response contains the usual JSON error payload, where the `kind` key is "unreachable-groups", the `msg` lists the names of the unreachable node groups, and the `details` key contains an array of the unreachable node group objects.

Related information

[Groups endpoint](#) on page 361

The `groups` endpoint is used to create, read, update, and delete groups.

[Node classifier errors](#) on page 406

Familiarize yourself with error responses to make working the node classifier service API easier.

Last class update endpoint

Use the last class update endpoint to retrieve the time that classes were last updated from the Puppet master.

GET /v1/last-class-update

Use the `/v1/last-class-update` endpoint to retrieve the time that classes were last updated from the Puppet master.

Response

The response is always an object with one field, `last_update`. If there has been an update, the value of `last_update` field is the time of the last update in ISO8601 format. If the node classifier has never updated from Puppet, the field is null.

Update classes endpoint

Use update classes endpoint to trigger the node classifier to update class and environment definitions from the Puppet master.

POST /v1/update-classes

Use the `/v1/update-classes` endpoint to trigger the node classifier to update class and environment definitions from the Puppet master. The classifier service uses this endpoint when you refresh classes in the console.

Note: If you don't use Code Manager *and* you changed the default value of the `environment-class-cache-enabled` server setting, you must [manually delete the environment cache](#) before using this endpoint.

Query parameters

The request accepts the following optional parameter:

Parameter	Value
<code>environment</code>	If provided, fetches classes for only the specified environment.

For example:

```
curl -X POST https://localhost:4433/classifier-api/v1/update-classes?
environment=production
--cert <PATH TO CERT>
--key <PATH TO KEY>
--cacert <PATH TO PUPPET CA CERT>
```

Response

For a successful update, the service returns a 201 response with an empty body.

Error responses

If the Puppet master returns an unexpected status to the node classifier, the service returns a 500: `Server Error` response with the following keys:

Key	Definition
<code>kind</code>	"unexpected-response"
<code>msg</code>	Describes the error

Key	Definition
details	A JSON object, which has <code>url</code> , <code>status</code> , <code>headers</code> , and <code>body</code> keys describing the response the classifier received from the Puppet master

Related information

[Enable or disable cached data when updating classes](#) on page 192

The optional `environment-class-cache-enabled` setting specifies whether cached data is used when updating classes in the console. When `true`, Puppet Server refreshes classes using file sync, improving performance.

Validation endpoints

Use validation endpoints to validate groups in the node classifier.

POST /v1/validate/group

Use the `/v1/validate/group` endpoint to validate groups in the node classifier.

Request format

The request contains a group object. The request uses the following keys:

Key	Definition
name	The name of the node group (required).
environment	The name of the node group's environment. This key is optional; if it's not provided, the default environment (<code>production</code>) is used.
environment_trumps	Whether this node group's environment overrides those of other node groups at classification-time. This key is optional; if it's not provided, the default value of <code>false</code> is used.
description	A string describing the node group. This key is optional; if it's not provided, the node group has no description and this key doesn't appear in responses.
parent	The ID of the node group's parent (required).
rule	The condition that must be satisfied for a node to be classified into this node group. The structure of this condition is described in the "Rule Condition Grammar" section above.
variables	An object that defines the names and values of any top-level variables set by the node group. The keys of the object are the variable names, and the corresponding value is that variable's value, which can be any sort of JSON value. The <code>variables</code> key is optional, and if a node group does not define any top-level variables then it can be omitted.

Key	Definition
<code>classes</code>	An object that defines the classes to be used by nodes in the node group. The <code>classes</code> key is required, and at minimum is an empty object (<code>{}</code>). The <code>classes</code> key also contains the parameters for each class. Some classes have required parameters. This is a two-level object; that is, the keys of the object are class names (strings), and each key's value is another object that defines class parameter values. This innermost object maps between class parameter names and their values. The keys are the parameter names (strings), and each value is the parameter's value, which can be any kind of JSON value. The <code>classes</code> key is not optional; if it is missing, the service returns a 400: Bad Request response.

Response format

If the group is valid, the service returns a 200 OK response with the validated group as the body.

If a validation error is encountered, the service returns one of the following 400-level error responses.

Responses and keys returned for create group requests depend on the type of error.

`schema-violation`

If any of the required keys are missing or the values of any of the defined keys do not match the required type, the service returns a 400: Bad Request response using the following keys:

Key	Definition
<code>kind</code>	"schema-violation"
<code>details</code>	<p>An object that contains three keys:</p> <ul style="list-style-type: none"> <code>code>submitted</code>: Describes the submitted object.</code> <code>schema</code>: Describes the schema that object is expected to conform to. <code>error</code>: Describes how the submitted object failed to conform to the schema.

`malformed-request`

If the request's body could not be parsed as JSON, the service returns a 400: Bad Request response using the following keys:

Key	Definition
<code>kind</code>	"malformed-request"
<code>details</code>	<p>An object that contains two keys:</p> <ul style="list-style-type: none"> <code>body</code>: Holds the request body that was received. <code>error</code>: Describes how the submitted object failed to conform to the schema.

`uniqueness-violation`

If your attempt to create the node group violates uniqueness constraints (such as the constraint that each node group name must be unique within its environment), the service returns a 422: `Unprocessable Entity` response using the following keys:

Key	Definition
kind	"uniqueness-violation"
msg	Describes which fields of the node group caused the constraint to be violated, along with their values.
details	<p>An object that contains two keys:</p> <ul style="list-style-type: none"> • <code>conflict</code>: An object whose keys are the fields of the node group that violated the constraint and whose values are the corresponding field values. • <code>constraintName</code>: The name of the database constraint that was violated.

missing-referents

If classes or class parameters defined by the node group, or inherited by the node group from its parent, do not exist in the submitted node group's environment, the service returns a 422: `Unprocessable Entity` response. In both cases the response object uses the following keys:

Key	Definition
kind	"missing-referents"
msg	Describes the error and lists the missing classes or parameters.
details	<p>An array of objects, where each object describes a single missing referent, and has the following keys:</p> <ul style="list-style-type: none"> • <code>kind</code>: "missing-class" or "missing-parameter", depending on whether the entire class doesn't exist, or the class just doesn't have the parameter. • <code>missing</code>: The name of the missing class or class parameter. • <code>environment</code>: The environment that the class or parameter is missing from; that is, the environment of the node group where the error was encountered. • <code>group</code>: The name of the node group where the error was encountered. Due to inheritance, this might not be the group where the parameter was defined. • <code>defined_by</code>: The name of the node group that defines the class or parameter.

missing-parent

If the parent of the node group does not exist, the service returns a 422: `Unprocessable Entity` response. The response object uses the following keys:

Key	Definition
kind	"missing-parent"
msg	Shows the parent UUID that did not exist.
details	The full submitted node group.

`inheritance-cycle`

If the request causes an inheritance cycle, the service returns a `422: Unprocessable Entity` response. The response object uses the following keys:

Key	Definition
<code>kind</code>	"inheritance-cycle"
<code>details</code>	An array of node group objects that includes each node group involved in the cycle
<code>msg</code>	A shortened description of the cycle, including a list of the node group names with each followed by its parent until the first node group is repeated.

Node classifier errors

Familiarize yourself with error responses to make working the node classifier service API easier.

Error response description

Errors from the node classifier service are JSON responses.

Error responses contain these keys:

Key	Definition
<code>kind</code>	A string classifying the error. It is the same for all errors that have the same kind of thing in their <code>details</code> key.
<code>msg</code>	A human-readable message describing the error, suitable for presentation to the user.
<code>details</code>	Additional machine-readable information about the error condition. The format of this key's value varies between kinds of errors but is the same for any given error kind.

Internal server errors

Any endpoint might return a `500: Internal Server Error` response in addition to its usual responses. There are two kinds of internal server error responses: `application-error` and `database-corruption`.

An `application-error` response is a catchall for unexpected errors. The `msg` of an `application-error` 500 contains the underlying error's message first, followed by a description of other information that can be found in `details`. The `details` contain the error's stack trace as an array of strings under the `trace` key, and might also contain `schema`, `value`, and `error` keys if the error was caused by a schema validation failure.

A `database-corruption` 500 response occurs when a resource that is retrieved from the database fails to conform to the schema expected of it by the application. This is probably just a bug in the software, but it could potentially indicate either genuine corruption in the database or that a third party has changed values directly in the database. The `msg` section contains a description of how the database corruption could have occurred. The `details` section contains `retrieved`, `schema`, and `error` keys, which have the resource as retrieved, the schema it is expected to conform to, and a description of how it fails to conform to that schema as the respective values.

Not found errors

Any endpoint where a resource identifier is supplied can produce a `404 Not Found Error` response if a resource with that identifier could not be found.

All not found error responses have the same form. The `kind` is "not-found", the `msg` is "The resource could not be found.", and the `details` key contains the URI of the request that resulted in this response.

Node classifier API v2

These are the endpoints for the node classifier v2 API.

Related information

[API index](#) on page 33

APIs allow you to interact with Puppet and Puppet Enterprise (PE) applications from your own code or application integration hooks.

Classification endpoint

The `classification` endpoint takes a node name and a set of facts, and returns information about how that node is classified. The output can help you test your classification rules.

POST /v2/classified/nodes/<name>

Use the `/v2/classified/nodes/\<name\>` endpoint to retrieve the classification information for the node with the given name and facts as supplied in the body of the request.

Request format

The request body can contain a JSON object describing the facts and trusted facts of the node to be classified. The object can have these keys:

Key	Definition
<code>fact</code>	The regular, non-trusted facts of the node. The value of this key is a further object, whose keys are fact names, and whose values are the fact values. Fact values can be a string, number, boolean, array, or object.
<code>trusted</code>	The trusted facts of the node. The values of this key are subject to the same restrictions as those on the value of the <code>fact</code> key.

Response format

The response is a JSON object describing the node post-classification, using these keys:

Key	Definition
<code>name</code>	The name of the node (a string).
<code>groups</code>	An array of the groups that this node was classified into. The value of this key is an array of hashes containing the <code>id</code> and the <code>name</code> , sorted by <code>name</code> .
<code>environment</code>	The name of the environment that this node uses, which is taken from the node groups the node was classified into.
<code>classes</code>	An array of the classes (strings) that this node received from the groups it was classified into.

Key	Definition
parameters	An object describing class parameter values for the above classes wherever they differ from the default. The keys of this object are class names, and the values are further objects describing the parameters for just the associated class. The keys of that innermost object are parameter names, and the values are the parameter values, which can be any sort of JSON value.

This is an example of a response from this endpoint:

```
{
  "name": "foo.example.com",
  "groups": [{ "id": "9c0c7d07-a199-48b7-9999-3cdf7654e0bf",
    "name": "a group" },
    { "id": "96d1a058-225d-48e2-a1a8-80819d31751d",
    "name": "b group" } ],
  "environment": "staging",
  "classes": ["apache"],
  "parameters": {
    "apache": {
      "keepalive_timeout": 30,
      "log_level": "notice"
    }
  }
}
```

Error responses

If the node is classified into multiple node groups that define conflicting classifications for the node, the service returns a 500 Server Error response.

The body of this response contains the usual JSON error object described in the errors documentation.

The `kind` key of the error is "classification-conflict", the `msg` describes generally why this happens, and the `details` key contains an object that describes the specific conflicts encountered.

The keys of these objects are the names of parameters that had conflicting values defined, and the values are arrays of value detail objects. The details object may have between one and all of the following three keys.

Key	Definition
environment	Maps directly to an array of value detail objects (described below).
variables	Contains an object with a key for each conflicting variable, whose values are an array of value detail objects.
classes	Contains an object with a key for each class that had conflicting parameter definitions, whose values are further objects that describe the conflicts for that class's parameters.

A value details object describes one of the conflicting values defined for the environment, a variable, or a class parameter. Each object contains these keys:

Key	Definition
value	The defined value, which is a string for environment and class parameters, but for a variable can be any JSON value.
from	The node group that the node was classified into that caused this value to be added to the node's classification. This group cannot define the value, because it can be inherited from an ancestor of this group.
defined_by	The node group that actually defined this value. This is often the from group, but could instead be an ancestor of that group.

This example shows a classification conflict error object with node groups truncated for clarity:

```
{
  "kind": "classification-conflict",
  "msg": "The node was classified into multiple unrelated groups that
defined conflicting class parameters or top-level variables. See `details`
for a list of the specific conflicts.",
  "details": {
    "classes": {
      "songColors": {
        "blue": [
          {
            "value": "Blue Suede Shoes",
            "from": {
              "name": "Elvis Presley",
              "classes": {},
              "rule": ["=", "nodename", "the-node"],
              ...
            },
            "defined_by": {
              "name": "Carl Perkins",
              "classes": {"songColors": {"blue": "Blue Suede Shoes"}},
              "rule": ["not", ["=", "nodename", "the-node"]],
              ...
            }
          },
          {
            "value": "Since You've Been Gone",
            "from": {
              "name": "Aretha Franklin",
              "classes": {"songColors": {"blue": "Since You've Been Gone"}},
              ...
            },
            "defined_by": {
              "name": "Aretha Franklin",
              "classes": {"songColors": {"blue": "Since You've Been Gone"}},
              ...
            }
          }
        ]
      }
    }
  }
}
```

In this example, the conflicting "Blue Suede Shoes" value was included in the classification because the node matched the "Elvis Presley" group (since that is the value of the "from" key), but that group doesn't define the "Blue

Suede Shoes" value. That value is defined by the "Carl Perkins" group, which is an ancestor of the "Elvis Presley" group, causing the latter to inherit the value from the former. The other conflicting value, "Since You've Been Gone", is defined by the same group that the node matched.

Node inventory API

These are the endpoints for the node inventory v1 API.

Related information

[API index](#) on page 33

APIs allow you to interact with Puppet and Puppet Enterprise (PE) applications from your own code or application integration hooks.

Node inventory API: forming requests

Make well-formed HTTP(S) requests to the Puppet inventory service API.

By default, the node inventory service listens on port 8143, and all endpoints are relative to the `/inventory/v1` path. For example, the full URL for the `/command/create-connection` endpoint on localhost is `https://localhost:8143/inventory/v1/command/create-connection`.

Token authentication

All requests made to the inventory service API require authentication. You do this for each endpoint in the service using user authentication tokens contained within the X-Authentication request header.

Example token usage: create a connection entry

To post a new connection entry to the inventory service when running on localhost, first generate a token with the puppet-access tool. Then copy that token and replace `<TOKEN>` in the following curl command.

```
curl -k -H 'X-Authentication:<TOKEN>' \
  -H "Content-Type: application/json" \
  https://localhost:8143/inventory/v1/command/create-connection -X POST \
  -d '{"certnames": ["new.node"], "type": "ssh", "parameters": {"tmpdir":
"/tmp", "port": 1234}, "sensitive_parameters": {"username": "root",
"password": "password"}, "duplicates": "replace"}'
```

Example token usage: query connections for a certname

To query the `/query/connections` endpoint, use the same token and header pattern.

```
curl -k -X GET https://localhost:8143/inventory/v1/query/connections?
certname="new.node" \
  -H 'X-Authentication:<TOKEN>'
```

Related information

[Token-based authentication](#) on page 242

Puppet Enterprise users generate tokens to authenticate their access to certain PE command-line tools and API endpoints. Authentication tokens are tied to the permissions granted to the user through RBAC, and provide the user with the appropriate access to HTTP requests.

POST /command/create-connection

Create a new connection entry in the inventory service database.

Connection entries contain connection options, such as credentials, which are used to connect to the certnames provided in the payload via the provided connection type.

This endpoint also inserts each of the provided certnames into PuppetDB with an empty fact set, if they are not already present. After certnames have been added to PuppetDB, you can view them from the **Nodes** page in the Puppet Enterprise console. You can also add them to an inventory node list when you set up a job to run tasks.

Important: When the Puppet orchestrator targets a certname to run a task, it first considers the value of the `hostname` key present in the `parameters`, if available. Otherwise, it uses the value of the `certnames` key as the hostname. A good practice is to include a `hostname` key only when the hostname differs from the `certname`. Do not include a `hostname` key for multiple `certname` connection entries.

Request format

The request body must be a JSON object.

certnames

required Array containing the list of certnames to associate with connection information.

type

required String containing either `ssh` or `winrm`. Instructs bolt-server which connection type to use to access the node when running a task.

parameters

required Object containing arbitrary key/value pairs. The necessary parameters for connecting to the provided certnames.



CAUTION: A `hostname` key entered here takes precedence over the values in `certnames` key.

sensitive_parameters

required Object containing arbitrary key/value pairs. The necessary sensitive data for connecting to the provided certnames, stored in an encrypted format.

duplicates

required String containing either `error` or `replace`. Instructs how to handle cases where one or more provided certnames conflict with existing certnames stored in the inventory connections database. `error` results in a 409 response if any certnames are duplicates. `replace` overwrites the existing certnames if there are conflicts.

Request examples

```
{
  "certnames": ["avery.gooddevice", "amediocre.device"],
  "type": "ssh",
  "parameters": {
    "tmpdir": "/tmp",
    "port": 1234
  },
  "sensitive_parameters": {
    "username": "root",
    "password": "password"
  },
  "duplicates": "replace"
}
```

```
curl -k -H 'X-Authentication:<AUTHENTICATION TOKEN>' \
  -H "Content-Type: application/json" \
  https://<MASTER-HOST>:8143/inventory/v1/command/create-connection -X
POST \
  -d '{"certnames": ["avery.gooddevice", "amediocre.device"],
    "type": "ssh", "parameters": {"tmpdir": "/tmp", "port": 1234},
```

```
"sensitive_parameters": {"username": "root", "password": "password"},
"duplicates": "replace"}
```

Response format

If the request is valid according to the schema and the entry is successfully recorded in the database, the server returns a 201 response. The response has the same format for single and multiple certname entries.

The response is a JSON object containing the `connection_id`.

connection_id

A unique identifier that can be used to reference the record.

Response example

```
{
  "connection_id": "3c4df64f-7609-4d31-9c2d-acfa52ed66ec"
}
```

POST /command/delete-connection

Delete certnames from all associated connection entries in the inventory service database and update those certname entries in the PuppetDB with the flag `preserve: false`.

Request format

The request body must be a JSON object containing a `certnames` key.

certnames

required Array containing the list of certnames to be removed.

Request examples

```
{
  "certnames": ["avery.gooddevice", "amediocre.device"]
}
```

```
curl -k -H 'X-Authentication:<AUTHENTICATION TOKEN>' \
  -H 'Content-Type: application/json' \
  https://<MASTER-HOST>:8143/inventory/v1/command/delete-connection -X
POST \
  -d '{"certnames": ["avery.gooddevice", "amediocre.device"]}'
```

Response format

If the request matches the schema and is processed successfully, the service responds with a 204 status code and no body payload. If the user is unauthorized, the service responds with a 403 error code.

Command endpoint error responses

General format of error responses.

Every error response from the inventory service is a JSON response. Each response is an object that contains the following keys:

kind

The kind of error encountered.

msg

The message associated with the error.

details

A hash with more information about the error.

For example, if the request is missing the required content type headers, the following error occurs:

```
{
  "kind" : "puppetlabs.inventory/not-acceptable",
  "msg" : "accept must include content-type json",
  "details" : ""
}
```

kind error responses

For this endpoint, the `kind` key of the error displays the conflict.

puppetlabs.inventory/unknown-error

If an unknown error occurs during the request, the server returns a 500 response.

puppetlabs.inventory/not-acceptable

If the content provided to the API contains an "accepts" header which does not allow for JSON, the server returns a 406 response.

puppetlabs.inventory/unsupported-type

If the content provided to the API contains a "content-type" header other than JSON, the server returns a 416 response.

puppetlabs.inventory/json-parse-error

If there is an error while processing the request body, the server returns a 400 response.

puppetlabs.inventory/schema-validation-error

If there is a violation of the required format for the request body, the server returns a 400 response.

puppetlabs.inventory/not-permitted

If the user requesting an action does not have the necessary permissions to do so, the server returns a 403 response.

puppetlabs.inventory/duplicate-certnames

If the `duplicates` parameter is not set, or is set as `error`, and one or more of the certnames in the request body already exist within the inventory, the server returns a 409 response.

GET /query/connections

List all the connections entries in the inventory database.

Request format

The request body must be a JSON object.

certname

optional String that represents the single certname to retrieve.

sensitive

optional String or boolean that instructs whether to return sensitive parameters for each connection in the response. This parameter is gated by permission validation.

extract

optional Array of keys to return for each certname. `connection_id` is always returned, regardless of whether it is included. When `extract` is not present in the body, all keys are returned.

Tip: In order to return sensitive parameters in the extract list, the `sensitive` query parameter must be present and resolve to true. Otherwise, they are excluded.

Response format

The response is a JSON object containing the known connections. The following keys are used:

items

Contains an array of all the known connections matching the specified (or not specified) filtering criteria. Each item under `items` is an object with the following keys:

connection_id

String that is the unique identifier for the connections entry.

certnames

Array of strings that contains the certnames of the matching connections entries.

type

String that describes the type of connection for the given information. For example, `ssh` or `winrm`.

parameters

Object containing arbitrary key/value pairs and describes connection options for the entry.

sensitive_parameters

when specified and permitted An object that contains arbitrary key/value pairs and describes the sensitive connection options for the entry.

Response examples

A response for a request to list all the connections entries in the inventory database: `GET /query/connections`

```
{
  "items": [
    {
      "connection_id": "3c4df64f-7609-4d31-9c2d-acfa52ed66ec",
      "certnames": ["devices.arecool", "dont.youthink"],
      "type": "ssh",
      "parameters": {
        "tmpdir": "/tmp",
        "port": 1234
      }
    },
    {
      "connection_id": "4932bfe7-69c4-412f-b15c-ac0a7c2883f1",
      "certnames": ["managing.devices", "is.evencooler"],
      "type": "winrm",
      "parameters": {
        "tmpdir": "/tmp",
        "port": 1234
      }
    }
  ]
}
```

A response for a request to list a specific certname: `GET /query/connections?certname="averygood.device"`

```
{
  "items": [
    {
      "connection_id": "3c4df64f-7609-4d31-9c2d-acfa52ed66ec",
      "certnames": ["averygood.device"],
    }
  ]
}
```

```

    "type": "ssh",
    "parameters": {
      "tmpdir": "/tmp",
      "port": 1234
    }
  ]
}

```

A response for request to list a specific certname and its sensitive parameters: GET /query/connections?certname="averygood.device"&sensitive=true example

```

{
  "items": [
    {
      "connection_id": "3c4df64f-7609-4d31-9c2d-acfa52ed66ec",
      "certnames": ["averygood.device"],
      "type": "ssh",
      "parameters": {
        "tmpdir": "/tmp",
        "port": 1234
      },
      "sensitive_parameters": {
        "username": "johnjohnjimmyjohn",
        "password": "C7b0$1s8lt0"
      }
    }
  ]
}

```

A response for request to list a specific certname and its value for the type key: GET /query/connections?certname="averygood.device"&extract=["type"]

```

{
  "items": [
    {
      "connection_id": "3c4df64f-7609-4d31-9c2d-acfa52ed66ec",
      "certnames": ["averygood.device"],
      "type": "ssh"
    }
  ]
}

```

POST /query/connections

Retrieve inventory connections in bulk via certname.

Request format

The request body must be a JSON object.

certnames

Array containing the list of certnames to retrieve from the inventory database. If this key is not included, all connections are returned.

extract

Array of keys to return for each certname. `connection_id` is always returned, regardless of whether it is included. When `extract` is not present in the body, all keys are returned.

Tip: In order to return sensitive parameters in the extract list, the `sensitive` query parameter must be present and resolve to true. Otherwise, they are excluded.

sensitive

optional String or boolean that instructs whether to return sensitive parameters for each connection in the response. This parameter is gated by permission validation.

Response format

The response is a JSON object containing the known connections. The following keys are used:

items

Contains an array of all the known connections matching the specified (or not specified) filtering criteria. Each item under `items` is an object with the following keys:

connection_id

String that is the unique identifier for the connections entry.

certnames

Array of strings that contains the certnames of the matching connections entries.

type

String that describes the type of connection for the given information. For example, `ssh` or `winrm`.

parameters

Object containing arbitrary key/value pairs and describes connection options for the entry.

sensitive_parameters

when specified and permitted An object that contains arbitrary key/value pairs and describes the sensitive connection options for the entry.

Request and response examples

An empty request body.

```
{}
```

A response for an empty request.

```
{
  "items": [
    {
      "connection_id": "3c4df64f-7609-4d31-9c2d-acfa52ed66ec",
      "certnames": ["devices.arecool", "dont.youthink"],
      "type": "winrm",
      "parameters": {
        "tmpdir": "/tmp",
        "port": 1234
      }
    },
    {
      "connection_id": "4932bfe7-69c4-412f-b15c-ac0a7c2883f1",
      "certnames": ["managing.devices", "is.evencooler"],
      "type": "ssh",
      "parameters": {
        "tmpdir": "/tmp",
        "port": 1234
      }
    }
  ]
}
```


A certnames request body.

```
{
  "certnames": ["averygood.device"]
}
```

A response for a certnames request.

```
{
  "items": [
    {
      "connection_id": "3c4df64f-7609-4d31-9c2d-acfa52ed66ec",
      "certnames": ["averygood.device"],
      "type": "ssh",
      "parameters": {
        "tmpdir": "/tmp",
        "port": 1234
      }
    }
  ]
}
```

A request to list a specific certname and its sensitive parameters: /query/connections?sensitive=true

```
{
  "certnames": ["averygood.device"],
  "extract": ["certnames", "sensitive_parameters"]
}
```

A response for a request for a specific certname and its sensitive parameters.

```
{
  "items": [
    {
      "connection_id": "3c4df64f-7609-4d31-9c2d-acfa52ed66ec",
      "certnames": ["averygood.device"],
      "sensitive_parameters": {
        "username": "johnjohnjimmyjohn",
        "password": "C7b0$ls8ltO"
      }
    }
  ]
}
```

Query endpoint error responses

General format of error responses.

Every error response from the inventory service is a JSON response. Each response is an object that contains the following keys:

kind

The kind of error encountered.

msg

The message associated with the error.

details

A hash with more information about the error.

For example, if the request is missing the required content type headers, the following error occurs:

```
{
  "kind" : "puppetlabs.inventory/not-acceptable",
  "msg" : "accept must include content-type json",
  "details" : ""
}
```

kind error responses

For this endpoint, the `kind` key of the error displays the conflict.

puppetlabs.inventory/unknown-error

If an unknown error occurs during the request, the server returns a 500 response.

puppetlabs.inventory/not-acceptable

If the content provided to the API contains an "accepts" header which does not allow for JSON, the server returns a 406 response.

puppetlabs.inventory/unsupported-type

If the content provided to the API contains a "content-type" header other than JSON, the server returns a 416 response.

puppetlabs.inventory/json-parse-error

If there is an error while processing the request body, the server returns a 400 response.

puppetlabs.inventory/schema-validation-error

If there is a violation of the required format for the request body, the server returns a 400 response.

Orchestrating Puppet, tasks, and plans

Puppet orchestrator is an effective tool for making on-demand changes to your infrastructure.

With orchestrator you can:

- Automate tasks and plans tasks to eliminate manual work across your infrastructure and applications.
- Initiate Puppet runs whenever you need to update agents.
- Group the servers in your network according to immediate business needs. By leveraging orchestrator with [PuppetDB](#), which stores detailed information about your nodes, you can search and filter servers based on metadata. No static lists of hosts, no reliance on complicated host-naming conventions.
- Connect with thousands of hosts at the same time without slowing down your network. The Puppet Communications Protocol (PCP) and message broker efficiently mediate communications on your network even as your operational demands grow.
- Distribute the Puppet agent workload by adding masters that are dedicated to catalog compilation as you increase in scale. Compilers efficiently process requests and compile code for environments that have thousands of nodes. For more information, see [Installing compilers](#).
- Take advantage of the tasks installed with PE. Use the package task to inspect, install, upgrade, and manage packages or the service task to start, stop, restart, and check the status of services running on your systems. Additional tasks are available from the Puppet [Forge](#).
- Write your own tasks in any programming language that your target nodes can run, such as Bash, PowerShell, or Python.
- Write plans in Puppet or YAML or download them from the [Forge](#).
- Integrate server logging, auditing, and per node role-based access control (RBAC).
- Use plans to create customized permissions for tasks.

Running jobs with Puppet orchestrator

With the Puppet orchestrator, you can run two types of "jobs": on-demand Puppet runs or Puppet tasks.

When you run Puppet on-demand with the orchestrator, you control the rollout of configuration changes when and how you want them. You control when Puppet runs and where node catalogs are applied (from the environment level to an individual node). You no longer need to wait on arbitrary run times to update your nodes.

Puppet tasks allow you to execute actions on target machines. A "task" is a single action that you execute on the target via an executable file. For example, do you want to upgrade a package or restart a particular service? Set up a Puppet task job to enforce to make those changes at will.

Tasks are packaged and distributed as Puppet modules.

Puppet orchestrator technical overview

The orchestrator uses pe-orchestration-services, a JVM-based service in PE, to execute on-demand Puppet runs on agent nodes in your infrastructure. The orchestrator uses PXP agents to orchestrate changes across your infrastructure.

The orchestrator (as part of pe-orchestration-services) controls the functionality for the `puppet - job` and `puppet - app` commands, and also controls the functionality for jobs and single node runs in the PE console.

The orchestrator is comprised of several components, each with their own configuration and log locations.

Puppet orchestrator architecture

The functionality of the orchestrator is derived from the Puppet Execution Protocol (PXP) and the Puppet Communications Protocol (PCP).

- PXP: A message format used to request that a task be executed on a remote host and receive responses on the status of that task. This is used by the pe-orchestration services to run Puppet on agents.
- PXP agent: A system service in the agent package that runs PXP.
- PCP: The underlying communication protocol that describes how PXP messages get routed to an agent and back to the orchestrator.
- PCP broker: A JVM-based service that runs in pe-orchestration-services on the master and in the pe-puppetservice on compilers. PCP brokers route PCP messages, which declare the content of the message via message type, and identify the sender and intended recipient. PCP brokers on compilers connect to the orchestrator, and the orchestrator uses the brokers to direct messages to PXP agents connected to the compilers. When using compilers, PXP agents running on PE components (the Puppet master, PuppetDB, and the PE console) connect directly to the orchestrator, but all other PXP agents connect to compilers via load balancers.

What happens during an on-demand run from the orchestrator ?

Several PE services interact when you run Puppet on demand from the orchestrator.

1. You use the `puppet - job` command to create a job in orchestrator.
2. The orchestrator validates your token with the PE RBAC service.
3. The orchestrator requests environment classification from the node classifier for the nodes targeted in the job, and it queries PuppetDB for the nodes.
4. The orchestrator requests the environment graph from Puppet Server.
5. The orchestrator creates the job ID and starts polling nodes in the job to check their statuses.
6. The orchestrator queries PuppetDB for the agent version on the nodes targeted in the job.
7. The orchestrator tells the PCP broker to start runs on the nodes targeted in the job, and Puppet runs start on those agents.
8. The agent sends its run results to the PCP broker.
9. The orchestrator receives run results, and requests the node run reports from PuppetDB.

What happens during a task run from the orchestrator?

Several services interact for a task run as well. Because tasks are Puppet code, they must be deployed into an environment on the Puppet master. Puppet Server then exposes the task metadata to the orchestrator. When a task is run, the orchestrator sends the PXP agent a URL of where to fetch the task from the master and the checksum of the task file. The PXP agent downloads the task file from the URL and caches it for future use. The file is validated against the checksum before every execution. The following are the steps in this process.

1. The PE client sends a task command.
2. The orchestrator checks if a user is authorized.
3. The orchestrator fetches the node target from PuppetDB if the target is a query, and returns the nodes.
4. The orchestrator requests task data from Puppet Server.
5. Puppet Server returns task metadata, file URIs, and file SHAs.
6. The orchestrator validates the task command and then sends the job ID back to the client.
7. The orchestrator sends task parameters and file information to the PXP agent.
8. The PXP agent sends a provisional response to the orchestrator, checks the SHA against the local cache, and requests the task file from Puppet Server.
9. Puppet Server returns the task file to the PXP agent.
10. The task runs.
11. The PXP agent sends the result to the orchestrator.
12. The client requests events from the orchestrator.
13. The orchestrator returns the result to the client.

Configuration notes for the orchestrator and related components

Configuration and tuning for the components in the orchestrator happens in various files.

- **pe-orchestration-services:** The underlying service for the orchestrator. The main configuration file is `/etc/puppetlabs/orchestration-services/conf.d`.

Additional configuration for large infrastructures can include tuning the pe-orchestration-services JVM heap size, increasing the limit on open file descriptors for pe-orchestration-services, and tuning ARP tables.

- **PCP broker:** Part of the pe-puppetserver service. The main configuration file is `/etc/puppetlabs/puppetserver/conf.d`.

The PCP broker requires JVM memory and file descriptors, and these resources scale linearly with the number of active connections. Specifically, the PCP broker requires:

- Approximately 40 KB of memory (when restricted with the `-Xmx` JVM option)`
- One file descriptor per connection
- An approximate baseline of 60 MB of memory and 200 file descriptors

For a deployment of 100 agents, expect to configure the JVM with at least `-Xmx64m` and 300 file descriptors. Message handling requires minimal additional memory.

- **PXP agent:** Configuration is managed by the agent profile (`puppet_enterprise::profile::agent`).

The PXP agent is configured to use Puppet's SSL certificates and point to one PCP broker endpoint. If high availability (HA) is configured, the agent points to additional PCP broker endpoints in the case of failover.

Note: If you reuse an existing agent with a new orchestrator instance, you must delete the pxp-agent spool directory, located at `/opt/puppetlabs/pxp-agent/spool` (*nix) or `C:\ProgramData\PuppetLabs\pxp-agent\var\spool` (Windows)

Debugging the orchestrator and related components

If you need to debug the orchestrator or any of its related components, the following log locations might be helpful.

- **pe-orchestration-services:** The main log file is `/var/log/puppetlabs/orchestration-services/orchestration-services.log`.
- **PCP:** The main log file for PCP brokers on compilers is `/var/log/puppetlabs/puppetserver/pcp-broker.log`. You can configure logback through the Puppet server configuration.

The main log file for PCP brokers on the master is `/var/log/puppetlabs/orchestration-services/pcp-broker.log`.

You can also enable an access log for messages.

- **PXP agent:** The main log file is `/var/log/puppetlabs/pxp-agent/pxp-agent.log` (on *nix) or `C:\ProgramData\PuppetLabs/pxp-agent/var/log/pxp-agent.log` (on Windows). You can configure this location as necessary.

Additionally, metadata about Puppet runs triggered via the PXP agent are kept in the `spool-dir`, which defaults to `/opt/puppetlabs/pxp-agent/spool` (on *nix) and `C:\ProgramData\PuppetLabs/pxp-agent/var/spool` (on Windows). Results are kept for 14 days.

Configuring Puppet orchestrator

Once you've installed PE or the client tools package, there are a few tasks you need to do to prepare your PE infrastructure for orchestration services.

- Set PE RBAC permissions and token authentication for Puppet orchestrator
- Enable cached catalogs for use with the orchestrator (optional)
- Review the orchestrator configuration files and adjust them as needed

All of these instructions assume that PE client tools are installed.

Related information

[Installing PE client tools](#) on page 169

PE client tools are a set of command line tools that let you access Puppet Enterprise services from a workstation that might or might not be managed by Puppet.

Orchestration services settings

`global.conf`: Global logging and SSL settings

`/etc/puppetlabs/orchestration-services/conf.d/global.conf` contains settings shared across the Puppet Enterprise (PE) orchestration services.

The file `global.certs` typically requires no changes and contains the following settings:

Setting	Definition	Default
<code>ssl-cert</code>	Certificate file path for the orchestrator host.	<code>/etc/puppetlabs/orchestration-services/ssl/<orchestrator-host-fqdn>.cert.pem</code>
<code>ssl-key</code>	Private key path for the orchestrator host.	<code>/etc/puppetlabs/orchestration-services/ssl/<orchestrator-host-fqdn>.private_key.pem</code>
<code>ssl-ca-cert</code>	CA file path	<code>/etc/puppetlabs/puppet/ssl/ca.pem</code>

The file `global.logging-config` is a path to `logback.xml` file that configures logging for most of the orchestration services. See <http://logback.qos.ch/manual/configuration.html> for documentation on the structure of the `logback.xml` file. It configures the log location, rotation, and formatting for the following:

- `orchestration-services` (appender section F1)
- `orchestration-services status` (STATUS)
- `pcp-broker` (PCP)
- `pcp-broker access` (PCP_ACCESS)
- `aggregate-node-count` (AGG_NODE_COUNT)

bootstrap.cfg: Allow list of trapperkeeper services to start

`/etc/puppetlabs/orchestration-services/bootstrap.cfg` is the list of trapperkeeper services from the orchestrator and `pcp-broker` projects that are loaded when the `pe-orchestration-services` system service starts.

- To disable a service in this list, remove it or comment it with a `#` character and restart `pe-orchestration-services`
- To enable an NREPL service for debugging, add `puppetlabs.trapperkeeper.services.nrepl.nrepl-service/nrepl-service` to this list and restart `pe-orchestration-services`.

webserver.conf and web-routes.conf: The pcp-broker and orchestrator HTTP services

`/etc/puppetlabs/orchestration-services/conf.d/webserver.conf` describes how and where to run `pcp-broker` and orchestrator web services, which accept HTTP API requests from the rest of the PE installation and from external nodes and users.

The file `webserver.orchestrator` configures the orchestrator web service. Defaults are as follows:

Setting	Definition	Default
<code>access-log-config</code>	A logback XML file configuring logging for orchestrator access messages.	<code>/etc/puppetlabs/orchestration-services/request-logging.xml</code>
<code>client-auth</code>	Determines the mode that the server uses to validate the client's certificate for incoming SSL connections.	<code>want</code> or <code>need</code>
<code>default-server</code>	Allows multi-server configurations to run operations without specifying a <code>server-id</code> . Without a <code>server-id</code> , operations will run on the selected default. Optional.	<code>true</code>
<code>ssl-ca-cert</code>	Sets the path to the CA certificate PEM file used for client authentication.	<code>/etc/puppetlabs/puppet/ssl/certs/ca.pem</code>
<code>ssl-cert</code>	Sets the path to the server certificate PEM file used by the web service for HTTPS.	<code>/etc/puppetlabs/orchestration-services/ssl/<orchestrator-host-fqdn>.cert.pem</code>
<code>ssl-crl-path</code>	Describes a path to a Certificate Revocation List file. Optional.	<code>/etc/puppetlabs/puppet/ssl/crl.pem</code>
<code>ssl-host</code>	Sets the host name to listen on for encrypted HTTPS traffic.	<code>0.0.0.0</code>

Setting	Definition	Default
<code>ssl-key</code>	Sets the path to the private key PEM file that corresponds with the <code>ssl-cert</code>	<code>/etc/puppetlabs/orchestration-services/ssl/<orchestrator-host-fqdn>.private_key.pem</code>
<code>ssl-port</code>	Sets the port to use for encrypted HTTPS traffic.	8143

The file `webserver.pcp-broker` configures the `pcp-broker` web service. Defaults are as follows:

Setting	Definition	Default
<code>client-auth</code>	Determines the mode that the server uses to validate the client's certificate for incoming SSL connections.	want or need
<code>ssl-ca-cert</code>	Sets the path to the CA certificate PEM file used for client authentication.	<code>/etc/puppetlabs/puppet/ssl/certs/ca.pem</code>
<code>ssl-cert</code>	Sets the path to the server certificate PEM file used by the web service for HTTPS.	<code>/etc/puppetlabs/orchestration-services/ssl/<orchestrator-host-fqdn>.cert.pem</code>
<code>ssl-crl-path</code>	Describes a path to a Certificate Revocation List file. Optional.	<code>/etc/puppetlabs/puppet/ssl/crl.pem</code>
<code>ssl-host</code>	Sets the host name to listen on for encrypted HTTPS traffic.	0.0.0.0.
<code>ssl-key</code>	Sets the path to the private key PEM file that corresponds with the <code>ssl-cert</code> .	<code>/etc/puppetlabs/orchestration-services/ssl/<orchestrator-host-fqdn>.private_key.pem</code>
<code>ssl-port</code>	Sets the port to use for encrypted HTTPS traffic.	8142

`/etc/puppetlabs/orchestration-services/conf.d/web-routes.conf` describes how to route HTTP requests made to the API web servers, designating routes for interactions with other services. These should not be modified. See the configuration options at the [trapperkeeper-webserver-jetty project's docs](#)

analytics.conf: Analytics trapperkeeper service configuration

`/etc/puppetlabs/orchestration-services/conf.d/analytics.conf` contains the internal setting for the [analytics](#) trapperkeeper service.

Setting	Definition	Default
<code>analytics.url</code>	Specifies the API root.	<code><puppetserver-host-url>:8140/analytics/v1</code>

auth.conf: Authorization trapperkeeper service configuration

`/etc/puppetlabs/orchestration-services/conf.d/auth.conf` contains internal settings for the authorization trapperkeeper service. See configuration options in the [trapperkeeper-authorization project's docs](#).

metrics.conf: JXM metrics trapperkeeper service configuration

/etc/puppetlabs/orchestration-services/conf.d/metrics.conf contains internal settings for the JXM metrics service built into orchestration-services. See the service configuration options in the [trapperkeeper-metrics project's docs](#).

orchestrator.conf: Orchestrator trapperkeeper service configuration

/etc/puppetlabs/orchestration-services/conf.d/orchestrator.conf contains internal settings for the orchestrator project's trapperkeeper service.

pcp-broker.conf: PCP broker trapperkeeper service configuration

/etc/puppetlabs/orchestration-services/conf.d/pcp-broker.conf contains internal settings for the pcp-broker project's trapperkeeper service. See the service configuration options in the [pcp-broker project's docs](#).

Setting PE RBAC permissions and token authentication for orchestrator

Before you run any orchestrator jobs, you need to set the appropriate permissions in PE role-based access control (RBAC) and establish token-based authentication.

Most orchestrator users require the following permissions to run orchestrator jobs or tasks:

Type	Permission	Definition
agent	Run on agent nodes.	The ability to run on nodes using the console or orchestrator. Instance must always be "*" .
Job orchestrator	Start, stop and view jobs	The ability to start and stop jobs and tasks, view jobs and job progress, and view an inventory of nodes that are connected to the PCP broker.
Tasks	Run tasks	The ability to run specific tasks on all nodes, a selected node group, or nodes that match a PQL query.
Nodes	View node data from .	The ability to view node data imported from . Object must always be "*" .

Note: If you do not have permissions to view a node group, or the node group doesn't have any matching nodes, that node group won't be listed as an option for viewing. In addition, a node group does not appear if no rules have been specified for it.

Assign task permissions to a user role

1. In the console, click **Access control** > **User roles**.
2. From the list of user roles, click the one you want to have task permissions.
3. On the **Permissions** tab, in the **Type** box, select **Tasks**.
4. For **Permission**, select **Run tasks**, and then select a task from the **Object** list. For example, **factor_task**.
5. Click **Add permission**, and then commit the change.

Using token authentication

Before running an orchestrator job, you must generate an RBAC access token to authenticate to the orchestration service. If you attempt to run a job without a token, PE prompts you to supply credentials.

For information about generating a token with the CLI, see the documentation on token-based authentication.

Related information

[Token-based authentication](#) on page 242

Puppet Enterprise users generate tokens to authenticate their access to certain PE command-line tools and API endpoints. Authentication tokens are tied to the permissions granted to the user through RBAC, and provide the user with the appropriate access to HTTP requests.

[Create a new user role](#) on page 235

RBAC has four predefined roles: Administrators, Code Deployers, Operators, and Viewers. You can also define your own custom user roles.

[Assign permissions to a user role](#) on page 235

You can mix and match permissions to create custom user roles that provide users with precise levels of access to PE actions.

Enable cached catalogs for use with the orchestrator (optional)

Enabling cached catalogs on your agents ensures Puppet does not enforce any catalog changes on your agents until you run an orchestrator job to enforce changes.

When you use the orchestrator to enforce change in a Puppet environment (for example, in the `production` environment), you want the agents in that environment to rely on their cached catalogs until you run an orchestrator job that includes configuration changes for those agents. Agents in such environments check in during the run interval (30 minutes by default) to reinforce configuration in their cached catalogs, and apply new configuration only when you run Puppet with an orchestration job.

Note: This is an optional configuration. You can run Puppet on nodes with the orchestrator in workflows that don't require cached catalogs.

1. Run Puppet on the new agents.

Important: Be sure you run Puppet on the new agents **before** assigning any application components to them or performing the next step.

2. In each agent's `puppet.conf` file, in the `[agent]` section, add `use_cached_catalog=true`. To complete this step, choose one of the following methods:

- From the command line on each agent machine, run the following command:

```
puppet config set use_cached_catalog true --section agent
```

- Add an `ini_setting` resource in the `node default {}` section of the environment's `site.pp`. This adds the setting to **all** agents in that environment.

```
if $facts['kernel'] = 'windows' {
  $config = 'C:/ProgramData/PuppetLabs/puppet/etc/puppet.conf'
} else {
  $config = $settings::config
}

ini_setting { 'use_cached_catalog':
  ensure => present,
  path   => $config,
  section => 'agent',
  setting => 'use_cached_catalog',
  value  => 'true',
}
```

3. Run Puppet on the agents again to enforce this configuration.

Orchestrator configuration files

The configuration file for the orchestrator allows you to run commands from the CLI without having to pass additional flags. Whether you are running the orchestrator from the Puppet master or from a separate work station, there are two types of configuration files: a global configuration file and a user-specified configuration file.

Orchestrator global configuration file

If you're running the orchestrator from a PE-managed machine, on either the Puppet master or an agent node, PE manages the global configuration file.

This file is installed on both managed and non-managed workstations at:

- ***nix systems** --- `/etc/puppetlabs/client-tools/orchestrator.conf`
- **Windows** --- `C:/ProgramData/PuppetLabs/client-tools/orchestrator.conf`

The class that manages the global configuration file is `puppet_enterprise::profile::controller`. The following parameters and values are available for this class:

Parameter	Value
<code>manage_orchestrator</code>	true or false (default is true)
<code>orchestrator_url</code>	url and port (default is Puppet master url and port 8143)

The only value PE sets in the global configuration file is the `orchestrator_url` (which sets the orchestrator's `service-url` in `/etc/puppetlabs/client-tools/orchestrator.conf`).

Important: If you're using a managed workstation, do not edit or change the global configuration file. If you're using an unmanaged workstation, you can edit this file as needed.

Orchestrator user-specified configuration file

You can manually create a user-specified configuration file and populate it with orchestrator configuration file settings. PE does not manage this file.

This file needs to be located at `~/.puppetlabs/client-tools/orchestrator.conf` for both *nix and Windows.

If present, the user specified configuration always takes precedence over the global configuration file. For example, if both files have contradictory settings for the **environment**, the user specified settings prevail.

Orchestrator configuration file settings

The orchestrator configuration file is formatted in JSON. For example:

```
{
  "options" : {
    "service-url": "https://<PUPPET MASTER HOSTNAME>:8143",
    "cacert": "/etc/puppetlabs/puppet/ssl/certs/ca.pem",
    "token-file": "~/.puppetlabs/token",
    "color": true
  }
}
```

The orchestrator configuration files (the user-specified or global files) can take the following settings:

Setting	Definition
service-url	The URL that points to the Puppet master and the port used to communicate with the orchestration service. (You can set this with the <code>orchestrator_url</code> parameter in the <code>puppet_enterprise::profile::controller</code> class.) Default value: <code>https://<PUPPET MASTER HOSTNAME>:8143</code>
environment	The environment used when you issue commands with Puppet orchestrator.
cacert	The path for the Puppet Enterprise CA cert. <ul style="list-style-type: none"> *nix: <code>/etc/puppetlabs/puppet/ssl/certs/ca.pem</code> Windows: <code>C:\ProgramData\PuppetLabs\puppet\etc\ssl\certs\ca.pem</code>
token-file	The location for the authentication token. Default value: <code>~/.puppetlabs/token</code>
color	Determines whether the orchestrator output uses color. Set to <code>true</code> or <code>false</code> .
noop	Determines whether the orchestrator runs the Puppet agent in no-op mode. Set to <code>true</code> or <code>false</code> .

PE Bolt services configuration

The PE Bolt server is a Ruby service that enables SSH and WinRM tasks from the console. The server accepts requests from the orchestrator, calls out to Bolt to run the task, and returns the result.

PE Bolt server configuration

The PE Bolt server provides an API for running tasks over SSH and WinRM using Bolt, which is the technology underlying PE tasks. The API server for tasks is available as `pe-bolt-server`.

The server is a [Puma](#) application that runs as a standalone service.

The server is configured in `/etc/puppetlabs/bolt-server/conf.d/bolt-server.conf`, managed by the `puppet_enterprise::profile::bolt_server` class, which includes these parameters:

Setting	Type	Description	Default
<code>bolt_server_loglevel</code>	string	Bolt log level. Acceptable values are <code>debug</code> , <code>info</code> , <code>notice</code> , <code>warn</code> , and <code>error</code> .	<code>notice</code>
<code>concurrency</code>	integer	Maximum number of server threads.	<code>100</code>
<code>master_host</code>	string	URI of the master where Bolt can download tasks.	<code>\$puppet_enterprise::puppet_ma</code>
<code>master_port</code>	integer	Port the Bolt server can access the master on.	<code>\$puppet_enterprise::puppet_ma</code>
<code>ssl_cipher_suites</code>	array[string]	TLS cipher suites in order of preference.	<code>\$puppet_enterprise::params::s</code>

Setting	Type	Description	Default
ssl_listen_port	integer	Port the Bolt server runs on.	62658 (\$puppet_enterprise::bolt_server_ssl_listen_port)
whitelist	array[string]	List of hosts that can connect to pe-bolt-server.	[\$certname]

PE ACE services configuration

The PE Agentless Catalog Executor server is a Ruby service that enables you to execute Bolt Tasks and Puppet catalogs remotely.

PE ACE server configuration

The PE ACE server provides an API for running tasks and catalogs against remote targets.

The server is a [Puma](#) application that runs as a standalone service.

The server is configured in `/etc/puppetlabs/ace-server/conf.d/ace-server.conf`, managed by the `puppet_enterprise::profile::ace_server` class, which includes these parameters:

Setting	Type	Description	Default
service_loglevel	string	Bolt log level. Acceptable values are debug, info, notice, warn, and error.	notice
concurrency	integer	Maximum number of server threads.	\$puppet_enterprise::ace_server_concurrency
master_host	string	URI that ACE can access the master on.	pe_repo::compile_master_pool_host default: \$puppet_enterprise::puppet_master_host
master_port	integer	Port that ACE can access the master on.	\$puppet_enterprise::puppet_master_port
hostcrl	string	The host CRL path	\$puppet_enterprise::params::hostcrl
ssl_cipher_suites	array[string]	TLS cipher suites in order of preference.	\$puppet_enterprise::params::ssl_cipher_suites
ssl_listen_port	integer	Port that ACE runs on.	44633 (\$puppet_enterprise::ace_server_ssl_listen_port)
whitelist	array[string]	List of hosts that can connect to pe-ace-server.	[\$certname]

Disabling application management or orchestration services

Both application management and orchestration services are on by default in PE. If you need to disable these services, refer to [Disabling application management](#) and [Disabling orchestration services](#).

Using Bolt with orchestrator

Bolt enables running a series of tasks — called *plans* — to help you automate the manual work of maintaining your infrastructure. When you pair Bolt with PE, you get advanced automation with the management and logging capabilities of PE.

Bolt connects directly to remote nodes with SSH or WinRM, so you are not required to install any agent software. To learn more about Bolt, see the [Bolt documentation](#).

You can configure Bolt to use the orchestrator API and perform actions on PE nodes. When you run Bolt plans, the plan logic is processed locally on the node running Bolt while corresponding commands, scripts, tasks, and file uploads run remotely using the orchestrator API.

Before you can use Bolt with PE, you must [install Bolt](#).

To set up Bolt to use the orchestrator API, you must:

- Install the `bolt_shim` module in a PE environment.
- Assign task permissions to a user role.
- Adjust the orchestrator configuration files, as needed.
- Configure Bolt to connect to PuppetDB.

Install the Bolt module in a PE environment

Bolt uses a task to execute commands, upload files, and run scripts over orchestrator. To install this task, install the [puppetlabs-bolt_shim module](#) from the Forge. Install the code in the same environment as the other tasks you want to run. Use the following Puppetfile line:

```
mod 'puppetlabs-bolt_shim', '0.3.0'
```

In addition to the `bolt_shim` module, any task or module content you want to execute over Puppet Communications Protocol (PCP) must be present in the PE environment. For details about downloading and installing modules for Bolt, see [Set up Bolt to download and install modules](#). By allowing only content that is present in the PE environment to be executed over PCP, you maintain role-based access control over the nodes you manage in PE.

To enable the Boltapply action, you must install the [puppetlabs-apply_helpers module](#). Use the following Puppetfile line:

```
mod 'puppetlabs-apply_helpers', '0.1.0'
```

Note: Bolt over orchestrator can require a large amount of memory to convey large messages, such as the plugins and catalogs sent by apply. You might need to [increase the Java heap size](#) for orchestration services.

Assign task permissions to a user role



CAUTION: By granting users access to Bolt tasks, you give them permission to run arbitrary commands and upload files as a super-user.

1. In the console, click **Access control** > **User roles**.
2. From the list of user roles, click the role you want to have task permissions.
3. On the **Permissions** tab, in the **Type** box, select **Tasks**.
4. For **Permission**, select **Run tasks**, and select **All** from the **Instance** drop-down list.
5. Click **Add permission**, and commit the change.

Adjust the orchestrator configuration files

Set up the orchestrator API for Bolt in the same configuration file that is used for PE client tools:

- ***nix** `/etc/puppetlabs/client-tools/orchestrator.conf`

- **Windows** `C:/ProgramData/PuppetLabs/client-tools/orchestrator.conf`

Note: If you use a global configuration file stored at `/etc/puppetlabs/client-tools/orchestrator.conf` (or `C:\ProgramData\PuppetLabs\client-tools\orchestrator.conf` for Windows), copy the file to your home directory.

Alternatively, you can configure Bolt to connect to orchestrator in the `pcp` section of the Bolt configuration file. This configuration is not shared with `puppet task`. By default, Bolt uses the production environment in PE when running tasks. To use a different environment, change the `task-environment` setting:

```
pcp:
  task-environment: development
```

Configure Bolt to connect to PuppetDB

Bolt can authenticate with PuppetDB through an SSL client certificate or a PE RBAC token. For more information see the Bolt docs for [Connecting Bolt to PuppetDB](#).

Specify the transport

Bolt runs tasks through the orchestrator when a target uses the `pcp` transport. Specify the transport for specific nodes by using the PCP protocol in the target's URI, like `pcp://puppet.certname`, or setting `transport` in a `config` section in `inventory.yaml`. Change the default transport for all nodes by setting `transport` in `bolt.yaml` or passing `--transport pcp` on the command line.

View available tasks

To view a list of available tasks from the orchestrator API, run the command `puppet task show` (instead of the command `bolt task show`).

Direct Puppet: a workflow for controlling change

The orchestrator—used alongside other PE tools, such as Code Manager—allows you to control when and how infrastructure changes are made before they reach your production environment.

The Direct Puppet workflow gives you precise control over rolling out changes, from updating data and classifying nodes, to deploying new Puppet code. In this workflow, you configure your agents to use cached catalogs during scheduled runs, and you send new catalogs only when you're ready, via orchestrator jobs. Scheduled runs continue to enforce the desired state of the last orchestration job until you send another new catalog.

Related information

[Enable cached catalogs for use with the orchestrator \(optional\)](#) on page 425

Enabling cached catalogs on your agents ensures Puppet does not enforce any catalog changes on your agents until you run an orchestrator job to enforce changes.

Direct Puppet workflow

In this workflow, you set up a node group for testing and validating code on a feature branch before you merge and promote it into your production environment.

Before you begin

- To use this workflow, you must enable cached catalogs for use with the orchestrator so that they enforce cached catalogs by default and compile new catalogs only when instructed to by orchestrator jobs.
- This workflow also assumes you're familiar with Code Manager. It involves making changes to your control repo—adding or updating modules, editing manifests, or changing your Hiera data. You'll also run deploy actions from the Code Manager command line tool and the orchestrator, so ensure you have access to a host with PE client tools installed.

Related information

[Installing PE client tools](#) on page 169

PE client tools are a set of command line tools that let you access Puppet Enterprise services from a workstation that might or might not be managed by Puppet.

Set up node groups for testing new features

The first step in the Direct Puppet workflow is to set up node groups for testing your new feature or code.

1. If they don't already exist, create environment node groups for branch testing, for example, you might create `Development environment` and `Test environment` node groups.
2. Within each of these environment node groups, create a child node group to enable on-demand testing of changes deployed in Git feature branch Puppet environments.
You now have at three levels of environment node groups: 1) the top-level parent environment node group, 2) node groups that represent your actual environments, and 3) node groups specific to feature testing.
3. In the **Rules** tab of the child node groups you created in the previous step, add this rule:

Option	Value
Fact	<code>agent_specified_environment</code>
Operator	<code>~</code>
Value	<code>^ . +</code>

This rule matches any nodes from the parent group that have the **agent_specified_environment** fact set. By matching nodes to this group, you give the nodes permission to override the server-specified environment and use their agent-specified environment instead.

Related information

[Create environment node groups](#) on page 322

Create custom environment node groups so that you can target deployment of Puppet code.

Create a feature branch

After you've set up a node group, create a new branch of your control repository on which you can make changes to your feature code.

1. Branch your control repository, and name the new branch, for example, `my_feature_branch`.

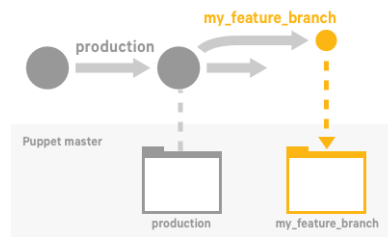


2. Make changes to the code on your feature branch. Commit and push the changes to the `my_feature_branch`.

Deploy code to the Puppet master and test it

Now that you've made some changes to the code on your feature branch, you're ready to use Code Manager to push those to the Puppet master.

1. To deploy the feature branch to the master, run the following Code Manager command: `puppet code deploy --wait my_feature_branch`

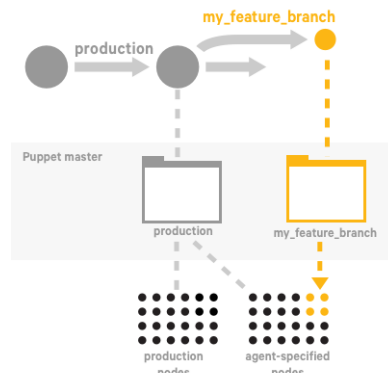


Note: After this code deployment, there is a short delay while Puppet Server loads the new code.

2. To test your changes, run Puppet on a few agent-specified development nodes in the **my_feature_branch** environment, run the following orchestrator command:

```
puppet job run --nodes my-dev-node1,my-dev-node2 --environment
my_feature_branch
```

Tip: You can also use the console to create a job targeted at a list of nodes in the **my_feature_branch** environment.



3. Validate your testing changes. Open the links in the orchestrator command output, or use the Job ID linked on the Job list page, to review the node run reports in the console. Ensure the changes have the effect you intend.

Related information

[Review jobs from the console](#) on page 505

View a list of recent jobs and drill down to see useful details about each one.

[Run Puppet on a node list](#) on page 434

Create a node list target for a job when you need to run Puppet on a specific set of nodes that isn't easily defined by a PQL query.

Merge and promote your code

If everything works as expected on the development nodes, and you're ready to promote your changes into production.

1. Merge `my_feature_branch` into the `production` branch in your control repo.



2. To deploy your updated `production` branch to the Puppet master, run the following Code Manager command: `puppet code deploy --wait production`

Preview the job

Before running Puppet across the `production` environment, preview the job with the `puppet job plan` command.

To ensure the job captures all the nodes in the `production` environment, as well as the agent-specified development nodes that just ran with the `my_feature_branch` environment, use the following query as the job target:

```
puppet job plan --query 'inventory {environment in ["production",
  "my_feature_branch"]}'
```

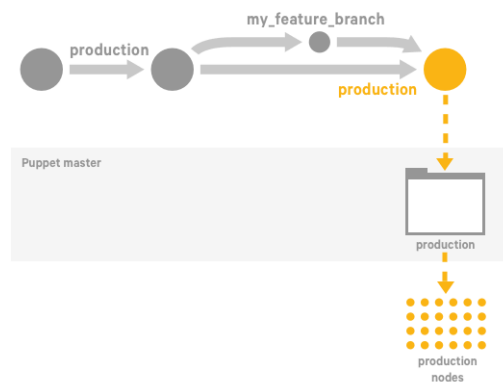
Run the job on the production environment

If you're satisfied with the changes in the preview, you're ready to enforce changes to the `production` environment.

Run the orchestrator job.

```
puppet job run --query 'inventory {environment in ["production",
  "my_feature_branch"]}'
```

Tip: You can also use the console to create a job targeted at this PQL query.



Related information

[Run Puppet on a PQL query](#) on page 435

For some jobs, you might want to target nodes that meet specific conditions. For such jobs, create a PQL query.

Validate your production changes

Finally, you're ready to validate your production changes.

Check the node run reports in the console to confirm that the changes were applied as intended. If so, you're done!

Repeat this process as you develop and promote your code.

Running Puppet on demand

The orchestrator gives you the ability to set up jobs in the console or on the command line to trigger on-demand Puppet runs.

Running Puppet on demand from the console

When you set up a job to run Puppet from the console, the orchestrator creates a job ID to track the job, shows you all nodes included in the job, and runs Puppet on those nodes in the appropriate order. Puppet compiles a new catalog for each node included in the job.

There are two ways to specify the job target (the nodes you want to run jobs on):

- A static node list
- A Puppet Query Language (PQL) query
- A node group

You can't combine these methods, and if you switch from one to the other with the **Inventory** drop-down list, the target list clears and starts over. In other words, if you create a target list by using a node list, switching to a PQL query clears the node list, and vice versa. You can do a one-time conversion of a PQL query to a static node list if you want to add or remove nodes from the query results.

Before you start, be sure you have the correct permissions for running jobs. To run jobs on PQL queries, you need the "View node data from PuppetDB" permission.

Run Puppet on a node list

Create a node list target for a job when you need to run Puppet on a specific set of nodes that isn't easily defined by a PQL query.

Before you begin

Make sure you have access to the nodes you want to target.

Make sure you have permissions to run jobs and queries.

Tip: You can add network devices to a node list when you have installed modules for device transports in your production environment. You can find such modules in [Puppet Forge](#).

1. In the console, in the **Run** section, click **Puppet**.
2. Optional: In the **Job description** field, provide a description. What you enter here appears on the job list and job details pages.
3. Optional: Under **Schedule**, select **Later** and choose a start date, time, and time zone for the job to run.

Tip: Based on the configuration of your console, the default time zone is either UTC, Coordinated Universal Time, or the local time, UTC offset, of your browser.

4. Select an environment:

- **Run nodes in their own assigned environment:** Nodes run in the environment specified by the Node Manager or their `puppet.conf` file.
- **Select an environment for nodes to run in:** Select an environment from the list. Nodes can run in an environment if their environment is agent-specified, if they're included in an application in that environment, or if they're classified in that environment by the node manager.

Note: If your job target includes application instances, the selected environment also determines the dependency order of your node runs.

5. Select the run mode for the job. The default run mode for a job always attempts to enforce new catalogs on nodes. To change the run mode, use the following selections:
 - **No-op:** Simulate a run on all nodes in the job without enforcing a new catalog.
 - **Override noop = true configuration:** If any nodes in the job have `noop = true` set in their `puppet.conf` files, ignores that setting and enforce a new catalog on those nodes. This setting corresponds to the `--no-noop` flag available on the orchestrator CLI.
6. In the **Inventory** list, select **Node list**.
7. To create a node list, in the search field, start typing in the names of nodes to search for, and click **Search**.

Note: The search does not handle regular expressions.
8. Select the nodes you want to add to the job. You can select nodes from multiple searches to create the node list target.
9. Click **Run job**.

Run Puppet on a PQL query

For some jobs, you might want to target nodes that meet specific conditions. For such jobs, create a PQL query.

Before you begin

Make sure you have access to the nodes you want to target.

Make sure you have permissions to run jobs and queries.

1. In the console, in the **Run** section, click **Puppet**.
2. Optional: In the **Job description** field, provide a description. What you enter here appears on the job list and job details pages.
3. Optional: Under **Schedule**, select **Later** and choose a start date, time, and time zone for the job to run.

Tip: Based on the configuration of your console, the default time zone is either UTC, Coordinated Universal Time, or the local time, UTC offset, of your browser.
4. Select an environment:
 - **Run nodes in their own assigned environment:** Nodes run in the environment specified by the Node Manager or their `puppet.conf` file.
 - **Select an environment for nodes to run in:** Select an environment from the list. Nodes can run in an environment if their environment is agent-specified, if they're included in an application in that environment, or if they're classified in that environment by the node manager.

Note: If your job target includes application instances, the selected environment also determines the dependency order of your node runs.
5. Select the run mode for the job. The default run mode for a job always attempts to enforce new catalogs on nodes. To change the run mode, use the following selections:
 - **No-op:** Simulate a run on all nodes in the job without enforcing a new catalog.
 - **Override noop = true configuration:** If any nodes in the job have `noop = true` set in their `puppet.conf` files, ignores that setting and enforce a new catalog on those nodes. This setting corresponds to the `--no-noop` flag available on the orchestrator CLI.
6. From the list of target types, select **PQL query**.

7. Specify a target by doing one of the following:

- Enter a query that selects the target you want. See the [Puppet Query Language \(PQL\) reference](#) for more information.
- Click **Common queries**. Select one of the queries and replace the defaults in the braces ({ }) with values that specify the target you want.

Note: These queries include [certname] as [<projection>] to restrict the output.

Target	PQL query
All nodes	<code>nodes[certname] { }</code>
Nodes with a specific resource (example: httpd)	<code>resources[certname] { type = "Service" and title = "httpd" }</code>
Nodes with a specific fact and value (example: OS name is)	<code>inventory[certname] { facts.os.name = "<OS>" }</code>
Nodes with a specific report status (example: last run failed)	<code>reports[certname] { latest_report_status = "failed" }</code>
Nodes with a specific class (example:)	<code>resources[certname] { type = "Class" and title = "Apache" }</code>
Nodes assigned to a specific environment (example: production)	<code>nodes[certname] { catalog_environment = "production" }</code>
Nodes with a specific version of a resource type (example: OpenSSL is v1.1.0e)	<code>resources[certname] { type = "Package" and title="openssl" and parameters.ensure = "1.0.1e-51.el7_2.7" }</code>
Nodes with a specific resource and operating system (example: httpd and)	<code>inventory[certname] { facts.operatingsystem = "CentOS" and resources { type = "Service" and title = "httpd" } }</code>

8. Click **Submit query** and click **Refresh** to update the node results.

9. If you change or edit the query after it runs, click **Submit query** again.

10. Optional: To convert the query target results to a node list, for use as a node list target, click **Convert query to static node list**.

Note: If you select this option, the job target becomes a node list. You can add or remove nodes from the node list before running the job, but you cannot edit the query.

11. Click **Run job**.

Important: When you run this job, the PQL query runs again, and the job might run on a different set of nodes than what is currently displayed. If you want the job to run only on the list as displayed, convert the query to a static node list before you run the job.

Run Puppet on a node group

Create a node target for a job when you need to run Puppet on a specific set of nodes in a pre-defined group.

Before you begin

Make sure you have access to the nodes you want to target.

Note: If you do not have permissions to view a node group, or the node group doesn't have any matching nodes, that node group won't be listed as an option for viewing. In addition, a node group does not appear if no rules have been specified for it.

1. In the console, in the **Run** section, click **Puppet**.
2. Optional: In the **Job description** field, provide a description. What you enter here appears on the job list and job details pages.
3. Optional: Under **Schedule**, select **Later** and choose a start date, time, and time zone for the job to run.
Tip: Based on the configuration of your console, the default time zone is either UTC, Coordinated Universal Time, or the local time, UTC offset, of your browser.
4. Select an environment:
 - **Run nodes in their own assigned environment:** Nodes run in the environment specified by the Node Manager or their `puppet.conf` file.
 - **Select an environment for nodes to run in:** Select an environment from the list. Nodes can run in an environment if their environment is agent-specified, if they're included in an application in that environment, or if they're classified in that environment by the node manager.

Note: If your job target includes application instances, the selected environment also determines the dependency order of your node runs.
5. Select the run mode for the job. The default run mode for a job always attempts to enforce new catalogs on nodes. To change the run mode, use the following selections:
 - **No-op:** Simulate a run on all nodes in the job without enforcing a new catalog.
 - **Override noop = true configuration:** If any nodes in the job have `noop = true` set in their `puppet.conf` files, ignores that setting and enforce a new catalog on those nodes. This setting corresponds to the `--no-noop` flag available on the orchestrator CLI.
6. From the list of target types, select **Node group**.
7. In the **Choose a node group** box, type or select a node group, and click **Select**.
8. Click **Run job**.

Run jobs throughout the console

You don't need to be in the Jobs section of the console to run a Puppet job on your nodes. You might encounter situations where you want to run jobs on lists of nodes derived from different pages in the console.

You can create jobs from the following pages:

Page	Description
Overview	This page shows a list of all your managed nodes, and gathers essential information about your infrastructure at a glance.
Events	Events let you view a summary of activity in your infrastructure, analyze the details of important changes, and investigate common causes behind related events. For instance, let's say you notice run failures because some nodes have out-of-date code. After you update the code, you can create a job target from the list of failed nodes to be sure you're directing the right fix to the right nodes. You can create new jobs from the Nodes with events category.
Classification node groups	Node groups are used to automate classification of nodes with similar functions in your infrastructure. If you make a classification change to a node group, you can quickly create a job to run Puppet on all the nodes in that group, pushing the change to all nodes at one time.

Make sure you have permissions to run jobs and queries.

1. In the console, in the **Run** section, click **Puppet**.

At this point, the list of nodes is converted to a new Puppet run job list target.

2. Optional: In the **Job description** field, provide a description. What you enter here appears on the job list and job details pages.
3. Select an environment:
 - **Run nodes in their own assigned environment:** Nodes run in the environment specified by the Node Manager or their `puppet.conf` file.
 - **Select an environment for nodes to run in:** Select an environment from the list. Nodes can run in an environment if their environment is agent-specified, if they're included in an application in that environment, or if they're classified in that environment by the node manager.

Note: If your job target includes application instances, the selected environment also determines the dependency order of your node runs.
4. Select the run mode for the job. The default run mode for a job always attempts to enforce new catalogs on nodes. To change the run mode, use the following selections:
 - **No-op:** Simulate a run on all nodes in the job without enforcing a new catalog.
 - **Override noop = true configuration:** If any nodes in the job have `noop = true` set in their `puppet.conf` files, ignores that setting and enforce a new catalog on those nodes. This setting corresponds to the `--no-noop` flag available on the orchestrator CLI.
5. **Important:** Do not change the Inventory from **Node list** to **PQL query**. This clears the node list target.
6. Click **Run job**.

Schedule a Puppet run

Schedule a job to deploy configuration changes at a particular date and time.

Before you begin

Make sure you have access to the nodes you want to target.

1. In the console, in the **Run** section, click **Puppet**.
2. Optional: In the **Job description** field, provide a description. What you enter here appears on the job list and job details pages.
3. Select an environment:
 - **Run nodes in their own assigned environment:** Nodes run in the environment specified by the Node Manager or their `puppet.conf` file.
 - **Select an environment for nodes to run in:** Select an environment from the list. Nodes can run in an environment if their environment is agent-specified, if they're included in an application in that environment, or if they're classified in that environment by the node manager.

Note: If your job target includes application instances, the selected environment also determines the dependency order of your node runs.
4. Under **Schedule**, select **Later** and choose a start date, time, and time zone for the job to run.

Tip: Based on the configuration of your console, the default time zone is either UTC, Coordinated Universal Time, or the local time, UTC offset, of your browser.
5. Optional: To repeat the job on a regular schedule, change the run frequency from **Once** to **Hourly**, **Daily**, or **Weekly**.

Note: If a recurring job runs longer than the selected frequency, the following job waits to start until the next available time interval.

6. Select the run mode for the job. The default run mode for a job always attempts to enforce new catalogs on nodes. To change the run mode, use the following selections:
 - **No-op:** Simulate a run on all nodes in the job without enforcing a new catalog.
 - **Override noop = true configuration:** If any nodes in the job have `noop = true` set in their `puppet.conf` files, ignores that setting and enforce a new catalog on those nodes. This setting corresponds to the `--no-noop` flag available on the orchestrator CLI.
7. From the list of target types, select the category of nodes you want to target.
 - **Node list** Add individual nodes by name.
 - **PQL Query** Use the query language to retrieve a list of nodes.
 - **Node group** Select an existing node group.
8. Click **Schedule job**.

Your job appears on the **Schedule Puppet run** tab of the **Jobs** page.

Stop a job in progress

You can stop a job if, for example, you realize you need to reconfigure a class or push a configuration change that the job needs.

When you stop an on-demand Puppet run, runs that are underway finish, and runs that have not started are canceled.

Stopping tasks depends on how many nodes are targeted in the job versus the concurrency limit you've set. If the concurrency limit is higher than the number of nodes targeted, all nodes complete the task, as those nodes have already started the task.

To stop a job:

- In the console, go to the **Job details** page and select the **Puppet run** or **Task** tab. From the list of jobs, find the one you want and click **Stop**.
- On the command line, press **CTRL + C**.

Delete a scheduled job

Delete a job that is scheduled to run at a later time.

Tip: You must have the appropriate role-based permissions to delete another user's scheduled job.

1. In the console, open the **Jobs** page.
2. Open the tab for the type of job you want to delete.
 - **Scheduled Puppet Run**
 - **Scheduled task**
3. From the list of jobs, find the one you want to delete and click **Remove**.

Running Puppet on demand from the CLI

Use the **puppet job run** command to enforce change on your agent nodes with on-demand Puppet runs.

Use the **puppet job** tool to enforce change across nodes. For example, when you add a new class parameter to a set of nodes or deploy code to a new Puppet environment, use this command to run Puppet across all the nodes in that environment.

If you run a job on a node that has relationships outside of the target (for example, it participates in an application that includes nodes not in the job target) the job still runs only on the node in the target you specified. In such cases, the orchestrator notifies you that external relationships exist. It prints the node with relationships, and it prints the applications that might be affected. For example:

```
**WARNING** target does not contain all nodes in this application.
```

You can run jobs on three types of targets, but these targets cannot be combined:

- An application or an application instance in an environment

- A list of nodes or a single node
- A PQL nodes query

When you execute a `puppet job run` command, the orchestrator creates a new Job ID, shows you all nodes included in the job, and proceeds to run Puppet on all nodes in the appropriate order. Puppet compiles a new catalog for all nodes included in the job.

The orchestrator command line tool includes the `puppet job` and `puppet app` commands.

- `puppet job`

The `puppet job` command is the main mechanism to control Puppet jobs. For a complete reference of the `puppet job` command, see:

- Checking job plans on the CLI
- Running jobs on the CLI
- Reviewing jobs on the CLI

- `puppet app`

The `puppet app` tool lets you view the application models and application instances you've written and stored on your Puppet master. For a complete reference of the `puppet app` command, see [Reviewing applications on the CLI](#).

Run Puppet on a PQL query

Use a PQL nodes query as a target when you want to target nodes that meet specific conditions. In this case, the orchestrator runs on a list of nodes returned from a PQL nodes query.

Before you begin

The first time you run a command, you need to authenticate. See the [orchestrator installation instructions](#) for information about setting RBAC permissions and token authorization.

If you're running this command from a managed or non-managed workstation, you must specify the full path to the command. For example, `c:\Program Files\Puppet Labs\Client\bin\puppet-task run`.

Install the tasks you want to use.

Make sure you have permission to run the tasks on all nodes.

Make sure you have permission to run the tasks.

Make sure you have access to the nodes you want to target.

Tip: You can add network devices to a node list when you have installed modules for device transports in your production environment. You can find such modules in [Puppet Forge](#).

Make sure you have permissions to run tasks and queries.

Make sure you have permissions to run jobs and queries.

Make sure you have permissions to run jobs and queries.

Log into your Puppet master or client tools workstation and run one of the following commands:

- To specify the query on the command line: `puppet job run --query '<QUERY>' <OPTIONS>`
- To pass the query in a text file: `puppet job run --query @/path/to/file.txt`

The following table shows some example targets and the associated queries you could run with the orchestrator.

Be sure to wrap the entire query in single quotes and use double quotes inside the query.

Target	PQL query
Single node by certname	<code>--query 'nodes { certname = "mynode" }'</code>
All nodes with "web" in certname	<code>--query 'nodes { certname ~ "web" }'</code>

Target	PQL query
All nodes	<code>--query 'inventory { facts.os.name = "CentOS" }'</code>
All nodes with httpd managed	<code>--query 'inventory { facts.operatingsystem = "CentOS" and resources { type = "Service" and title = "httpd" } }'</code>
All nodes with failed reports	<code>--query 'reports { latest_report? = true and status = "failed" }'</code>
All nodes matching the environment for the last received catalog	<code>--query 'nodes { catalog_environment = "production" }'</code>

Run Puppet on a list of nodes or a single node

Use a node list target for an orchestrator job when you need to run a job on a specific set of nodes that don't easily resolve to a PQL query. Use a single node or a comma-separated list of nodes.

Before you begin

The first time you run a command, you need to authenticate. See the [orchestrator installation instructions](#) for information about setting RBAC permissions and token authorization.

If you're running this command from a managed or non-managed workstation, you must specify the full path to the command. For example, `c:\Program Files\Puppet Labs\Client\bin\puppet-task run`.

Install the tasks you want to use.

Make sure you have permission to run the tasks on all nodes.

Make sure you have permission to run the tasks.

Make sure you have access to the nodes you want to target.

Tip: You can add network devices to a node list when you have installed modules for device transports in your production environment. You can find such modules in [Puppet Forge](#).

Make sure you have permissions to run tasks and queries.

Make sure you have permissions to run jobs and queries.

Make sure you have permissions to run jobs and queries.

Log into your Puppet master or client tools workstation and run one of the following commands:

- To run a job on a single node:

```
puppet job run --nodes <NODE NAME> <OPTIONS>
```

- To run a job on a list of nodes, use a **comma-separated** list of node names:

```
puppet job run --nodes <NODE NAME>, <NODE NAME>, <NODE NAME>, <NODE NAME>
<OPTIONS>
```

Note: Do not add spaces in the list of nodes.

- To run a job on a node list from a text file:

```
puppet job run --nodes @/path/to/file.txt
```

Note: If passing a list of nodes in the text file, put each node on a separate line.

Run Puppet on a node group

Similar to running Puppet on a list of nodes, you can run it on a node group..

Before you begin

Make sure you have access to the nodes you want to target.

1. Log into your master or client tools workstation.
2. Run the command: `puppet job run --node-group <node-group-id>`

Tip: Use the `/v1/groups` endpoint to retrieve a list node groups and their IDs.

Related information

[GET /v1/groups](#) on page 361

Use the `/v1/groups` endpoint to retrieve a list of all node groups in the node classifier.

[Review jobs from the console](#) on page 505

View a list of recent jobs and drill down to see useful details about each one.

Run Puppet on an application or an application instance in an environment

Use applications as a job target to enforce Puppet runs in order on all nodes found in a specific application instance, or to enforce Puppet runs in order on all nodes that are found in each instance of an application.

Before you begin

The first time you run a command, you need to authenticate. See the [orchestrator installation instructions](#) for information about setting RBAC permissions and token authorization.

If you're running this command from a managed or non-managed workstation, you must specify the full path to the command. For example, `c:\Program Files\Puppet Labs\Client\bin\puppet-task run`.

Install the tasks you want to use.

Make sure you have permission to run the tasks on all nodes.

Make sure you have permission to run the tasks.

Make sure you have access to the nodes you want to target.

Tip: You can add network devices to a node list when you have installed modules for device transports in your production environment. You can find such modules in [Puppet Forge](#).

Make sure you have permissions to run tasks and queries.

Make sure you have permissions to run jobs and queries.

Make sure you have permissions to run jobs and queries.

Log into your Puppet master or client tools workstation and run one of the following commands:

- To run a job on all instances of an application: `puppet job run --application <APPLICATION> --environment <ENVIRONMENT>`
- To run a job on an instance of an application in an environment: `puppet job run --application <APPLICATION INSTANCE> --environment <ENVIRONMENT>`

Tip: You can use `-a` in place of `--application`.

puppet job run command options

The following are common options you can use with the run action. For a complete list of global options run `puppet job --help`.

Option	Value	Description
<code>--noop</code>	Flag, default false	Run a job on all nodes to simulate changes from a new catalog without actually enforcing a new catalog. Cannot be used in conjunction with <code>--no-noop</code> flag.
<code>--no-noop</code>	Flag	All nodes run in enforcement mode, and a new catalog is enforced on all nodes. This flag overrides the <code>agent noop = true</code> in <code>puppet.conf</code> . Cannot be used in conjunction with <code>--noop</code> flag.
<code>--environment, -e</code>	Environment name	Overrides the environment specified in the orchestrator configuration file. The orchestrator uses this option to: <ul style="list-style-type: none"> • Instruct nodes what environment to run in. If any nodes can't run in the environment, those node runs fail. A node runs in an environment if: <ul style="list-style-type: none"> • The node is included in an application in that environment. These runs may fail if the node is classified into a different environment in the <code>node</code> classifier. • The node is classified into that environment in the <code>node</code> classifier. • Load the application code used to plan run order
<code>--no-enforce-environment</code>	Flag, default false	Ignores the environment set by the <code>--environment</code> flag for agent runs. When you use this flag, agents run in the environment specified by the PE Node Manager or their <code>puppet.conf</code> files.
<code>--description</code>	Flag, defaults to empty	Provide a description for the job, to be shown on the job list and job details pages, and returned with the <code>puppet job show</code> command.
<code>--concurrency</code>	Integer	Limits how many nodes can run concurrently. Default is unlimited. You can tune concurrent compile requests in the console.

Post-run node status

After Puppet runs, the orchestrator returns a list of nodes and their run statuses.

Node runs can be in progress, completed, skipped, or failed.

- For a completed node run, the orchestrator prints the configuration version, the transaction ID, a summary of resource events, and a link to the full node run report in the console.
- For an in progress node run, the orchestrator prints how many seconds ago the run started.
- For a failed node run, the orchestrator prints an error message indicating why the run failed. In this case, any additional runs are skipped.

When a run fails, the orchestrator also prints any applications that were affected by the failure, as well as any applications that were affected by skipped node runs.

You can view the status of all running, completed, and failed jobs with the `puppet job show` command, or you can view them from the **Job details** page in the console.

Additionally, you can use the console to review a list of jobs or to view the details of jobs that have previously run or are in progress. Refer to [Reviewing jobs in the PE console](#) for more information.

Stop a job in progress

You can stop a job if, for example, you realize you need to reconfigure a class or push a configuration change that the job needs.

When you stop an on-demand Puppet run, runs that are underway finish, and runs that have not started are canceled.

Stopping tasks depends on how many nodes are targeted in the job versus the concurrency limit you've set. If the concurrency limit is higher than the number of nodes targeted, all nodes complete the task, as those nodes have already started the task.

To stop a job:

- In the console, go to the **Job details** page and select the **Puppet run** or **Task** tab. From the list of jobs, find the one you want and click **Stop**.
- On the command line, press **CTRL + C**.

Viewing job plans

The `puppet job plan` command allows you to preview a plan for a Puppet job without actually enforcing any change. Use this command to ensure your job runs as expected.

You can preview plans for three types of targets, but these targets cannot be combined:

- View a plan to enforce change on a list of nodes or a single node
- View a plan to enforce change based on a PQL nodes query
- View a plan to enforce change on an application or an application instance in an environment

Results from the job plan command

The `puppet job plan` command returns the following about a job:

- The environment the job runs in.
- The target for the job.
- The maximum concurrency setting for the job.
- The total number of nodes in the job run.
- A list of application instances included in the job, if applicable.
- A list of nodes sorted topographically, with components and application instances listed below each node. The node list is organized in levels by dependencies. Nodes shown at the top, level 0, have no dependencies. Nodes in level 1 have dependencies in level 0. A node can run after Puppet has finished running on all its dependencies.

Note: The `puppet job plan` **does not** generate a job ID. A job ID is created, and shown in the job plan, only when you use `puppet job run`. Use the job ID to view jobs with the `puppet job show` command.

After you view the plan:

- Use `puppet job run <SAME TARGET AND OPTIONS>` to create and run a job like this.
- Remember that node catalogs might have changed after this plan was generated.
- Review and address any errors that the service reports. For example, it detects any dependency cycles or components that not properly mapped to nodes.

Example job plan

The following is an example plan summary for a job that runs on a list of nodes that contain a partial application instance.

```
$ puppet job plan --nodes db1.example.com, noapp1.example.com, noapp2.example.com

+-----+-----+
| Environment | production |
| Concurrency limit | none |
| Nodes | 3 |
+-----+-----+

Application instances: 1
- Lamp[example]

Node run order (nodes in each level can run simultaneously, nodes in level 0 will run before their dependent nodes
in level 1, etc.):
0 -----
db1.example.com
  Lamp[example] - Lamp::Db[example]
noapp1.example.com
noapp2.example.com

Started puppet run on db1.example.domain.com ...
Started puppet run on web3.example.domain.com ...
Started puppet run on web2.example.domain.com ...
Started puppet run on web.example.domain.com ...

Use 'puppet job run --nodes db1.example.com, noapp1.example.com, noapp2.example.com' to create and run this job.
Node catalogs may have changed since this plan was generated.
```

View a job plan for a list of nodes or a single node

You can view a plan for enforcing change on a single node or a comma-separated list of nodes.

Before you begin

The first time you run a command, you need to authenticate. See the [orchestrator installation instructions](#) for information about setting RBAC permissions and token authorization.

If you're running this command from a managed or non-managed workstation, you must specify the full path to the command. For example, `c:\Program Files\Puppet Labs\Client\bin\puppet-task run`.

Install the tasks you want to use.

Make sure you have permission to run the tasks on all nodes.

Make sure you have permission to run the tasks.

Make sure you have access to the nodes you want to target.

Tip: You can add network devices to a node list when you have installed modules for device transports in your production environment. You can find such modules in [Puppet Forge](#).

Make sure you have permissions to run tasks and queries.

Make sure you have permissions to run jobs and queries.

Log into your Puppet master or client tools workstation and run one of the following commands:

- To view a plan for a single node: `puppet job plan --nodes <NODE NAME> <OPTIONS>`
- To view a plan for a list of nodes, use a **comma-separated** list of node names. `puppet job plan --nodes <NODE NAME>, <NODE NAME>, <NODE NAME>, <NODE NAME> <OPTIONS>`

Tip: You can use `-n` in place of `--nodes`.

View a job plan for a PQL nodes query

Use a PQL nodes query as a job target. In this case, the orchestrator presents a plan for a job that could run on a list of nodes returned from a PQL nodes query.

1. Ensure you have the correct permissions to use PQL queries.
2. Log into your Puppet master or client tools workstation and run the following command: `puppet job plan --query '<QUERY>' <OPTIONS>`

Tip: You can use `-q` in place of `--query`.

The following table shows some example targets and the associated PQL queries you could run with the orchestrator.

Be sure to wrap the entire query in single quotes and use double quotes inside the query.

Target	PQL query
Single node by certname	<code>--query 'nodes { certname = "mynode" }'</code>
All nodes with "web" in certname	<code>--query 'nodes { certname ~ "web" }'</code>
All CentOS nodes	<code>--query 'inventory { facts.os.name = "CentOS" }'</code>
All CentOS nodes with httpd managed	<code>--query 'inventory { facts.operatingsystem = "CentOS" and resources { type = "Service" and title = "httpd" } }'</code>
All nodes with failed reports	<code>--query 'reports { latest_report? = true and status = "failed" }'</code>
All nodes matching the environment for the last received catalog	<code>--query 'nodes { catalog_environment = "production" }'</code>

View a job plan for applications or application instances

You can view a plan for enforcing change on all instances of an application or for a specific instance of an application.

Log into your Puppet master or client tools workstation and run one of the following commands:

- To view a job plan for all instances of an application: `puppet job plan --application <APPLICATION> <OPTIONS>`
- To view a job plan for an application instance in an environment: `puppet job plan --application <APPLICATION> <INSTANCE> <OPTIONS>`

Tip: You can use `-a` in place of `--application`.

Puppet job plan command options

The following table shows common options for the `puppet job plan` command. For a complete list of options, run `puppet job --help`.

Option	Value	Description
<code>--environment, -e</code>	Environment name	<p>Overrides the environment specified in the orchestrator configuration file. The orchestrator uses this option to:</p> <ul style="list-style-type: none"> Instruct nodes what environment to run in. If any nodes can't run in the environment, those node runs fail. A node runs in an environment if: <ul style="list-style-type: none"> The node is included in an application in that environment. These runs may fail if the node is classified into a different environment in the node classifier. The node is classified into that environment in the node classifier. Load the application code used to plan run order
<code>--concurrency</code>	Integer	Limits how many nodes can run concurrently. Default is unlimited. You can tune concurrent compile requests in the console.

POST /command/deploy

Run the orchestrator across all nodes in an environment.

Request format

The request body must be a JSON object using these keys:

Key	Definition
<code>environment</code>	The environment to deploy. This key is required.
<code>scope</code>	Object, required unless <code>target</code> is specified. The PuppetDB query, a list of nodes, a classifier node group id, or an application/application instance to deploy.
<code>description</code>	String, a description of the job.
<code>noop</code>	Boolean, whether to run the agent in no-op mode. The default is <code>false</code> .
<code>no_noop</code>	Boolean, whether to run the agent in enforcement mode. Defaults to <code>false</code> . This flag overrides <code>noop = true</code> if set in the agent's <code>puppet.conf</code> , and cannot be set to true at the same time as the <code>noop</code> flag.
<code>concurrency</code>	Integer, the maximum number of nodes to run at one time. The default, if unspecified, is unlimited.

Key	Definition
<code>enforce_environment</code>	Boolean, whether to force agents to run in the same environment in which their assigned applications are defined. This key is required to be <code>false</code> if <code>environment</code> is an empty string
<code>debug</code>	Boolean, whether to use the <code>--debug</code> flag on Puppet agent runs.
<code>trace</code>	Boolean, whether to use the <code>--trace</code> flag on Puppet agent runs.
<code>evaltrace</code>	Boolean, whether to use the <code>--evaltrace</code> flag on Puppet agent runs.
<code>target</code>	String, required unless <code>scope</code> is specified. The name of an application or application instance to deploy. If an application is specified, all instances of that application are deployed. If this key is left blank or unspecified without a scope, the entire environment is deployed. This key is deprecated.

For example, to deploy the `node1.example.com` environment in no-op mode, the following request is valid:

```
{
  "environment" : "",
  "noop" : true,
  "scope" : {
    "nodes" : [ "node1.example.com" ]
  }
}
```

Scope

Scope is a JSON object containing exactly one of these keys:

Key	Definition
<code>application</code>	The name of an application or application instance to deploy. If an application type is specified, all instances of that application are deployed.
<code>nodes</code>	A list of node names to target.
<code>query</code>	A PuppetDB or PQL query to use to discover nodes. The target is built from <code>certname</code> values collected at the top level of the query.
<code>node_group</code>	A classifier node group ID. The ID must correspond to a node group that has defined rules. It is not sufficient for parent groups of the node group in question to define rules. The user must also have permissions to view the node group.

To deploy an application instance in the production environment:

```
{
  "environment" : "production",
  "scope" : {
    "application" : "Wordpress_app[demo]"
  }
}
```



```
}
}
```

To deploy a list of nodes:

```
{
  "environment" : "production",
  "scope" : {
    "nodes" : ["node1.example.com", "node2.example.com"]
  }
}
```

To deploy a list of nodes with the certname value matching a regex:

```
{
  "environment" : "production",
  "scope" : {
    "query" : ["from", "nodes", ["~", "certname", ".*"]]
  }
}
```

To deploy to the nodes defined by the "All Nodes" node group:

```
{
  "environment" : "production",
  "scope" : {
    "node_group" : "000000000-0000-4000-8000-0000000000000"
  }
}
```

Response format

If all node runs succeed, and the environment is successfully deployed, the server returns a 202 response.

The response is a JSON object containing a link to retrieve information about the status of the job and uses any one of these keys:

Key	Definition
id	An absolute URL that links to the newly created job.
name	The name of the newly created job.

For example:

```
{
  "job" : {
    "id" : "https://orchestrator.example.com:8143/orchestrator/v1/jobs/1234"
    "name" : "1234"
  }
}
```

Error responses

For this endpoint, the kind key of the error displays the conflict.

Key	Definition
puppetlabs.orchestrator/unknown-environment	If the environment does not exist, the server returns a 404 response.

Key	Definition
<code>puppetlabs.orchestrator/empty-environment</code>	If the environment requested contains no applications or no nodes, the server returns a 400 response.
<code>puppetlabs.orchestrator/empty-target</code>	If the application instance specified to deploy does not exist or is empty, the server returns a 400 response.
<code>puppetlabs.orchestrator/dependency-cycle</code>	If the application code contains a cycle, the server returns a 400 response.
<code>puppetlabs.orchestrator/puppetdb-error</code>	If the orchestrator is unable to make a query to PuppetDB, the server returns a 400 response.
<code>puppetlabs.orchestrator/query-error</code>	If a user does not have appropriate permissions to run a query, or if the query is invalid, the server returns a 400 response.

Related information

[Puppet orchestrator API: forming requests](#) on page 509

Instructions on interacting with this API.

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

Tasks and plans in PE

Use the orchestrator to set up jobs in the console or on the command line and run tasks and plans across systems in your infrastructure.

You can install pre-existing tasks and plans and run them from the console and from the command line.

Note: If you have set up compilers and you want to use tasks and/or plans, you must either set `master_uris` or you `server_list` on agents to point to your compilers. This setting is described in the section on configuring compilers for orchestrator scale.

A *task* is a single action that you execute on target machines. With tasks, you can troubleshoot and deploy changes to individual or multiple systems in your infrastructure.

A *plan* is a bundle of tasks that can be combined with other logic. They allow you to do complex task operations, like running multiple tasks with one command or running certain tasks based on the output of another task.

You can run tasks and plans from your tool of choice: the console, the orchestrator command line interface (CLI), or the orchestrator API `/command/task` and `/command/plan_jobs` endpoint.

You can launch a task or plan and check on the status or view the output later with the console or CLI.

Running a task or plan does not update your Puppet configuration. If you run a task or plan that changes the state of a resource that Puppet is managing, a subsequent Puppet run changes the state of that resource back to what is defined in your Puppet configuration. For example, if you use a task to update the version of a managed package, the version of that package is reset to whatever is specified in a manifest on the next Puppet run.

RBAC for tasks and plans

Tasks rely on role-based access control (RBAC) to determine who can run which tasks on your infrastructure. You can delegate access, even on a per-task basis, to ensure that teams or individuals can run task and plan jobs only on infrastructure they manage.

RBAC for plans and tasks do **not** intersect. This means that if a user does not have access to a specific task but they have access to run a plan containing that task, they are still able to run the plan. This allows you to implement more

customized security options by wrapping tasks within plans. Read more about RBAC considerations for writing plans [here](#).

Installing tasks and plans

Puppet Enterprise comes with some pre-installed tasks, but you must install all plans or any other tasks you want to use.

Note: Many of the modules that come pre-installed with the open-source Bolt tool will not be available by default in PE.

PE comes with the following tasks already installed:

- `package` - Inspect, install, upgrade, and manage packages.
- `service` - Start, stop, restart, and check the status of a service.
- `facter_task` - Inspect the value of system facts.
- `puppet_conf` - Inspect Puppet agent configuration settings.

PE does not come with any pre-installed plans. You must download or write them.

Tasks and plans are packaged in Puppet modules.

Tasks can be installed from the [Forge](#) and managed with a Puppetfile and Code Manager.

Plans must be installed using Code Manager. See the section on code deployment for plans in the [Plan caveats and limitations](#) on page 464 doc.

If you plan to run tasks or plans from the console, they must be installed into the `production` environment.

Additional modules that include helpful content include:

- `puppet_agent`: Install Puppet Agent package.
- `reboot`: Manage system reboots.

Related information

[Managing environment content with a Puppetfile](#) on page 555

A Puppetfile specifies detailed information about each environment's Puppet code and data, including where to get that code and data from, where to install it, and whether to update it.

Running tasks in PE

Use the orchestrator to set up jobs in the console or on the command line and run Bolt tasks across systems in your infrastructure.

Note: If you have set up compilers and you want to use tasks, you must either set `master_uris` or you `server_list` on agents to point to your compilers. This setting is described in the section on configuring compilers for orchestrator scale.

Related information

[Configure compilers](#) on page 168

Compilers must be configured to appropriately route communication between your master and agent nodes.

Running tasks from the console

Run ad-hoc tasks on target machines to upgrade packages, restart services, or perform any other type of single-action executions on your nodes.

When you set up a job to run a task from the console, the orchestrator creates a job ID to track the job, shows you all nodes included in the job, and runs the tasks on those nodes in the appropriate order. Puppet compiles a new catalog for each node included in the job.

There are three ways to specify the job target (the nodes you want to run tasks on):

- A static node list
- A Puppet Query Language (PQL) query

- A node group

You can't combine these methods, and if you switch from one to the other, the target list clears and starts over. In other words, if you create a target list by using a node list, switching to a PQL query clears the node list. You can do a one-time conversion of a PQL query to a static node list if you want to add or remove nodes from the query results.

Important: Running a task does not update your Puppet configuration. If you run a task that changes the state of a resource that Puppet is managing (such as upgrading a service or package), a subsequent Puppet run changes the state of that resource back to what is defined in your Puppet configuration. For example, if you use a task to update the version of a managed package, the version of that package is reset to whatever is specified in the relevant manifest on the next Puppet run.

Run a task on a node list

Create a list of target nodes when you need to run a task on a specific set of nodes that isn't easily defined by a PQL query.

Before you begin

Install the tasks you want to use.

Make sure you have permission to run the tasks.

Make sure you have access to the nodes you want to target.

Tip: You can add network devices to a node list when you have installed modules for device transports in your production environment. You can find such modules in [Puppet Forge](#).

1. In the console, in the **Run** section, click **Task**.
2. In the **Task** field, select a task to run, for example `service`.

Note: If the tasks you expect are not available, you either have no tasks installed, or you don't have the correct permissions to run them.

3. Optional: In the **Job description** field, provide a description. What you enter here appears on the job list and job details pages.
4. Set any parameters and values for the task. Click **Add parameter** for each parameter-value pair you add.

If you chose the `service` task, then you have two required parameters. For **action**, you can choose `restart`. For **service**, enter `nginx`.

Note: Parameters marked with an asterisk (*) are required. Optional parameters are available in a dropdown. If there are no required or optional parameters, you can add arbitrary parameters provided those parameters are allowed in the task.

Express values as strings, arrays, objects, integers, or booleans (true or false). You must express empty strings as two double quotes with no space (" "). Wrap boolean values in double quotes (for example, "false").

5. Optional: Under **Schedule**, select **Later** and choose a start date, time, and time zone for the job to run.

Tip: Based on the configuration of your console, the default time zone is either UTC, Coordinated Universal Time, or the local time, UTC offset, of your browser.

6. From the list of target types, select **Node list**.
7. Create the node list.

- a) Expand the **Inventory nodes** target.
- b) Enter the name of the node you want to find and click **Search**.

Note: The search does not handle regular expressions.

- c) From the list of results, select the nodes that you want to add to your list. They are added to a table below.
- d) Repeat the search to add other nodes. You can select nodes from multiple searches to create the node list.

Tip: To remove a node from the table, select the checkbox next to it and click **Remove selected**.

8. After you have selected all your target nodes, click **Run job** or **Schedule job**.

Your job appears on the **Jobs** page.

Related information

[Review jobs from the console](#) on page 505

View a list of recent jobs and drill down to see useful details about each one.

Run a task over SSH

Use the SSH protocol to run tasks on target nodes that do not have the Puppet agent installed.

Before you begin

Install the tasks you want to use.

Make sure you have permission to run the tasks on all nodes.

1. In the console, in the **Run** section, click **Task**.
2. In the **Task** field, select a task to run, for example `service`.

Note: If the tasks you expect are not available, you either have no tasks installed, or you don't have the correct permissions to run them.

3. Optional: In the **Job description** field, provide a description. What you enter here appears on the job list and job details pages.
4. Set any parameters and values for the task. Click **Add parameter** for each parameter-value pair you add.

If you chose the `service` task, then you have two required parameters. For **action**, you can choose `restart`. For **service**, enter `nginx`.

Note: Parameters marked with an asterisk (*) are required. Optional parameters are available in a dropdown. If there are no required or optional parameters, you can add arbitrary parameters provided those parameters are allowed in the task.

Express values as strings, arrays, objects, integers, or booleans (true or false). You must express empty strings as two double quotes with no space (" "). Wrap boolean values in double quotes (for example, "false").

5. Optional: Under **Schedule**, select **Later** and choose a start date, time, and time zone for the job to run.

Tip: Based on the configuration of your console, the default time zone is either UTC, Coordinated Universal Time, or the local time, UTC offset, of your browser.

6. From the list of target types, select **Node list**.
7. Create the node list.

- a) Expand the **SSH nodes** target.

Note: This target is available only for tasks permitted to run on all nodes.

- b) Enter the target host names and the credentials required to access them. If you use an SSH key, include begin and end tags.
- c) Optional: Select additional target options.
For example, to add a target port number, select **Target port** from the drop-down list, enter the number, and click **Add**.
- d) Click **Add nodes**. They are added to a table below.
- e) Repeat these steps to add other nodes. You can add SSH nodes with different credentials to create the node list.

Tip: To remove a node from the table, select the checkbox next to it and click **Remove selected**.

8. After you have selected all your target nodes, click **Run job** or **Schedule job**.

Your job appears on the **Jobs** page.

Related information

[Review jobs from the console](#) on page 505

View a list of recent jobs and drill down to see useful details about each one.

Run a task over WinRM

Use the Windows Remote Management (WinRM) to run tasks on target nodes that do not have the Puppet agent installed.

Before you begin

Install the tasks you want to use.

Make sure you have permission to run the tasks on all nodes.

1. In the console, in the **Run** section, click **Task**.
2. In the **Task** field, select a task to run, for example `service`.

Note: If the tasks you expect are not available, you either have no tasks installed, or you don't have the correct permissions to run them.

3. Optional: In the **Job description** field, provide a description. What you enter here appears on the job list and job details pages.
4. Set any parameters and values for the task. Click **Add parameter** for each parameter-value pair you add.

If you chose the `service` task, then you have two required parameters. For **action**, you can choose `restart`. For **service**, enter `nginx`.

Note: Parameters marked with an asterisk (*) are required. Optional parameters are available in a dropdown. If there are no required or optional parameters, you can add arbitrary parameters provided those parameters are allowed in the task.

Express values as strings, arrays, objects, integers, or booleans (true or false). You must express empty strings as two double quotes with no space (" "). Wrap boolean values in double quotes (for example, `"false"`).

5. Optional: Under **Schedule**, select **Later** and choose a start date, time, and time zone for the job to run.

Tip: Based on the configuration of your console, the default time zone is either UTC, Coordinated Universal Time, or the local time, UTC offset, of your browser.

6. From the list of target types, select **Node list**.
7. Create the node list.

- a) Expand the **WinRM nodes** target.

Note: This target is available only for tasks permitted to run on all nodes.

- b) Enter the target host names and the credentials required to access them.

- c) Optional: Select additional target options.

For example, to add a target port number, select **Target Port** from the drop-down list, enter the number, and click **Add**.

- d) Click **Add nodes**. They are added to a table below.

- e) Repeat these steps to add other nodes. You can add nodes with different credentials to create the node list.

Tip: To remove a node from the table, select the checkbox next to it and click **Remove selected**.

8. After you have selected all your target nodes, click **Run job** or **Schedule job**.

Your job appears on the **Jobs** page.

Run a task on a PQL query

Create a PQL query to run tasks on nodes that meet specific conditions.

Before you begin

Install the tasks you want to use.

Make sure you have access to the nodes you want to target.

Make sure you have permissions to run tasks and queries.

1. In the console, in the **Run** section, click **Task**.
2. In the **Task** field, select a task to run, for example `service`.

Note: If the tasks you expect are not available, you either have no tasks installed, or you don't have the correct permissions to run them.

3. Optional: In the **Job description** field, provide a description. What you enter here appears on the job list and job details pages.
4. Set any parameters and values for the task. Click **Add parameter** for each parameter-value pair you add.

If you chose the `service` task, then you have two required parameters. For **action**, you can choose `restart`. For **service**, enter `nginx`.

Note: Parameters marked with an asterisk (*) are required. Optional parameters are available in a dropdown. If there are no required or optional parameters, you can add arbitrary parameters provided those parameters are allowed in the task.

Express values as strings, arrays, objects, integers, or booleans (true or false). You must express empty strings as two double quotes with no space (" "). Wrap boolean values in double quotes (for example, "false").

5. Optional: Under **Schedule**, select **Later** and choose a start date, time, and time zone for the job to run.

Tip: Based on the configuration of your console, the default time zone is either UTC, Coordinated Universal Time, or the local time, UTC offset, of your browser.

6. From the list of target types, select **PQL query**.
7. Specify a target by doing one of the following:
 - Enter a query that selects the target you want. See the [Puppet Query Language \(PQL\) reference](#) for more information.
 - Click **Common queries**. Select one of the queries and replace the defaults in the braces ({ }) with values that specify the target you want.

Target	PQL query
All nodes	<code>nodes[certname] { }</code>
Nodes with a specific resource (example: <code>httpd</code>)	<code>resources[certname] { type = "Service" and title = "httpd" }</code>
Nodes with a specific fact and value (example: OS name is)	<code>inventory[certname] { facts.os.name = "<OS>" }</code>
Nodes with a specific report status (example: last run failed)	<code>reports[certname] { latest_report_status = "failed" }</code>
Nodes with a specific class (example:)	<code>resources[certname] { type = "Class" and title = "Apache" }</code>
Nodes assigned to a specific environment (example: <code>production</code>)	<code>nodes[certname] { catalog_environment = "production" }</code>
Nodes with a specific version of a resource type (example: <code>OpenSSL</code> is <code>v1.1.0e</code>)	<code>resources[certname] { type = "Package" and title="openssl" and parameters.ensure = "1.0.1e-51.el7_2.7" }</code>
Nodes with a specific resource and operating system (example: <code>httpd</code> and)	<code>inventory[certname] { facts.operatingsystem = "CentOS" and resources { type = "Service" and title = "httpd" } }</code>

8. Click **Submit query** and click **Refresh** to update the node results.
9. If you change or edit the query after it runs, click **Submit query** again.
10. Optional: To convert the query target results to a node list, for use as a node list target, click **Convert query to static node list**.

Note: If you select this option, the job target becomes a node list. You can add or remove nodes from the node list before running the job, but you cannot edit the query.

11. After you have selected all your target nodes, click **Run job** or **Schedule job**.

Your job appears on the **Jobs** page.

Important: When you run this job, the PQL query runs again, and the job might run on a different set of nodes than what is currently displayed. If you want the job to run only on the list as displayed, convert the query to a static node list before you run the job.

Related information

[Review jobs from the console](#) on page 505

View a list of recent jobs and drill down to see useful details about each one.

Run a task on a node group

Similar to running a task on a list of nodes that you create in the console, you can run a task on a node group.

Before you begin

Install the tasks you want to use.

Make sure you have permission to run the tasks.

Make sure you have access to the nodes you want to target.

Note: If you do not have permissions to view a node group, or the node group doesn't have any matching nodes, that node group won't be listed as an option for viewing. In addition, a node group does not appear if no rules have been specified for it.

1. In the console, in the **Run** section, click **Task**.
2. In the **Task** field, select a task to run, for example `service`.

Note: If the tasks you expect are not available, you either have no tasks installed, or you don't have the correct permissions to run them.

3. Optional: In the **Job description** field, provide a description. What you enter here appears on the job list and job details pages.
4. Set any parameters and values for the task. Click **Add parameter** for each parameter-value pair you add.

If you chose the `service` task, then you have two required parameters. For **action**, you can choose `restart`. For **service**, enter `nginx`.

Note: Parameters marked with an asterisk (*) are required. Optional parameters are available in a dropdown. If there are no required or optional parameters, you can add arbitrary parameters provided those parameters are allowed in the task.

Express values as strings, arrays, objects, integers, or booleans (true or false). You must express empty strings as two double quotes with no space (" "). Wrap boolean values in double quotes (for example, "false").

5. Optional: Under **Schedule**, select **Later** and choose a start date, time, and time zone for the job to run.

Tip: Based on the configuration of your console, the default time zone is either UTC, Coordinated Universal Time, or the local time, UTC offset, of your browser.

6. From the list of target types, select **Node group**.
7. In the **Choose a node group** box, type or select a node group, and click **Select**.
8. After you have selected all your target nodes, click **Run job** or **Schedule job**.

Your job appears on the **Jobs** page.

Schedule a task

Schedule a job to run a task at a particular date and time or on a regular schedule.

Before you begin

Install the tasks you want to use.

Make sure you have access to the nodes you want to target.

1. In the console, in the **Run** section, click **Task**.
2. Optional: In the **Job description** field, provide a description. What you enter here appears on the job list and job details pages.
3. In the **Task** field, select a task to run, for example `service`.

Note: If the tasks you expect are not available, you either have no tasks installed, or you don't have the correct permissions to run them.

4. Set any parameters and values for the task. Click **Add parameter** for each parameter-value pair you add.

If you chose the `service` task, then you have two required parameters. For **action**, you can choose `restart`. For **service**, enter `nginx`.

Note: Parameters marked with an asterisk (*) are required. Optional parameters are available in a dropdown. If there are no required or optional parameters, you can add arbitrary parameters provided those parameters are allowed in the task.

Express values as strings, arrays, objects, integers, or booleans (true or false). You must express empty strings as two double quotes with no space (" "). Wrap boolean values in double quotes (for example, "false").

5. Under **Schedule**, select **Later** and choose a start date, time, and time zone for the job to run.

Tip: Based on the configuration of your console, the default time zone is either UTC, Coordinated Universal Time, or the local time, UTC offset, of your browser.

6. Optional: To repeat the job on a regular schedule, change the run frequency from **Once** to **Hourly**, **Daily**, or **Weekly**.

Note: If a recurring job runs longer than the selected frequency, the following job waits to start until the next available time interval.

7. From the list of target types, select the category of nodes you want to target.

- **Node list** Add individual nodes by name.
- **PQL Query** Use the query language to retrieve a list of nodes.
- **Node group** Select an existing node group.

8. Click **Schedule job**.

Your task appears on the **Scheduled task** tab of the **Jobs** page.

Related information

[Console and console-services parameters](#) on page 142

Use these parameters to customize the behavior of the console and console-services. Parameters that begin with `puppet_enterprise::profile` can be modified from the console itself. See the configuration methods documents for more information on how to change parameters in the console or Hiera.

[Delete a scheduled job](#) on page 439

Delete a job that is scheduled to run at a later time.

[Review jobs from the console](#) on page 505

View a list of recent jobs and drill down to see useful details about each one.

Stop a job in progress

You can stop a job if, for example, you realize you need to reconfigure a class or push a configuration change that the job needs.

When you stop an on-demand Puppet run, runs that are underway finish, and runs that have not started are canceled.

Stopping tasks depends on how many nodes are targeted in the job versus the concurrency limit you've set. If the concurrency limit is higher than the number of nodes targeted, all nodes complete the task, as those nodes have already started the task.

To stop a job:

- In the console, go to the **Job details** page and select the **Puppet run** or **Task** tab. From the list of jobs, find the one you want and click **Stop**.
- On the command line, press **CTRL + C**.

Delete a scheduled job

Delete a job that is scheduled to run at a later time.

Tip: You must have the appropriate role-based permissions to delete another user's scheduled job.

1. In the console, open the **Jobs** page.
2. Open the tab for the type of job you want to delete.
 - **Scheduled Puppet Run**
 - **Scheduled task**
3. From the list of jobs, find the one you want to delete and click **Remove**.

Running tasks from the command line

Use the `puppet task run` command to run tasks on agent nodes.

Before you begin

The first time you run a command, you need to authenticate. See the [orchestrator installation instructions](#) for information about setting RBAC permissions and token authorization.

If you're running this command from a managed or non-managed workstation, you must specify the full path to the command. For example, `c:\Program Files\Puppet Labs\Client\bin\puppet-task run`.

Install the tasks you want to use.

Make sure you have permission to run the tasks on all nodes.

Make sure you have permission to run the tasks.

Make sure you have access to the nodes you want to target.

Tip: You can add network devices to a node list when you have installed modules for device transports in your production environment. You can find such modules in [Puppet Forge](#).

Make sure you have permissions to run tasks and queries.

Make sure you have permissions to run jobs and queries.

Use the `puppet task` tool and the relevant module to make changes arbitrarily, rather than through a Puppet configuration change. For example, to inspect a package or quickly stop a particular service.

You can run tasks on a single node, on nodes identified in a static list, on nodes retrieved by a PQL query, or on nodes in a node group.

Use the orchestrator command `puppet task` to trigger task runs.

Run a task on a PQL query

Create a PQL query to run tasks on nodes that meet specific conditions.

Before you begin

Install the tasks you want to use.

Make sure you have access to the nodes you want to target.

Make sure you have permissions to run tasks and queries.

Log into your master or client tools workstation and run one of the following commands:

- To specify the query on the command line: `puppet task run <TASK NAME> <PARAMETER>=<VALUE>, <PARAMETER>=<VALUE> --query '<QUERY>' <OPTIONS>`
- To pass the query in a text file: `puppet task run <TASK NAME> <PARAMETER>=<VALUE>, <PARAMETER>=<VALUE> --query @/path/to/file.txt`

For example, to run the service task with two required parameters, on nodes with "web" in their certname:

```
puppet task run service action=status service=nginx --query 'nodes
{ certname ~ "web" }'
```

Tip: Use `puppet task show <TASK NAME>` to see a list of available parameters for a task. Not all tasks require parameters.

Refer to the `puppet task` command options to see how to pass parameters with the `--params` flag.

The following table shows some example targets and the associated queries you could run with the orchestrator.

Be sure to wrap the entire query in single quotes and use double quotes inside the query.

Target	PQL query
Single node by certname	<code>--query 'nodes { certname = "mynode" }'</code>
All nodes with "web" in certname	<code>--query 'nodes { certname ~ "web" }'</code>
All nodes	<code>--query 'inventory { facts.os.name = "CentOS" }'</code>
All nodes with httpd managed	<code>--query 'inventory { facts.operatingsystem = "CentOS" and resources { type = "Service" and title = "httpd" } }'</code>
All nodes with failed reports	<code>--query 'reports { latest_report? = true and status = "failed" }'</code>
All nodes matching the environment for the last received catalog	<code>--query 'nodes { catalog_environment = "production" }'</code>

Tip: You can use `-q` in place of `--query`.

Run a task on a list of nodes or a single node

Use a node list target when you need to run a job on a set of nodes that doesn't easily resolve to a PQL query. Use a single node or a comma-separated list of nodes.

Before you begin

Install the tasks you want to use.

Make sure you have access to the nodes you want to target.

Make sure you have permissions to run tasks and queries.

Log into your master or client tools workstation and run one of the following commands:

- To run a task job on a single node: `puppet task run <TASK NAME> <PARAMETER>=<VALUE> , <PARAMETER>=<VALUE> --nodes <NODE NAME> <OPTIONS>`
- To run a task job on a list of nodes, use a comma-separated list of node names: `puppet task run <TASK NAME> <PARAMETER>=<VALUE> , <PARAMETER>=<VALUE> --nodes <NODE NAME> , <NODE NAME> , <NODE NAME> <OPTIONS>`

Note: Do not add spaces in the list of nodes.

- To run a task job on a node list from a text file: `puppet task run <TASK NAME> <PARAMETER>=<VALUE> , <PARAMETER>=<VALUE> --nodes @/path/to/file.txt`

Note: In the text file, put each node on a separate line.

For example, to run the service task with two required parameters, on three specific hosts:

```
puppet task run service action=status service=nginx --nodes
host1,host2,host3
```

Tip: Use `puppet task show <TASK NAME>` to see a list of available parameters for a task. Not all tasks require parameters.

Refer to the `puppet task` command options to see how to pass parameters with the `--params` flag.

Run a task on a node group

Similar to running a task on a list of nodes, you can run a task on a node group.

Before you begin

Install the tasks you want to use.

Make sure you have access to the nodes you want to target.

1. Log into your master or client tools workstation.
2. Run the command: `puppet task run <TASK NAME> --node-group <node-group-id>`

Tip: Use the `/v1/groups` endpoint to retrieve a list node groups and their IDs.

Related information

[GET /v1/groups](#) on page 361

Use the `/v1/groups` endpoint to retrieve a list of all node groups in the node classifier.

puppet task run command options

The following are common options you can use with the `task` action. For a complete list of global options run `puppet task --help`.

Option	Value	Description
<code>--noop</code>	Flag, default false	Run a task to simulate changes without actually enforcing the changes.

Option	Value	Description
<code>--params</code>	String	Specify a JSON object that includes the parameters, or specify the path to a JSON file containing the parameters, prefaced with <code>@</code> , for example, <code>@/path/to/file.json</code> . Do not use this flag if specifying parameter-value pairs inline; see more information below.
<code>--environment, -e</code>	Environment name	Use tasks installed in the specified environment.
<code>--description</code>	Flag, defaults to empty	Provide a description for the job, to be shown on the job list and job details pages, and returned with the <code>puppet job show</code> command.

You can pass parameters into the task one of two ways:

- Inline, using the `<PARAMETER>=<VALUE>` syntax:

```
puppet task run <TASK NAME> <PARAMETER>=<VALUE>, <PARAMETER>=<VALUE> --
nodes <LIST OF NODES>
puppet task run my_task action=status service=my_service timeout=8 --nodes
host1,host2,host3
```

- With the `--params` option, as a JSON object or reference to a JSON file:

```
puppet task run <TASK NAME> --params '<JSON OBJECT>' --nodes <LIST OF
NODES>
puppet task run my_task --params '{ "action":"status",
  "service":"my_service", "timeout":8 }' --nodes host1,host2,host3
puppet task run my_task --params @/path/to/file.json --nodes
host1,host2,host3
```

You can't combine these two ways of passing in parameters; choose either inline or `--params`. If you use the inline way, parameter types other than string, integer, double, and Boolean will be interpreted as strings. Use the `--params` method if you want them read as their original type.

Reviewing task job output

The output the orchestrator returns depends on the type of task you run. Output is either standard output (STDOUT) or structured output. At minimum, the orchestrator prints a new job ID and the number of nodes in the task.

The following example shows a task to check the status of the Puppet service running on a list of nodes derived from a PQL query.

```
[example@orch-master ~]$ puppet task run service service=puppet
  action=status -q 'nodes {certname ~ "br"}' --environment=production
Starting job ...
New job ID: 2029
Nodes: 8

Started on bronze-11 ...
Started on bronze-8 ...
Started on bronze-3 ...
Started on bronze-6 ...
Started on bronze-2 ...
Started on bronze-5 ...
Started on bronze-7 ...
Started on bronze-10 ...
```

```

Finished on node bronze-11
  status : running
  enabled : true
Finished on node bronze-3
  status : running
  enabled : true
Finished on node bronze-8
  status : running
  enabled : true
Finished on node bronze-7
  status : running
  enabled : true
Finished on node bronze-2
  status : running
  enabled : true
Finished on node bronze-6
  status : running
  enabled : true
Finished on node bronze-5
  status : running
  enabled : true
Finished on node bronze-10
  status : running
  enabled : true

Job completed. 8/8 nodes succeeded.
Duration: 1 sec

```

Tip: To view the status of all running, completed, and failed jobs run the `puppet job show` command, or view them from the **Job details** page in the console.

Related information

[Review jobs from the console](#) on page 505

View a list of recent jobs and drill down to see useful details about each one.

[Review jobs from the command line](#) on page 508

Use the `puppet job show` command to view running or completed jobs.

Inspect tasks

View the tasks that you have installed and have permission to run, as well as the documentation for those tasks.

Before you begin

The first time you run a command, you need to authenticate. See the [orchestrator installation instructions](#) for information about setting RBAC permissions and token authorization.

If you're running this command from a managed or non-managed workstation, you must specify the full path to the command. For example, `c:\Program Files\Puppet Labs\Client\bin\puppet-task run`.

Install the tasks you want to use.

Make sure you have permission to run the tasks on all nodes.

Make sure you have permission to run the tasks.

Make sure you have access to the nodes you want to target.

Tip: You can add network devices to a node list when you have installed modules for device transports in your production environment. You can find such modules in [Puppet Forge](#).

Make sure you have permissions to run tasks and queries.

Make sure you have permissions to run jobs and queries.

Install the tasks you want to use.

Make sure you have permissions to run tasks and queries.

Log into your master or client tools workstation and run one of the following commands:

- To check the documentation for a specific task: `puppet task show <TASK>`. The command returns the following:
 - The command format for running the task
 - Any parameters available to use with the task
- To view a list of your permitted tasks: `puppet task show`
- To view a list of all installed tasks pass the `--all` flag: `puppet task show --all`

Stop a job in progress

You can stop a job if, for example, you realize you need to reconfigure a class or push a configuration change that the job needs.

When you stop an on-demand Puppet run, runs that are underway finish, and runs that have not started are canceled.

Stopping tasks depends on how many nodes are targeted in the job versus the concurrency limit you've set. If the concurrency limit is higher than the number of nodes targeted, all nodes complete the task, as those nodes have already started the task.

To stop a job:

- In the console, go to the **Job details** page and select the **Puppet run** or **Task** tab. From the list of jobs, find the one you want and click **Stop**.
- On the command line, press **CTRL + C**.

Running plans in PE

Use the orchestrator to set up jobs in the console or on the command line and run plans across systems in your infrastructure.

Running plans from the console

Run ad hoc plans from the console.

Before you begin

Install the plans you want to use.

Make sure you have permission to run the plans.

When you set up a job to run a plan from the console, the orchestrator creates a job ID to track the job, shows the nodes included in the job, and runs the plan on those nodes in the appropriate order. Puppet compiles a new catalog for each node included in the job.

Note: Running a plan does not update your Puppet configuration. If you run a plan that changes the state of a resource that is managing (such as upgrading a service or package), a subsequent run changes the state of that resource back to what is defined in your configuration.

1. In the console, in the **Run** section, click **Plan**.
2. Optional: Under **Job description**, name the job that is running the plan something unique so you can find it later in the job details page.
3. Select any parameters you want to include. For example, **nodes**.

Note: To find out what parameters can be included in a plan, view the plan using the command line. For more information, see [Inspecting plans](#) on page 467

4. After you have added parameters, click **Run job**.

Your plan status and output will appear in the **Job** section on the **Plan** tab.

Running plans from the command line

Run a plan using the `puppet plan run` command.

On the command line, run the command `puppet plan run` with the following information included:

- The full name of the plan, formatted as `<MODULE>::<PLAN>`.
- Any plan parameters.
- Credentials, if required, formatted with the `--user` and `--password` flags.

For example, if a plan defined in `mymodule/plans/myplan.pp` accepts a `load_balancer` parameter, run:

```
puppet plan run mymodule::myplan load_balancer=lb.myorg.com
```

You can pass a comma-separated list of node names, wildcard patterns, or group names to a plan parameter that is passed to a run function or that the plan resolves using `get_targets`.

Plan command options

The following are common options you can use with the `plan` action. For a complete list of global options run `puppet plan --help`.

- `--params` - A string value used to specify a JSON object that includes the parameters, or specify the path to a JSON file containing the parameters, prefaced with `@`. For example, `@/path/to/file.json`. Do not use this flag if specifying parameter-value pairs inline.
- `--environment` or `-e` - The name of the environment where plans are installed.
- `--description` - A flag used to provide a description for the job to be shown on the job list and job details pages and returned with the `puppet job show` command. It defaults to empty.

You can pass parameters into the plan one of two ways:

- Inline, using the `<PARAMETER>=<VALUE>` syntax
- With the `--params` option, as a JSON object or reference to a JSON file.

For example, review this plan:

```
plan example::test_params(Targetspec $nodes, String $command){
  run_command($command, $nodes)
}
```

You can pass parameters using either option below:

- ```
puppet plan run example::test_params nodes=my-node.company.com
 command=whoami
```
- ```
puppet plan run example::test_params --params '{ "nodes": "my-
node.company.com", "command": "whoami" }'
```

You can't combine these two ways of passing in parameters. Choose either inline or `--params`.

If you use the inline way, parameter types other than string, integer, double, and Boolean will be interpreted as strings. Use the `--params` method if you want them read as their original type.

Plan caveats and limitations

Some plan language and features are not available in PE or have some caveats.

Bolt modules are not included

Many modules that come pre-installed with the open-source Bolt tool will not be available by default in PE.

Plan language functions

Some plan functions will not work in PE and using them will cause a plan to fail:

- `apply_prep`
- `file::read`
- `file::write`
- `file::readable`
- `file::exists`
- `set_feature`
- `add_to_group`
- `set_config`

Apply blocks

The `apply` feature for managing resources in a plan is not yet supported in PE. Using it will cause a plan to fail at the `apply` step.

Target groups

Support for target groups is unavailable in PE. Using `add_to_group` will cause a plan to fail and referencing a group name in `get_targets` will not return any nodes. When using `get_targets` you **must** reference either node certnames or supply a PuppetDB query.

Here is an example of a plan using `get_targets` with node certnames:

```
plan example::get_targets_example () {
  $nodes = get_targets(['node1.mydomain.com', 'node2.mydomain.com'])
  run_command('whoami', $nodes)
}
```

Target features

Support for target features is unavailable in PE. Using `set_feature` will cause a plan to fail and referencing a target object's features list will always return an empty array.

Here is an example of a plan that would return an empty features list:

```
plan example::empty_features_example (TargetSpec $node) {
  return get_target($node).features()
}
```

Target configuration

While you can set up node transport configuration through the PE inventory for nodes to use SSH or WinRM, plans in PE do not support setting or changing the configuration settings for targets from within a plan. Using the `set_config` function in a plan will cause a plan to fail and referencing a target object's configuration hash will always return an empty hash.

The use of URIs in a target name to override the transport is also not supported. All references to targets (i.e. when using `get_targets`) must be either PuppetDB queries or valid certnames that are already in the PE inventory.

Here is an example of a plan that uses `get_targets` correctly:

```
plan example::get_targets_example () {
  ## NOTE! If you used ssh://node1.mydomain.com as the first entry, this
  ## plan would fail!
  $nodes = get_targets(['node1.mydomain.com', 'node2.mydomain.com'])
  run_command('whoami', $nodes)
}
```

`_run_as` options in plan functions

Plans in PE does not support the `_run_as` parameter for changing the user that accesses hosts or executes actions. If this parameter is supplied to any plan function, the plan will execute but the user will not change.

For example, the following plan will succeed, but will not run as `other_user`:

```
plan example::run_as_example (TargetSpec $nodes) {
  run_command('whoami', $nodes, _run_as => 'other_user')
}
```

The `localhost` target

The special target `localhost` is not available for plans in PE. Using `localhost` anywhere in a plan will result in a plan failure. If you need to run a plan on the PE master host, use the master's certname to reference it.

For example, you can use the following plan for PE master host `my-pe-master.company.com`:

```
plan example::referencing_the_master(){
  # Note that if you tried to use `localhost` instead of `my-pe-master` this
  # plan would fail!
  run_command('whoami', 'my-pe-master.company.com')
}
```

Standard Puppet log functions

Some of Puppet's built-in logging functions, like `warn`, will not log anywhere. Plans with those functions will still work, but the logging messages will not appear.

If you need to use a logging function in your plan, the `out::message()` plan function will write an event to the orchestrator's event stream.

For example, the following will log the message correctly to the event stream:

```
plan example::logging(){
  out::message("This will output to the orchestrator event stream")
}
```

This will not log any messages:

```
plan example::bad_logging(){
  # the following will output any messages anywhere
  warn("this will also go nowhere!")
}
```

Script and file sources

When using `run_script` or `download_file`, the source location for the files **must** be from a module that uses a `modulename/filename` selector for a file or directory in `$MODULEROOT/files`. PE does not support file sources that reference absolute paths.

Here is an example of a module structure and a plan that correctly uses the `modulename/filename` selector:

```
example/
### files
  ###my_script.sh
### plans
  ###run_script_example.pp

plan example::run_script_example (TargetSpec $nodes) {
```

```

    run_script('example/my_script.sh', $nodes)
  }

```

Code deployment for plans

For plans in PE to work, you must have code manager enabled to deploy code to your master.

Masters will now deploy a second codedir for plans to load code from. The second code location on masters have some effects on standard module functionality:

- Installation modules using the `puppet module install` command will not work for plans because the `puppet module tool` will not install to the secondary location for plans. The `puppet module install` command will still work for normal Puppet code executed and compiled from Puppet Server.
- A `$modulepath` configuration that uses fully qualified paths might not work for plans if they reference the standard `/etc/puppetlabs/code` location. We recommend using relative paths in `$modulepath`.

Inspecting plans

View the plans you have installed and which ones you have permissions to run, as well as details about individual plans.

To see information about plans, log into your master or client tools workstation and run one of the following commands:

- `puppet plan show` - View a list of your permitted plans.
- `puppet plan show --all` - View a list of all installed plans.

There is currently no way to view plan metadata using the `puppet plan show` function. See [Orchestration services known issues](#) on page 46 for more information.

Writing tasks

Bolt tasks are similar to scripts, but they are kept in modules and can have metadata. This allows you to reuse and share them.

You can write tasks in any programming language the target nodes run, such as Bash, PowerShell, or Python. A task can even be a compiled binary that runs on the target. Place your task in the `./tasks` directory of a module and add a metadata file to describe parameters and configure task behavior.

For a task to run on remote *nix systems, it must include a shebang (`#!`) line at the top of the file to specify the interpreter.

For example, the Puppet `mysql::sql` task is written in Ruby and provides the path to the Ruby interpreter. This example also accepts several parameters as JSON on `stdin` and returns an error.

```

#!/opt/puppetlabs/puppet/bin/ruby
require 'json'
require 'open3'
require 'puppet'

def get(sql, database, user, password)
  cmd = ['mysql', '-e', "#{sql} "]
  cmd << "--database=#{database}" unless database.nil?
  cmd << "--user=#{user}" unless user.nil?
  cmd << "--password=#{password}" unless password.nil?
  stdout, stderr, status = Open3.capture3(*cmd) # rubocop:disable Lint/
  UselessAssignment
  raise Puppet::Error, _("#{stderr: ' %{stderr}')} % { stderr: stderr }") if
    status != 0
  { status: stdout.strip }
end

```

```

params = JSON.parse(STDIN.read)
database = params['database']
user = params['user']
password = params['password']
sql = params['sql']

begin
  result = get(sql, database, user, password)
  puts result.to_json
  exit 0
rescue Puppet::Error => e
  puts({ status: 'failure', error: e.message }.to_json)
  exit 1
end

```

Related information

[Task compatibility](#) on page 23

This table shows which version of the Puppet task specification is compatible with each version of PE.

Secure coding practices for tasks

Use secure coding practices when you write tasks and help protect your system.

Note: The information in this topic covers basic coding practices for writing secure tasks. It is not an exhaustive list.

One of the methods attackers use to gain access to your systems is remote code execution, where by running an allowed script they gain access to other parts of the system and can make arbitrary changes. Because Bolt executes scripts across your infrastructure, it is important to be aware of certain vulnerabilities, and to code tasks in a way that guards against remote code execution.

Adding task metadata that validates input is one way to reduce vulnerability. When you require an enumerated (enum) or other non-string types, you prevent improper data from being entered. An arbitrary string parameter does not have this assurance.

For example, if your task has a parameter that selects from several operational modes that are passed to a shell command, instead of

```
String $mode = 'file'
```

use

```
Enum[file,directory,link,socket] $mode = file
```

If your task has a parameter that identifies a file on disk, ensure that a user can't specify a relative path that takes them into areas where they shouldn't be. Reject file names that have slashes.

Instead of

```
String $path
```

use

```
Pattern[/\A[^\/*\z/] $path
```

In addition to these task restrictions, different scripting languages each have their own ways to validate user input.

PowerShell

In PowerShell, code injection exploits calls that specifically evaluate code. Do not call `Invoke-Expression` or `Add-Type` with user input. These commands evaluate strings as C# code.

Reading sensitive files or overwriting critical files can be less obvious. If you plan to allow users to specify a file name or path, use `Resolve-Path` to verify that the path doesn't go outside the locations you expect the task to access. Use `Split-Path -Parent $path` to check that the resolved path has the desired path as a parent.

For more information, see [PowerShell Scripting](#) and [Powershell's Security Guiding Principles](#).

Bash

In Bash and other command shells, shell command injection takes advantage of poor shell implementations. Put quotations marks around arguments to prevent the vulnerable shells from evaluating them.

Because the `eval` command evaluates all arguments with string substitution, avoid using it with user input; however you can use `eval` with sufficient quoting to prevent substituted variables from being executed.

Instead of

```
eval "echo $input"
```

use

```
eval "echo '$input'"
```

These are operating system-specific tools to validate file paths: `realpath` or `readlink -f`.

Python

In Python malicious code can be introduced through commands like `eval`, `exec`, `os.system`, `os.popen`, and `subprocess.call` with `shell=True`. Use `subprocess.call` with `shell=False` when you include user input in a command or escape variables.

Instead of

```
os.system('echo '+input)
```

use

```
subprocess.check_output(['echo', input])
```

Resolve file paths with `os.realpath` and confirm them to be within another path by looping over `os.path.dirname` and comparing to the desired path.

For more information on the vulnerabilities of Python or how to escape variables, see Kevin London's blog post on [Dangerous Python Functions](#).

Ruby

In Ruby, command injection is introduced through commands like `eval`, `exec`, `system`, backtick (```) or `%x()` execution, or the `Open3` module. You can safely call these functions with user input by passing the input as additional arguments instead of a single string.

Instead of

```
system("echo #{flag1} #{flag2}")
```

use

```
system('echo', flag1, flag2)
```

Resolve file paths with `Pathname#realpath`, and confirm them to be within another path by looping over `Pathname#parent` and comparing to the desired path.

For more information on securely passing user input, see the blog post [Stop using backtick to run shell command in Ruby](#).

Naming tasks

Task names are named based on the filename of the task, the name of the module, and the path to the task within the module.

You can write tasks in any language that runs on the target nodes. Give task files the extension for the language they are written in (such as `.rb` for Ruby), and place them in the top level of your module's `./tasks` directory.

Task names are composed of one or two name segments, indicating:

- The name of the module where the task is located.
- The name of the task file, without the extension.

For example, the `puppetlabs-mysql` module has the `sql` task in `./mysql/tasks/sql.rb`, so the task name is `mysql::sql`. This name is how you refer to the task when you run tasks.

The task filename `init` is special: the task it defines is referenced using the module name only. For example, in the `puppetlabs-service` module, the task defined in `init.rb` is the `service` task.

Each task or plan name segment must begin with a lowercase letter and:

- Must start with a lowercase letter.
- Can include digits.
- Can include underscores.
- Namespace segments must match the following regular expression `\A[a-z][a-z0-9_]*\Z`
- The file extension must not use the reserved extensions `.md` or `.json`.

Single-platform tasks

A task can consist of a single executable with or without a corresponding metadata file. For instance, `./mysql/tasks/sql.rb` and `./mysql/tasks/sql.json`. In this case, no other `./mysql/tasks/sql.*` files can exist.

Cross-platform tasks

A task can have multiple implementations, with metadata that explains when to use each one. A primary use case for this is to support different implementations for different target platforms, referred to as *cross-platform tasks*.

A task can also have multiple implementations, with metadata that explains when to use each one. A primary use case for this is to support different implementations for different target platforms, referred to as *cross-platform tasks*. For instance, consider a module with the following files:

```
- tasks
- sql_linux.sh
- sql_linux.json
- sql_windows.ps1
- sql_windows.json
- sql.json
```

This task has two executables (`sql_linux.sh` and `sql_windows.ps1`), each with an implementation metadata file and a task metadata file. The executables have distinct names and are compatible with older task runners such as Puppet Enterprise 2018.1 and earlier. Each implementation has its own metadata which documents how to use the implementation directly or marks it as private to hide it from UI lists.

An implementation metadata example:

```
{
  "name": "SQL Linux",
  "description": "A task to perform sql operations on linux targets",
```

```
    "private": true
  }
}
```

The task metadata file contains an implementations section:

```
{
  "implementations": [
    { "name": "sql_linux.sh", "requirements": ["shell"] },
    { "name": "sql_windows.ps1", "requirements": ["powershell"] }
  ]
}
```

Each implementations has a name and a list of requirements. The requirements are the set of *features* which must be available on the target in order for that implementation to be used. In this case, the `sql_linux.sh` implementation requires the `shell` feature, and the `sql_windows.ps1` implementations requires the PowerShell feature.

The set of features available on the target is determined by the task runner. You can specify additional features for a target via `set_feature` or by adding features in the inventory. The task runner chooses the *first* implementation whose requirements are satisfied.

The following features are defined by default:

- `puppet-agent`: Present if the target has the Puppet agent package installed. This feature is automatically added to hosts with the name `localhost`.
- `shell`: Present if the target has a posix shell.
- `powershell`: Present if the target has PowerShell.

Sharing executables

Multiple task implementations can refer to the same executable file.

Executables can access the `_task` metaparameter, which contains the task name. For example, the following creates the tasks `service::stop` and `service::start`, which live in the executable but appear as two separate tasks.

`myservice/tasks/init.rb`

```
#!/usr/bin/env ruby
require 'json'

params = JSON.parse(STDIN.read)
action = params['action'] || params['_task']
if ['start', 'stop'].include?(action)
  `systemctl #{params['_task']} #{params['service']}`
end
```

`myservice/tasks/start.json`

```
{
  "description": "Start a service",
  "parameters": {
    "service": {
      "type": "String",
      "description": "The service to start"
    }
  },
  "implementations": [
    { "name": "init.rb" }
  ]
}
```

```

}

myservice/tasks/stop.json

{
  "description": "Stop a service",
  "parameters": {
    "service": {
      "type": "String",
      "description": "The service to stop"
    }
  },
  "implementations": [
    { "name": "init.rb" }
  ]
}

```

Sharing task code

Multiple tasks can share common files between them. Tasks can additionally pull library code from other modules.

To create a task that includes additional files pulled from modules, include the `files` property in your metadata as an array of paths. A path consists of:

- the module name
- one of the following directories within the module:
 - `files` — Most helper files. This prevents the file from being treated as a task or added to the Puppet Ruby loadpath.
 - `tasks` — Helper files that can be called as tasks on their own.
 - `lib` — Ruby code that might be reused by types, providers, or Puppet functions.
- the remaining path to a file or directory; directories must include a trailing slash /

All path separators must be forward slashes. An example would be `stdlib/lib/puppet/`.

The `files` property can be included both as a top-level metadata property, and as a property of an implementation, for example:

```

{
  "implementations": [
    { "name": "sql_linux.sh", "requirements": ["shell"], "files": ["mymodule/
files/lib.sh"] },
    { "name": "sql_windows.ps1", "requirements": ["powershell"], "files":
["mymodule/files/lib.ps1"] }
  ],
  "files": ["emoji/files/emojis/"]
}

```

When a task includes the `files` property, all files listed in the top-level property and in the specific implementation chosen for a target are copied to a temporary directory on that target. The directory structure of the specified files is preserved such that paths specified with the `files` metadata option are available to tasks prefixed with `_install_dir`. The task executable itself is located in its module location under the `_install_dir` as well, so other files can be found at `../../mymodule/files/` relative to the task executable's location.

For example, you can create a task and metadata in a module at `~/puppetlabs/bolt/site-modules/mymodule/tasks/task.{json,rb}`.

Metadata

```

{
  "files": ["multi_task/files/rb_helper.rb"]
}

```



```
}
```

File resource

```
multi_task/files/rb_helper.rb
```

```
def useful_ruby
  { helper: "ruby" }
end
```

Task

```
#!/usr/bin/env ruby
require 'json'

params = JSON.parse(STDIN.read)
require_relative File.join(params['_installdir'], 'multi_task', 'files',
  'rb_helper.rb')
# Alternatively use relative path
# require_relative File.join(__dir__, '..', '..', 'multi_task', 'files',
  'rb_helper.rb')
puts useful_ruby.to_json
```

Output

```
Started on localhost...
Finished on localhost:
{
  "helper": "ruby"
}
Successful on 1 node: localhost
Ran on 1 node in 0.12 seconds
```

Task helpers

To help with writing tasks, Bolt includes [python_task_helper](#) and [ruby_task_helper](#). It also makes a useful demonstration of including code from another module.

Python example

Create task and metadata in a module at `~/puppetlabs/bolt/site-modules/mymodule/tasks/task.{json,py}`.

Metadata

```
{
  "files": ["python_task_helper/files/task_helper.py"],
  "input_method": "stdin"
}
```

Task

```
#!/usr/bin/env python
import os, sys
sys.path.append(os.path.join(os.path.dirname(__file__), '..', '..',
  'python_task_helper', 'files'))
from task_helper import TaskHelper

class MyTask(TaskHelper):
  def task(self, args):
    return {'greeting': 'Hi, my name is '+args['name']}
```

```
if __name__ == '__main__':
    MyTask().run()
```

Output

```
$ bolt task run mymodule::task -n localhost name='Julia'
Started on localhost...
Finished on localhost:
{
  "greeting": "Hi, my name is Julia"
}
Successful on 1 node: localhost
Ran on 1 node in 0.12 seconds
```

Ruby example

Create task and metadata in a new module at `~/ .puppetlabs/bolt/site-modules/mymodule/tasks/mytask.{json,rb}`.

Metadata

```
{
  "files": ["ruby_task_helper/files/task_helper.rb"],
  "input_method": "stdin"
}
```

Task

```
#!/usr/bin/env ruby
require_relative '../.. /ruby_task_helper/files/task_helper.rb'

class MyTask < TaskHelper
  def task(name: nil, **kwargs)
    { greeting: "Hi, my name is #{name}" }
  end
end

MyTask.run if __FILE__ == $0
```

Output

```
$ bolt task run mymodule::mytask -n localhost name="Robert"); DROP TABLE
Students;--"
Started on localhost...
Finished on localhost:
{
  "greeting": "Hi, my name is Robert"); DROP TABLE Students;--"
}
Successful on 1 node: localhost
Ran on 1 node in 0.12 seconds
```

Writing remote tasks

Some targets are hard or impossible to execute tasks on directly. In these cases, you can write a task that runs on a proxy target and remotely interacts with the real target.

For example, a network device might have a limited shell environment or a cloud service might be driven only by HTTP APIs. By writing a remote task, Bolt allows you to specify connection information for remote targets in their inventory file and injects them into the `_target` metaparam.

This example shows how to write a task that posts messages to Slack and reads connection information from `inventory.yaml`:

```
#!/usr/bin/env ruby
# modules/slack/tasks/message.rb

require 'json'
require 'net/http'

params = JSON.parse(STDIN.read)
# the slack API token is passed in from inventory
token = params['_target']['token']

uri = URI('https://slack.com/api/chat.postMessage')
http = Net::HTTP.new(uri.host, uri.port)
http.use_ssl = true

req = Net::HTTP::Post.new(uri, 'Content-type' => 'application/json')
req['Authorization'] = "Bearer #{params['_target']['token']}"
req.body = { channel: params['channel'], text: params['message'] }.to_json

resp = http.request(req)

puts resp.body
```

To prevent accidentally running a normal task on a remote target and breaking its configuration, Bolt won't run a task on a remote target unless its metadata defines it as remote:

```
{
  "remote": true
}
```

Add Slack as a remote target in your inventory file:

```
---
nodes:
  - name: my_slack
    config:
      transport: remote
      remote:
        token: <SLACK_API_TOKEN>
```

Finally, make `my_slack` a target that can run the `slack::message`:

```
bolt task run slack::message --nodes my_slack message="hello" channel=<slack
channel id>
```

Defining parameters in tasks

Allow your task to accept parameters as either environment variables or as a JSON hash on standard input.

Tasks can receive input as either environment variables, a JSON hash on standard input, or as PowerShell arguments. By default, the task runner submits parameters as both environment variables and as JSON on `stdin`.

If your task needs to receive parameters only in a certain way, such as `stdin` only, you can set the input method in your task metadata. For Windows tasks, it's usually better to use tasks written in PowerShell. See the related topic about task metadata for information about setting the input method.

Environment variables are the easiest way to implement parameters, and they work well for simple JSON types such as strings and numbers. For arrays and hashes, use structured input instead because parameters with undefined values

(nil, undef) passed as environment variables have the String value null. For more information, see [Structured input and output](#) on page 477.

To add parameters to your task as environment variables, pass the argument prefixed with the Puppet task prefix PT_.

For example, to add a message parameter to your task, read it from the environment in task code as PT_message. When the user runs the task, they can specify the value for the parameter on the command line as message=hello , and the task runner submits the value hello to the PT_message variable.

```
#!/usr/bin/env bash
echo your message is $PT_message
```

Defining parameters in Windows

For Windows tasks, you can pass parameters as environment variables, but it's easier to write your task in PowerShell and use named arguments. By default tasks with a .ps1 extension use PowerShell standard argument handling.

For example, this PowerShell task takes a process name as an argument and returns information about the process. If no parameter is passed by the user, the task returns all of the processes.

```
[CmdletBinding()]
Param(
    [Parameter(Mandatory = $False)]
    [String]
    $Name
)

if ($Name -eq $null -or $Name -eq "") {
    Get-Process
} else {
    $processes = Get-Process -Name $Name
    $result = @(
        foreach ($process in $processes) {
            $result += @{
                "Name" = $process.ProcessName;
                "CPU" = $process.CPU;
                "Memory" = $process.WorkingSet;
                "Path" = $process.Path;
                "Id" = $process.Id
            }
        }
    )
    if ($result.Count -eq 1) {
        ConvertTo-Json -InputObject $result[0] -Compress
    } elseif ($result.Count -gt 1) {
        ConvertTo-Json -InputObject @{"_items" = $result} -Compress
    }
}
```

To pass parameters in your task as environment variables (PT_parameter), you must set input_method in your task metadata to environment. To run Ruby tasks on Windows, the Puppet agent must be installed on the target nodes.

Returning errors in tasks

To return a detailed error message if your task fails, include an Error object in the task's result.

When a task exits non-zero, the task runner checks for an error key (_error). If one is not present, the task runner generates a generic error and adds it to the result. If there is no text on stdout but text is present on stderr, the stderr text is included in the message.

```
{
  "_error": {
    "msg": "Task exited 1:\nSomething on stderr",
  }
}
```

```

    "kind": "puppetlabs.tasks/task-error",
    "details": { "exitcode": 1 }
  }

```

An error object includes the following keys:

msg

A human readable string that appears in the UI.

kind

A standard string for machines to handle. You can share kinds between your modules or namespace kinds per module.

details

An object of structured data about the tasks.

Tasks can provide more details about the failure by including their own error object in the result at `_error`.

```

#!/opt/puppetlabs/puppet/bin/ruby

require 'json'

begin
  params = JSON.parse(STDIN.read)
  result = {}
  result['result'] = params['dividend'] / params['divisor']

rescue ZeroDivisionError
  result[:_error] = { msg: "Cannot divide by zero",
                     # namespace the error to this module
                     kind: "puppetlabs-example_modules/dividebyzero",
                     details: { divisor: divisor },
                   }
rescue Exception => e
  result[:_error] = { msg: e.message,
                     kind: "puppetlabs-example_modules/unknown",
                     details: { class: e.class.to_s },
                   }
end

puts result.to_json

```

Structured input and output

If you have a task that has many options, returns a lot of information, or is part of a task plan, consider using structured input and output with your task.

The task API is based on JSON. Task parameters are encoded in JSON, and the task runner attempts to parse the output of the tasks as a JSON object.

The task runner can inject keys into that object, prefixed with `_`. If the task does not return a JSON object, the task runner creates one and places the output in an `_output` key.

Structured input

For complex input, such as hashes and arrays, you can accept structured JSON in your task.

By default, the task runner passes task parameters as both environment variables and as a single JSON object on stdin. The JSON input allows the task to accept complex data structures.

To accept parameters as JSON on stdin, set the `params` key to accept JSON on stdin.

```

#!/opt/puppetlabs/puppet/bin/ruby

```

```

require 'json'

params = JSON.parse(STDIN.read)

exitcode = 0
params['files'].each do |filename|
  begin
    FileUtils.touch(filename)
    puts "updated file #{filename}"
  rescue
    exitcode = 1
    puts "couldn't update file #{filename}"
  end
end
exit exitcode

```

If your task accepts input on `stdin` it should specify `"input_method": "stdin"` in its `metadata.json` file, or it might not work with `sudo` for some users.

Returning structured output

To return structured data from your task, print only a single JSON object to `stdout` in your task.

Structured output is useful if you want to use the output in another program, or if you want to use the result of the task in a Puppet task plan.

```

#!/usr/bin/env python
import json
import sys
minor = sys.version_info
result = { "major": sys.version_info.major, "minor":
  sys.version_info.minor }
json.dump(result, sys.stdout)

```

Converting scripts to tasks

To convert an existing script to a task, you can either write a task that wraps the script or you can add logic in your script to check for parameters in environment variables.

If the script is already installed on the target nodes, you can write a task that wraps the script. In the task, read the script arguments as task parameters and call the script, passing the parameters as the arguments.

If the script isn't installed or you want to make it into a cohesive task so that you can manage its version with code management tools, add code to your script to check for the environment variables, prefixed with `PT_`, and read them instead of arguments.



CAUTION: For any tasks that you intend to use with PE and assign RBAC permissions, make sure the script safely handles parameters or validate them to prevent shell injection vulnerabilities.

Given a script that accepts positional arguments on the command line:

```

version=$1
[ -z "$version" ] && echo "Must specify a version to deploy" && exit 1

if [ -z "$2" ]; then
  filename=$2
else
  filename=~ /myfile
fi

```

To convert the script into a task, replace this logic with task variables:

```
version=$PT_version #no need to validate if we use metadata
if [ -z "$PT_filename" ]; then
    filename=$PT_filename
else
    filename=~ /myfile
fi
```

Wrapping an existing script

If a script is not already installed on targets and you don't want to edit it, for example if it's a script someone else maintains, you can wrap the script in a small task without modifying it.



CAUTION: For any tasks that you intend to use with PE and assign RBAC permissions, make sure the script safely handles parameters or validate them to prevent shell injection vulnerabilities.

Given a script, `myscript.sh`, that accepts 2 positional args, `filename` and `version`:

1. Copy the script to the module's `files/` directory.
2. Create a metadata file for the task that includes the parameters and file dependency.

```
{ "input_method": "environment", "parameters": { "filename": { "type":
  "String[1]" }, "version": { "type": "String[1]" } }, "files":
  [ "script_example/files/myscript.sh" ] }
```

3. Create a small wrapper task that reads environment variables and calls the task.

```
#!/usr/bin/env bash set -e script_file="$PT__installdir/script_example/
files/myscript.sh" # If this task is going to be run from windows nodes
the wrapper must make sure it's executable chmod +x $script_file
commandline=( "$script_file" "$PT_filename" "$PT_version" ) # If the
stderr output of the script is important redirect it to stdout.
"${commandline[@]}" 2>&1
```

Supporting no-op in tasks

Tasks support no-operation functionality, also known as no-op mode. This function shows what changes the task would make, without actually making those changes.

No-op support allows a user to pass the `--noop` flag with a command to test whether the task will succeed on all targets before making changes.

To support no-op, your task must include code that looks for the `_noop` metaparameter. No-op is supported only in Puppet Enterprise.

If the user passes the `--noop` flag with their command, this parameter is set to `true`, and your task must not make changes. You must also set `supports_noop` to `true` in your task metadata or the task runner will refuse to run the task in noop mode.

No-op metadata example

```
{
  "description": "Write content to a file.",
  "supports_noop": true,
  "parameters": {
    "filename": {
      "description": "the file to write to",
      "type": "String[1]"
    },
    "content": {
      "description": "The content to write",
```

```

    "type": "String"
  }
}
}

```

No-op task example

```

#!/usr/bin/env python
import json
import os
import sys

params = json.load(sys.stdin)
filename = params['filename']
content = params['content']
noop = params.get('_noop', False)

exitcode = 0

def make_error(msg):
    error = {
        "_error": {
            "kind": "file_error",
            "msg": msg,
            "details": {},
        }
    }
    return error

try:
    if noop:
        path = os.path.abspath(os.path.join(filename, os.pardir))
        file_exists = os.access(filename, os.F_OK)
        file_writable = os.access(filename, os.W_OK)
        path_writable = os.access(path, os.W_OK)

        if path_writable == False:
            exitcode = 1
            result = make_error("Path %s is not writable" % path)
        elif file_exists == True and file_writable == False:
            exitcode = 1
            result = make_error("File %s is not writable" % filename)
        else:
            result = { "success": True , '_noop': True }
    else:
        with open(filename, 'w') as fh:
            fh.write(content)
            result = { "success": True }
except Exception as e:
    exitcode = 1
    result = make_error("Could not open file %s: %s" % (filename, str(e)))
print(json.dumps(result))
exit(exitcode)

```

Task metadata

Task metadata files describe task parameters, validate input, and control how the task runner executes the task.

Your task must have metadata to be published and shared on the Forge. Specify task metadata in a JSON file with the naming convention `<TASKNAME>.json`. Place this file in the module's `./tasks` folder along with your task file.

For example, the module `puppetlabs-mysql` includes the `mysql::sql` task with the metadata file, `sql.json`.

```
{
  "description": "Allows you to execute arbitrary SQL",
  "input_method": "stdin",
  "parameters": {
    "database": {
      "description": "Database to connect to",
      "type": "Optional[String[1]]"
    },
    "user": {
      "description": "The user",
      "type": "Optional[String[1]]"
    },
    "password": {
      "description": "The password",
      "type": "Optional[String[1]]",
      "sensitive": true
    },
    "sql": {
      "description": "The SQL you want to execute",
      "type": "String[1]"
    }
  }
}
```

Adding parameters to metadata

To document and validate task parameters, add the parameters to the task metadata as JSON object, `parameters`.

If a task includes `parameters` in its metadata, the task runner rejects any parameters input to the task that aren't defined in the metadata.

In the parameter object, give each parameter a description and specify its Puppet type. For a complete list of types, see the [types documentation](#).

For example, the following code in a metadata file describes a `provider` parameter:

```
"provider": {
  "description": "The provider to use to manage or inspect the service,
defaults to the system service manager",
  "type": "Optional[String[1]]"
}
```

Define sensitive parameters

You can define task parameters as sensitive, for example, passwords and API keys. These values are masked when they appear in logs and API responses. When you want to view these values, set the log file to `level: debug`.

To define a parameter as sensitive within the JSON metadata, add the `"sensitive": true` property.

```
{
  "description": "This task has a sensitive property denoted by its
metadata",
  "input_method": "stdin",
  "parameters": {
    "user": {
      "description": "The user",
      "type": "String[1]"
    },
    "password": {
      "description": "The password",

```

```

    "type": "String[1]",
    "sensitive": true
  }
}

```

Task metadata reference

The following table shows task metadata keys, values, and default values.

Table 2: Task metadata

Metadata key	Description	Value	Default
"description"	A description of what the task does.	String	None
"input_method"	What input method the task runner should use to pass parameters to the task.	<ul style="list-style-type: none"> environment stdin powershell 	Both environment and stdin unless .ps1 tasks, in which case powershell
"parameters"	The parameters or input the task accepts listed with a puppet type string and optional description. See adding parameters to metadata for usage information.	Array of objects describing each parameter	None
"puppet_task_version"	The version of the spec used.	Integer	1 (This is the only valid value.)
"supports_noop"	Whether the task supports no-op mode. Required for the task to accept the <code>--noop</code> option on the command line.	Boolean	False
"implementations"	A list of task implementations and the requirements used to select one to run. See Cross-platform tasks on page 470 for usage information.	Array of Objects describing each implementation	None
"files"	A list of files to be provided when running the task, addressed by module. See Sharing task code on page 472 for usage information.	Array of Strings	None

Task metadata types

Task metadata can accept most Puppet data types.

Restriction:

Some types supported by Puppet can not be represented as JSON, such as `Hash[Integer, String]`, `Object`, or `Resource`. These should not be used in tasks, because they can never be matched.

Table 3: Common task data types

Type	Description
<code>String</code>	Accepts any string.
<code>String[1]</code>	Accepts any non-empty string (a <code>String</code> of at least length 1).
<code>Enum[choice1, choice2]</code>	Accepts one of the listed choices.
<code>Pattern[/\A\w+\Z/]</code>	Accepts Strings matching the regex <code>/\w+/</code> or non-empty strings of word characters.
<code>Integer</code>	Accepts integer values. JSON has no <code>Integer</code> type so this can vary depending on input.
<code>Optional[String[1]]</code>	Optional makes the parameter optional and permits null values. Tasks have no required nullable values.
<code>Array[String]</code>	Matches an array of strings.
<code>Hash</code>	Matches a JSON object.
<code>Variant[Integer, Pattern[/\A\d+\Z/]]</code>	Matches an integer or a <code>String</code> of an integer
<code>Boolean</code>	Accepts Boolean values.

Specifying parameters

Parameters for tasks can be passed to the `bolt` command as CLI arguments or as a JSON hash.

To pass parameters individually to your task or plan, specify the parameter value on the command line in the format `parameter=value`. Pass multiple parameters as a space-separated list. Bolt attempts to parse each parameter value as JSON and compares that to the parameter type specified by the task or plan. If the parsed value matches the type, it is used; otherwise, the original string is used.

For example, to run the `mysql::sql` task to show tables from a database called `mydatabase`:

```
bolt task run mysql::sql database=mydatabase sql="SHOW TABLES" --nodes
neptune --modules ~/modules
```

To pass a string value that is valid JSON to a parameter that would accept both quote the string. For example to pass the string `true` to a parameter of type `Variant[String, Boolean]` use `'foo="true"'`. To pass a `String` value wrapped in `"` quote and escape it `'string="\val\"'`. Alternatively, you can specify parameters as a single JSON object with the `--params` flag, passing either a JSON object or a path to a parameter file.

To specify parameters as JSON, use the `parameters` flag followed by the JSON: `--params '{ "name": "openssl" }'`

To set parameters in a file, specify parameters in JSON format in a file, such as `params.json`. For example, create a `params.json` file that contains the following JSON:

```
{
  "name": "openssl"
}
```

Then specify the path to that file (starting with an at symbol, `@`) on the command line with the `parameters` flag: `--params @params.json`

Writing plans

Plans allow you to run more than one task with a single command, compute values for the input to a task, process the results of tasks, or make decisions based on the result of running a task.

Write plans in the Puppet language, giving them a `.pp` extension, and placing them in the module's `/plans` directory.

Plans can use any combination of Bolt functions or built-in Puppet functions.

Writing plans in YAML

YAML plans run a list of steps in order, which allows you to define simple workflows. Steps can contain embedded Puppet code expressions to add logic where necessary.

Note: YAML plans are an experimental feature and might experience breaking changes in future minor releases.

Naming plans

It is important to name plans correctly according to the module name, file name, and path to ensure easy code readability.

Place plan files in your module's `/plans` directory, using these file extensions:

- Puppet plans — `.pp`
- YAML plans — `.yaml`, not `.yml`

Plan names are composed of two or more name segments, indicating:

- The name of the module the plan is located in.
- The name of the plan file, without the extension.
- The path within the module, if the plan is in a subdirectory of `/plans`.

For example, given a module called `mymodule` with a plan defined in `/mymodule/plans/myplan.pp`, the plan name is `mymodule::myplan`.

A plan defined in `/mymodule/plans/service/myplan.pp` would be `mymodule::service::myplan`. This name is how you refer to the plan when you run commands.

The plan filename `init` is special: the plan it defines is referenced using the module name only. For example, in a module called `mymodule`, the plan defined in `init.pp` is the `mymodule` plan.

Avoid giving plans the same names as constructs in the Puppet language. Although plans do not share their namespace with other language constructs, giving plans these names makes your code difficult to read.

Each plan name segment must begin with a lowercase letter and:

- May include lowercase letters.
- May include digits.
- May include underscores.
- Must not be a [reserved word](#).
- Must not have the same name as any Puppet data types.
- Namespace segments must match the following regular expression `\A[a-z][a-z0-9_]*\Z`

RBAC considerations for writing plans

Take user permissions into consideration when writing plans by understanding how RBAC for plans works.

RBAC for plans is separate from RBAC for individual tasks. This means that a user can be excluded from running a certain task, but still have permission to run a plan that contains that task. Setting one permission does not affect the other.

This structure allows you to write plans with more robust, custom control over task permissions. Instead of allowing a user free rein to run a task that can potentially damage your infrastructure under the right conditions, you can wrap a task in a plan and only allow them to run it under circumstances you can control.

For example, say you are configuring permissions for a new user and allow them to run the plan `plan`

`infra::upgrade_git`:

```
plan infra::upgrade_git (
  TargetSpec $nodes,
  Integer $version,
) {
  run_task('package', $nodes, name => 'git', action => 'upgrade', version =>
    $version)
}
```

Within this plan, they can run the `package` task, but can only interact with the `git` package. The plan does not allow them to use any other parameters for the `package` task.

Even though they can run this plan, they do not have access to individually run the `package` task outside of this plan unless you grant them permission to do so. In that case, they would have the option to add any parameters they want to the task.

Use parameter types to fine-tune access

Writing parameter types into plan code provides even more control. In the `upgrade_git` example above, the plan only provides access to the `git` package, but the user can choose whatever version of `git` they want.

Let's say there are known vulnerabilities in some versions of the `git` package and you are concerned with your new user having the ability to use the versions you deem unsafe. You can use parameter types like `Enum` to restrict the version parameter to versions that are safe enough for deployment.

In this example, the `Enum` restricts the `$version` parameter to versions `1:2.17.0-lubuntu1` and `1:2.17.1-lubuntu0.4` only:

```
plan infra::upgrade_git (
  TargetSpec $nodes,
  Enum['1:2.17.0-lubuntu1', '1:2.17.1-lubuntu0.4'] $version,
) {
  run_task('package', $nodes, name => 'git', action => 'upgrade', version =>
    $version)
}
```

Any user attempting to run this plan must choose one of these versions for the plan to run.

You can also use PuppetDB queries to select parameter types. Using the same example, let's say you need to restrict the nodes that `infra::upgrade_git` can run on. Use a PuppetDB query to identify which nodes get selected for the `git` upgrade. It should look something like this:

```
plan infra::upgrade_git (
  Enum['1:2.17.0-lubuntu1', '1:2.17.1-lubuntu0.4'] $version,
) {
  # Use puppetdb to find the nodes from the "other" team's web cluster
  $query = [from, nodes, ['='], [fact, cluster], "other_team"]]
  $selected_nodes = puppetdb_query($query).map() |$target| {
    $target[certname]
  }
  run_task('package', $selected_nodes, name => 'git', action => 'upgrade',
    version => $version)
}
```

Using these ideas, you can write powerful plans that give users exactly what they need without giving them the keys to the kingdom.

Plan structure

YAML plans contain a list of steps with optional parameters and results.

YAML maps accept these keys:

- `steps`: The list of steps to perform
- `parameters`: (Optional) The parameters accepted by the plan
- `return`: (Optional) The value to return from the plan

Steps key

The `steps` key is an array of step objects, each of which corresponds to a specific action to take.

When the plan runs, each step is executed in order. If a step fails, the plan halts execution and raises an error containing the result of the step that failed.

Steps use these fields:

- `name`: A unique name that can be used to refer to the result of the step later
- `description`: (Optional) An explanation of what the step is doing

Other available keys depend on the type of step.

Command step

Use a command step to run a single command on a list of targets and save the results, containing stdout, stderr, and exit code.

The step fails if the exit code of any command is non-zero.

Command steps use these fields:

- `command`: The command to run
- `target`: A target or list of targets to run the command on

For example:

```
steps:
  - command: hostname -f
    target:
      - web1.example.com
      - web2.example.com
      - web3.example.com
    description: "Get the webserver hostnames"
```

Task step

Use a task step to run a Bolt task on a list of targets and save the results.

Task steps use these fields:

- `task`: The task to run
- `target`: A target or list of targets to run the task on
- `parameters`: (Optional) A map of parameter values to pass to the task

For example:

```
steps:
  - task: package
    target:
      - web1.example.com
      - web2.example.com
      - web3.example.com
    description: "Check the version of the openssl package on the
webserver"
```

```
parameters:
  action: status
  name: openssl
```

Script step

Use a script step to run a script on a list of targets and save the results.

The script must be in the `files/` directory of a module. The name of the script must be specified as `<modulename>/path/to/script`, omitting the `files` directory from the path.

Script steps use these fields:

- `script`: The script to run
- `target`: A target or list of targets to run the script on
- `arguments`: (Optional) An array of command-line arguments to pass to the script

For example:

```
steps:
- script: mymodule/check_server.sh
  target:
    - web1.example.com
    - web2.example.com
    - web3.example.com
  description: "Run mymodule/files/check_server.sh on the webservers"
  arguments:
    - "/index.html"
    - 60
```

File upload step

Use a file upload step to upload a file to a specific location on a list of targets.

The file to upload must be in the `files/` directory of a Puppet module. The source for the file must be specified as `<modulename>/path/to/file`, omitting the `files` directory from the path.

File upload steps use these fields:

- `source`: The location of the file to be uploaded
- `destination`: The location to upload the file to

For example:

```
steps:
- source: mymodule/motd.txt
  destination: /etc/motd
  target:
    - web1.example.com
    - web2.example.com
    - web3.example.com
  description: "Upload motd to the webservers"
```

Plan step

Use a plan step to run another plan and save its result.

Plan steps use these fields:

- `plan`: The name of the plan to run
- `parameters`: (Optional) A map of parameter values to pass to the plan

For example:

```
steps:
  - plan: facts
    description: "Gather facts for the webserver using the built-in facts plan"
    parameters:
      nodes:
        - web1.example.com
        - web2.example.com
        - web3.example.com
```

Resources step

Use a `resources` step to apply a list of Puppet resources. A resource defines the desired state for part of a target. Bolt ensures each resource is in its desired state. Like the steps in a plan, if any resource in the list fails, the rest are skipped.

For each `resources` step, Bolt executes the `apply_prep` plan function against the targets specified with the `targets` field. For more information about `apply_prep` see the Applying manifest block section.

Resources steps use these fields:

- `resources`: An array of resources to apply
- `target`: A target or list of targets to apply the resources on

Each resource is a YAML map with a `type` and `title`, and optionally a `parameters` key. The resource type and title can either be specified separately with the `type` and `title` keys, or can be specified in a single line by using the type name as a key with the title as its value.

For example:

```
steps:
  - resources:
    # This resource is type 'package' and title 'nginx'
    - package: nginx
      parameters:
        ensure: latest
    # This resource is type 'service' and title 'nginx'
    - type: service
      title: nginx
      parameters:
        ensure: running
    target:
      - web1.example.com
      - web2.example.com
      - web3.example.com
    description: "Set up nginx on the webserver"
```

Parameters key

Plans accept parameters with the `parameters` key. The value of `parameters` is a map, where each key is the name of a parameter and the value is a map describing the parameter.

Parameter values can be referenced from steps as variables.

Parameters use these fields:

- `type`: (Optional) A valid [Puppet data type](#). The value supplied must match the type or the plan fails.
- `default`: (Optional) Used if no value is given for the parameter
- `description`: (Optional)

For example, this plan accepts a `load_balancer` name as a string, two sets of nodes called `frontends` and `backends`, and a version string:

```
parameters:
  # A simple parameter definition doesn't need a type or description
  load_balancer:
  frontends:
    type: TargetSpec
    description: "The frontend web servers"
  backends:
    type: TargetSpec
    description: "The backend application servers"
  version:
    type: String
    description: "The new application version to deploy"
```

How strings are evaluated

The behavior of strings is defined by how they're written in the plan.

'single-quoted strings' are treated as string literals without any interpolation.

"double-quoted strings" are treated as Puppet language double-quoted strings with variable interpolation.

| block-style strings are treated as expressions of arbitrary Puppet code. Note the string itself must be on a new line after the | character.

bare strings are treated dynamically based on their content. If they begin with a \$, they're treated as Puppet code expressions. Otherwise, they're treated as YAML literals.

Here's an example of different kinds of strings in use:

```
parameters:
  message:
    type: String
    default: "hello"

steps:
  - eval: hello
    description: 'This will evaluate to: hello'
  - eval: $message
    description: 'This will evaluate to: hello'
  - eval: '$message'
    description: 'This will evaluate to: $message'
  - eval: "${message} world"
    description: 'This will evaluate to: hello world'
  - eval: |
    [$message, $message, $message].join(" ")
    description: 'This will evaluate to: hello hello hello'
```

Using variables and simple expressions

The simplest way to use a variable is to reference it directly by name. For example, this plan takes a parameter called `nodes` and passes it as the target list to a step:

```
parameters:
  nodes:
    type: TargetSpec

steps:
  - command: hostname -f
    target: $nodes
```

Variables can also be interpolated into string values. The string must be double-quoted to allow interpolation. For example:

```
parameters:
  username:
    type: String

steps:
  - task: echo
    message: "hello ${username}"
    target: $nodes
```

Many operations can be performed on variables to compute new values for step parameters or other fields.

Indexing arrays or hashes

You can retrieve a value from an Array or a Hash using the `[]` operator. This operator can also be used when interpolating a value inside a string.

```
parameters:
  users:
    # Array[String] is a Puppet data type representing an array of strings
    type: Array[String]

steps:
  - task: user::add
    target: 'host.example.com'
    parameters:
      name: $users[0]
  - task: echo
    target: 'host.example.com'
    parameters:
      message: "hello ${users[0]}"
```

Calling functions

You can call a built-in [Bolt function](#) or [Puppet function](#) to compute a value.

```
parameters:
  users:
    type: Array[String]

steps:
  - task: user::add
    parameters:
      name: $users.first
  - task: echo
    message: "hello ${users.join(',')}"
```

Using code blocks

Some Puppet functions take a block of code as an argument. For instance, you can filter an array of items based on the result of a block of code.

The result of the `filter` function is an array here, not a string, because the expression isn't inside quotes

```
parameters:
  numbers:
    type: Array[Integer]

steps:
```

```
- task: sum
  description: "add up the numbers > 5"
  parameters:
    indexes: $numbers.filter |$num| { $num > 5 }
```

Connecting steps

You can connect multiple steps by using the result of one step to compute the parameters for another step.

name key

The name key makes its results available to later steps in a variable with that name.

This example uses the map function to get the value of stdout from each command result and then joins them into a single string separated by commas.

```
parameters:
  nodes:
    type: TargetSpec

steps:
- name: hostnames
  command: hostname -f
  target: $nodes
- task: echo
  parameters:
    message: $hostnames.map |$hostname_result|
    { $hostname_result['stdout'] }.join(',')
}
```

eval step

The eval step evaluates an expression and saves the result in a variable. This is useful to compute a variable to use multiple times later.

```
parameters:
  count:
    type: Integer

steps:
- name: double_count
  eval: $count * 2
- task: echo
  target: web1.example.com
  parameters:
    message: "The count is ${count}, and twice the count is
    ${double_count}"
}
```

Returning results

You can return a result from a plan by setting the return key at the top level of the plan. When the plan finishes, the return key is evaluated and returned as the result of the plan. If no return key is set, the plan returns undef

```
steps:
- name: hostnames
  command: hostname -f
  target: $nodes

return: $hostnames.map |$hostname_result| { $hostname_result['stdout'] }
```

Computing complex values

To compute complex values, you can use a Puppet code expression as the value of any field of a step except the `name`.

Bolt loads the plan as a YAML data structure. As it executes each step, it evaluates any expressions embedded in the step. Each plan parameter and the values of every previous named step are available in scope.

This lets you take advantage of the power of Puppet language in the places it's necessary, while keeping the rest of your plan simple.

When your plans need more sophisticated control flow or error handling beyond running a list of steps in order, it's time to convert them to [Puppet Language plans](#).

Converting YAML plans to Puppet language plans

You can convert a YAML plan to a Puppet language plan with the `bolt plan convert` command.

```
bolt plan convert path/to/my/plan.yaml
```

This command takes the relative or absolute path to the YAML plan to be converted and prints the converted Puppet language plan to stdout.

Note: Converting a YAML plan might result in a Puppet plan which is syntactically correct, but behaves differently. Always manually verify a converted Puppet language plan's functionality. There are some constructs that do not translate from YAML plans to Puppet language plans. These are listed [TODO: insert link to section below!] below. If you convert a YAML plan to Puppet and it changes behavior, [file an issue](#) in Bolt's Git repo.

For example, with this YAML plan:

```
# site-modules/mymodule/plans/yamlplan.yaml
parameters:
  nodes:
    type: TargetSpec
steps:
  - name: run_task
    task: sample
    target: $nodes
    parameters:
      message: "hello world"
return: $run_task
```

Run the following conversion:

```
$ bolt plan convert site-modules/mymodule/plans/yamlplan.yaml
# WARNING: This is an autogenerated plan. It may not behave as expected.
plan mymodule::yamlplan(
  TargetSpec $nodes
) {
  $run_task = run_task('sample', $nodes, {'message' => "hello world"})
  return $run_task
}
```

Quirks when converting YAML plans to Puppet language

There are some quirks and limitations associated with converting a plan expressed in YAML to a plan expressed in the Puppet language. In some cases it is impossible to accurately translate from YAML to Puppet. In others, code that is generated from the conversion is syntactically correct but not idiomatic Puppet code.

Named `eval` step

The `eval` step allows snippets of Puppet code to be expressed in YAML plans. When converting a multi-line `eval` step to Puppet code and storing the result in a variable, use the `with lambda`.

For example, here is a YAML plan with a multi-line eval step:

```
parameters:
  foo:
    type: Optional[Integer]
    description: foo
    default: 0

steps:
  - eval: |
      $x = $foo + 1
      $x * 2
    name: eval_step

return: $eval_step
```

And here is the same plan, converted to the Puppet language:

```
plan yml_plans::with_lambda(
  Optional[Integer] $foo = 0
) {
  $eval_step = with() || {
    $x = $foo + 1
    $x * 2
  }

  return $eval_step
}
```

Writing this plan from scratch using the Puppet language, you would probably not use the lambda. In this example the converted Puppet code is correct, but not as natural or readable as it could be.

Resource step variable interpolation

When applying Puppet resources in a resource step, variable interpolation behaves differently in YAML plans and Puppet language plans. To illustrate this difference, consider this YAML plan:

```
steps:
  - target: localhost
    description: Apply a file resource
    resources:
      - type: file
        title: '/tmp/foo'
        parameters:
          content: $facts['os']['family']
          ensure: present
      - name: file_contents
        description: Read contents of file managed with file resource
        eval: >
          file::read('/tmp/foo')

return: $file_contents
```

This plan performs `apply_prep` on a `localhost` target. Then it uses a Puppet `file` resource to write the OS family discovered from the Puppet `$facts` hash to a temporary file. Finally, it reads the value written to the file and returns it. Running `bolt plan convert` on this plan produces this Puppet code:

```
plan yml_plans::interpolation_pp() {
  apply_prep('localhost')
  $interpolation = apply('localhost') {
    file { '/tmp/foo':
```

```

        content => $facts['os']['family'],
        ensure => 'present',
      }
    }
  }
  $file_contents = file::read('/tmp/foo')

  return $file_contents
}

```

This Puppet language plan works as expected, whereas the YAML plan it was converted from fails. The failure stems from the `$facts` variable being resolved as a plan variable, instead of being evaluated as part of compiling the manifest code in an `applyblock`.

Dependency order

The resources in a `resources` list are applied in order. It is possible to set dependencies explicitly, but when doing so you must refer to them in a particular way. Consider the following YAML plan:

```

parameters:
  nodes:
    type: TargetSpec
steps:
  - name: pkg
    target: $nodes
    resources:
      - title: openssh-server
        type: package
        parameters:
          ensure: present
          before: File['/etc/ssh/sshd_config']
      - title: /etc/ssh/sshd_config
        type: file
        parameters:
          ensure: file
          mode: '0600'
          content: ''
          require: Package['openssh-server']

```

Executing this plan fails during catalog compilation because of how Bolt parses the resources referenced in the `before` and `require` parameters. You will see the error message `Could not find resource 'File['/etc/ssh/sshd_config']' in parameter 'before'`. The solution is to not quote the resource titles:

```

parameters:
  nodes:
    type: TargetSpec
steps:
  - name: pkg
    target: $nodes
    resources:
      - title: openssh-server
        type: package
        parameters:
          ensure: present
          before: File[/etc/ssh/sshd_config]
      - title: /etc/ssh/sshd_config
        type: file
        parameters:
          ensure: file
          mode: '0600'
          content: ''
          require: Package[openssh-server]

```

In general, declare resources in order. This is an unusual example to illustrate a case where parameter parsing leads to non-intuitive results.

Writing plans in Puppet Language

Plans allow you to run more than one task with a single command, compute values for the input to a task, process the results of tasks, or make decisions based on the result of running a task.

Write plans in the Puppet language, giving them a `.pp` extension, and place them in the module's `/plans` directory.

Plans can use any combination of Bolt functions or built in Puppetfunctions.

Naming plans

It is important to name plans correctly according to the module name, file name, and path to ensure easy code readability.

Place plan files in your module's `/plans` directory, using these file extensions:

- Puppet plans — `.pp`
- YAML plans — `.yaml`, not `.yml`

Plan names are composed of two or more name segments, indicating:

- The name of the module the plan is located in.
- The name of the plan file, without the extension.
- The path within the module, if the plan is in a subdirectory of `/plans`.

For example, given a module called `mymodule` with a plan defined in `./mymodule/plans/myplan.pp`, the plan name is `mymodule::myplan`.

A plan defined in `./mymodule/plans/service/myplan.pp` would be `mymodule::service::myplan`. This name is how you refer to the plan when you run commands.

The plan filename `init` is special: the plan it defines is referenced using the module name only. For example, in a module called `mymodule`, the plan defined in `init.pp` is the `mymodule` plan.

Avoid giving plans the same names as constructs in the Puppet language. Although plans do not share their namespace with other language constructs, giving plans these names makes your code difficult to read.

Each plan name segment must begin with a lowercase letter and:

- May include lowercase letters.
- May include digits.
- May include underscores.
- Must not be a [reserved word](#).
- Must not have the same name as any Puppet data types.
- Namespace segments must match the following regular expression `\A[a-z][a-z0-9_]*\Z`

RBAC considerations for writing plans

Take user permissions into consideration when writing plans by understanding how RBAC for plans works.

RBAC for plans is separate from RBAC for individual tasks. This means that a user can be excluded from running a certain task, but still have permission to run a plan that contains that task. Setting one permission does not affect the other.

This structure allows you to write plans with more robust, custom control over task permissions. Instead of allowing a user free rein to run a task that can potentially damage your infrastructure under the right conditions, you can wrap a task in a plan and only allow them to run it under circumstances you can control.

For example, say you are configuring permissions for a new user and allow them to run the plan `plan infra::upgrade_git`:

```
plan infra::upgrade_git (
  TargetSpec $nodes,
```

```

    Integer $version,
  ) {
    run_task('package', $nodes, name => 'git', action => 'upgrade', version =>
      $version)
  }

```

Within this plan, they can run the `package` task, but can only interact with the `git` package. The plan does not allow them to use any other parameters for the `package` task.

Even though they can run this plan, they do not have access to individually run the `package` task outside of this plan unless you grant them permission to do so. In that case, they would have the option to add any parameters they want to the task.

Use parameter types to fine-tune access

Writing parameter types into plan code provides even more control. In the `upgrade_git` example above, the plan only provides access to the `git` package, but the user can choose whatever version of `git` they want.

Let's say there are known vulnerabilities in some versions of the `git` package and you are concerned with your new user having the ability to use the versions you deem unsafe. You can use parameter types like `Enum` to restrict the version parameter to versions that are safe enough for deployment.

In this example, the `Enum` restricts the `$version` parameter to versions `1:2.17.0-lubuntu1` and `1:2.17.1-lubuntu0.4` only:

```

plan infra::upgrade_git (
  TargetSpec $nodes,
  Enum['1:2.17.0-lubuntu1', '1:2.17.1-lubuntu0.4'] $version,
) {
  run_task('package', $nodes, name => 'git', action => 'upgrade', version =>
    $version)
}

```

Any user attempting to run this plan must choose one of these versions for the plan to run.

You can also use PuppetDB queries to select parameter types. Using the same example, let's say you need to restrict the nodes that `infra::upgrade_git` can run on. Use a PuppetDB query to identify which nodes get selected for the `git` upgrade. It should look something like this:

```

plan infra::upgrade_git (
  Enum['1:2.17.0-lubuntu1', '1:2.17.1-lubuntu0.4'] $version,
) {
  # Use puppetdb to find the nodes from the "other" team's web cluster
  $query = [from, nodes, ['=', [fact, cluster], "other_team"]]
  $selected_nodes = puppetdb_query($query).map() |$target| {
    $target[certname]
  }
  run_task('package', $selected_nodes, name => 'git', action => 'upgrade',
    version => $version)
}

```

Using these ideas, you can write powerful plans that give users exactly what they need without giving them the keys to the kingdom.

Defining plan parameters

Specify parameters within your plan.

Specify each parameter in your plan with its data type. For example, you might want parameters to specify which nodes to run different parts of your plan on.

The following example shows node parameters specified as data type `TargetSpec`. This allows this parameter to be passed as a single URL, comma-separated URL list, `Target` data type, or `Array` of either. For more information about these data types, see the common data types table in the related metadata type topic.

This allows the user to pass, for each parameter, either a node name or a URI that describes the protocol to use, the hostname, username, and password.

The plan then calls the `run_task` function, specifying which nodes to run the tasks on. The `Target` names are collected and stored in `$webserver_names` by iterating over the list of `Target` objects returned by `get_targets`. Task parameters are serialized to JSON format; therefore, extracting the names into an array of strings ensures that the `webserver`s parameter is in a format that can be converted to JSON.

```
plan mymodule::my_plan(
  TargetSpec $load_balancer,
  TargetSpec $webserver,
) {

  # Extract the Target name from $webserver
  $webserver_names = get_targets($webserver).map |$n| { $n.name }

  # process webserver
  run_task('mymodule::lb_remove', $load_balancer, webserver =>
    $webserver_names)
  run_task('mymodule::update_frontend_app', $webserver, version => '1.2.3')
  run_task('mymodule::lb_add', $load_balancer, webserver =>
    $webserver_names)
}
```

To execute this plan from the command line, pass the parameters as `parameter=value`. The `TargetSpec` accepts either an array as json or a comma separated string of target names.

```
bolt plan run mymodule::myplan --modulepath ./PATH/TO/MODULES
  load_balancer=lb.myorg.com
  webserver='["kermit.myorg.com", "gonzo.myorg.com"]'
```

Parameters that are passed to the `run_*` plan functions are serialized to JSON.

To illustrate this concept, consider this plan:

```
plan test::parameter_passing (
  TargetSpec $nodes,
  Optional[String[1]] $example_nul = undef,
) {
  return run_task('test::demo_undef_bash', $nodes, example_nul =>
    $example_nul)
}
```

The default value of `$example_nul` is `undef`. The plan calls the `test::demo_undef_bash` with the `example_nul` parameter. The implementation of the `demo_undef_bash.sh` task is:

```
#!/bin/bash
example_env=$PT_example_nul
echo "Environment: $PT_example_nul"
echo "Stdin:"
cat -
```

By default, the task expects parameters passed as a JSON string on stdin to be accessible in prefixed environment variables.

Consider the output of running the plan against localhost:

```
bolt@bolt: bolt plan run test::parameter_passing -n localhost
Starting: plan test::parameter_passing
Starting: task test::demo_undef_bash on localhost
Finished: task test::demo_undef_bash with 0 failures in 0.0 sec
Finished: plan test::parameter_passing in 0.01 sec
Finished on localhost:
  Environment: null
  Stdin:
  { "example_nul":null, "_task":"test::demo_undef_bash" }
  {
  }
Successful on 1 node: localhost
Ran on 1 node
```

The parameters `example_nul` and `_task` metadata are passed to the task as a JSON string over stdin.

Similarly, parameters are made available to the task as environment variables where the name of the parameter is converted to an environment variable prefixed with `PT_`. The prefixed environment variable points to the `String` representation in JSON format of the parameter value. So, the `PT_example_nul` environment variable has the value of `null` of type `String`.

Returning results from plans

Use plans to return results that you can use in other plans or save for use outside of Bolt

Plans, unlike functions, are primarily run for side effects but they can optionally return a result. To return a result from a plan use the `return` function. Any plan that does not call the `return` function returns `undef`.

```
plan return_result(
  $nodes
) {
  return run_task('mytask', $nodes)
}
```

The result of a plan must match the `PlanResult` type alias. This roughly includes JSON types as well as the Plan language types which have well defined JSON representations in Bolt.

- `Undef`
- `String`
- `Numeric`
- `Boolean`
- `Target`
- `Result`
- `ResultSet`
- `Error`
- Array with only `PlanResult`
- Hash with `String` keys and `PlanResult` values

or

```
Variant[Data, String, Numeric, Boolean, Error, Result, ResultSet, Target,
  Array[Boltlib::PlanResult], Hash[String, Boltlib::PlanResult]]
```

Returning errors in plans

To return an error if your plan fails, call the `fail_plan` function.

Specify parameters to provide details about the failure.

For example, if called with `run_plan('mymodule::myplan')`, this would return an error to the caller.

```
plan mymodule::myplan {
  Error(
    message => "Sorry, this plan does not work yet.",
    kind     => 'mymodule/error',
    issue_code => 'NOT_IMPLEMENTED'
  )
}
fail_plan("Sorry, this plan does not work yet.", 'mymodule/error')
```

Plan success and failure

There are indicators that a plan has run successfully or failed.

Any plan that completes execution without an error is considered successful. The `bolt` command exits 0 and any calling plans continue execution. If any calls to `run_` functions fail **without** `_catch_errors` then the plan halts execution and is considered a failure. Any calling plans also halt until a `run_plan` call with `_catch_errors` or a `catch_errors` block is reached. If one isn't reached, the `bolt` command performs an exit 2. When writing a plan if you have reason to believe it has failed, you can fail the plan with the `fail_plan` function. This causes the `bolt` command to exit 2 and prevents calling plans executing any further, unless `run_plan` was called with `_catch_errors` or in a `catch_errors` block.

Failing plans

If `upload_file`, `run_command`, `run_script`, or `run_task` are called without the `_catch_errors` option and they fail on any nodes, the plan itself fails. To fail a plan directly call the `fail_plan` function. Create a new error with a message and include the kind, details, or issue code, or pass an existing error to it.

```
fail_plan('The plan is failing', 'mymodules/pear-shaped', {'failednodes' =>
  $result.error_set.names})
# or
fail_plan($errorobject)
```

Catching errors in plans

Bolt includes a `catch_errors` function that executes a block of code and returns the error if an error is raised, or returns the result of the block if no errors are raised. You might get an `Error` object returned if you call `run_plan` with `_catch_errors`, use a `catch_errors` block, or call the `Error` method on a result.

The `Error` data type includes:

- `msg`: The error message string.
- `kind`: A string that defines the kind of error similar to an error class.
- `details`: A hash with details about the error from a task or from information about the state of a plan when it fails, for example, `exit_code` or `stack_trace`.
- `issue_code`: A unique code for the message that can be used for translation.

Use the `Error` data type in a case expression to match against different kind of errors. To recover from certain errors, while failing on or ignoring others, set up your plan to include conditionals based on errors that occur while your plan runs. For example, you can set up a plan to retry a task when a timeout error occurs, but to fail when there is an authentication error.

Below, the first plan continues whether it succeeds or fails with a `mymodule/not-serious` error. Other errors cause the plan to fail.

```
plan mymodule::handle_errors {
  $result = run_plan('mymodule::myplan', '_catch_errors' => true)
  case $result {
    Error['mymodule/not-serious'] : {
```

```

        notice("${result.message}")
    }
    Error : { fail_plan($result) } }
    run_plan('mymodule::plan2')
}

```

Using the `catch_errors` function:

```

plan test (String[1] $role) {
    $result_or_error = catch_errors(['bolt/puppetdb-error']) || {
        puppetdb_query("inventory[certname] { app_role == ${role} }")
    }
    $targets = if $result_or_error =~ Error {
        # If the PuppetDB query fails
        warning("Could not fetch from puppet. Using defaults instead")
        # TargetSpec string
        "all"
    } else {
        $result_or_error
    }
}

```

Puppet and Ruby functions in plans

You can define and call Puppet language and Ruby functions in plans.

This is useful for packaging common general logic in your plan. You can also call the plan functions, such as `run_task` or `run_plan`, from within a function.

Not all Puppet language constructs are allowed in plans. The following constructs are not allowed:

- Defined types.
- Classes.
- Resource expressions, such as `file { title: mode => '0777' }`
- Resource default expressions, such as `File { mode => '0666' }`
- Resource overrides, such as `File['/tmp/foo'] { mode => '0444' }`
- Relationship operators: `->` `<-` `~>` `<~`
- Functions that operate on a catalog: `include`, `require`, `contain`, `create_resources`.
- Collector expressions, such as `SomeType <| |>`, `SomeType <<| |>>`
- ERB templates are not supported. Use EPP instead.

Be aware of a few other Puppet behaviors in plans:

- The `--strict_variables` option is on, so if you reference a variable that is not set, you get an error.
- `--strict=error` is always on, so minor language issues generate errors. For example `{ a => 10, a => 20 }` is an error because there is a duplicate key in the hash.
- Most Puppet settings are empty and not-configurable when using Bolt.
- Logs include "source location" (file, line) instead of resource type or name.

Handling plan function results

Each execution function returns an object type `ResultSet`. For each node that the execution takes place on, this object contains a `Result` object. The `apply` action returns a `ResultSet` containing `ApplyResult` objects.

A `ResultSet` has the following methods:

- `names()`: The `String` names (node URIs) of all nodes in the set as an `Array`.
- `empty()`: Returns `Boolean` if the execution result set is empty.
- `count()`: Returns an `Integer` count of nodes.
- `first()`: The first `Result` object, useful to unwrap single results.
- `find(String $target_name)`: Look up the `Result` for a specific target.

- `error_set()`: A `ResultSet` containing only the results of failed nodes.
- `ok_set()`: A `ResultSet` containing only the successful results.
- `filter_set(block)`: Filters a `ResultSet` with the given block and returns a `ResultSet` object (where the [filter function](#) returns an array or hash).
- `targets()`: An array of all the `Target` objects from every `Result` in the set.
- `ok()`: Boolean that is the same as `error_nodes.empty`.
- `to_data()`: An array of hashes representing either `Result` or `ApplyResults`.

A `Result` has the following methods:

- `value()`: The hash containing the value of the `Result`.
- `target()`: The `Target` object that the `Result` is from.
- `error()`: An `Error` object constructed from the `_error` in the value.
- `message()`: The `_output` key from the value.
- `ok()`: Returns `true` if the `Result` was successful.
- `[]`: Accesses the value hash directly.
- `to_data()`: Hash representation of `Result`.
- `action()`: String representation of result type (task, command, etc.).

An `ApplyResult` has the following methods:

- `report()`: The hash containing the Puppet report from the application.
- `target()`: The `Target` object that the `Result` is from.
- `error()`: An `Error` object constructed from the `_error` in the value.
- `ok()`: Returns `true` if the `Result` was successful.
- `to_data()`: Hash representation of `ApplyResult`.
- `action()`: String representation of result type (apply).

An instance of `ResultSet` is `Iterable` as if it were an `Array[Variant[Result, ApplyResult]]` so that iterative functions such as `each`, `map`, `reduce`, or `filter` work directly on the `ResultSet` returning each result.

This example checks if a task ran correctly on all nodes. If it did not, the check fails:

```
$r = run_task('sometask', ..., '_catch_errors' => true)
unless $r.ok {
  fail("Running sometask failed on the nodes ${r.error_nodes.names}")
}
```

You can do iteration and checking if the result is an `Error`. This example outputs feedback about the result of a task:

```
$r = run_task('sometask', ..., '_catch_errors' => true)
$r.each |$result| {
  $node = $result.target.name
  if $result.ok {
    notice("${node} returned a value: ${result.value}")
  } else {
    notice("${node} errored with a message: ${result.error.message}")
  }
}
```

Similarly, you can iterate over the array of hashes returned by calling `to_data` on a `ResultSet` and access hash values. For example:

```
$r = run_command('whoami', 'localhost,local://0.0.0.0')
$r.to_data.each |$result_hash| { notice($result_hash['result']['stdout']) }
```

You can also use `filter_set` to filter a `ResultSet` and apply a `ResultSet` function such as `targets` to the output:

```
$filtered = $result.filter_set |$r| {
  $r['tag'] == "you're it"
}.targets
```

Passing sensitive data to tasks

Task parameters defined as sensitive are masked when they appear in plans.

You define a task parameter as sensitive with the metadata property `"sensitive": true`. When a task runs, the values for these sensitive parameters are masked.

```
run_task('task_with_secrets', ..., password => '$secret!')
```

Working with the sensitive function

In Puppet you use the `Sensitive` function to mask data in output logs. Because plans are written in Puppet DSL, you can use this type freely. The `run_task()` function does not allow parameters of `Sensitive` function to be passed. When you need to pass a sensitive value to a task, you must unwrap it prior to calling `run_task()`.

```
$pass = Sensitive('$secret!')
run_task('task_with_secrets', ..., password => $pass.unwrap)
```

Target objects

The `Target` object represents a node and its specific connection options.

The state of a target is stored in the inventory for the duration of a plan allowing you to collect facts or set vars for a target and retrieve them later. You can get a printable representation via the `name` function, as well as access components of the target: `protocol`, `host`, `port`, `user`, `password`.

TargetSpec

The execution function takes a parameter with the type alias `TargetSpec`. This alias accepts the pattern strings allowed by `--nodes`, a single `Target` object, or an `Array` of `Targets` and node patterns. Generally, use this type for plans that accept a set of targets as a parameter, to ensure clean interaction with the CLI and other plans. To operate on individual nodes, resolve it to a list via `get_targets`. For example, to loop over each node in a plan accept a `TargetSpec` argument, but call `get_targets` on it before looping.

```
plan loop(TargetSpec $nodes) {
  get_targets($nodes).each |$target| {
    run_task('my_task', $target)
  }
}
```

If your plan accepts a single `TargetSpec` parameter you can call that parameter `nodes` so that it can be specified with the `--nodes` flag from the command line.

Variables and facts on targets

When Bolt runs, it loads transport config values, variables, and facts from the inventory. These can be accessed with the `$target.facts()` and `$target.vars()` functions. During the course of a plan, you can update the facts or variables for any target. Facts usually come from running `facter` or another fact collection application on the target or from a fact store like `PuppetDB`. Variables are computed externally or assigned directly.

Set variables in a plan using `$target.set_var`:

```
plan vars(String $host) {
  $target = get_targets($host)[0]
```

```

$target.set_var('newly_provisioned', true)
$targetvars = $target.vars
run_command("echo 'Vars for ${host}': ${$targetvars}'", $host)
}

```

Or set variables in the inventory file using the `vars` key at the group level.

```

groups:
- name: my_nodes
  nodes:
  - localhost
  vars:
    operatingsystem: windows
  config:
    transport: ssh

```

Collect facts from the targets

The `facts` plan connects to the target and discovers facts. It then stores these facts on the targets in the inventory for later use.

The methods used to collect facts:

- On `ssh` targets, it runs a Bash script.
- On `winrm` targets, it runs a PowerShell script.
- On `pcp` or targets where the Puppet agent is present, it runs `Facter`.

This example collects facts with the `facts` plan and then uses those facts to decide which task to run on the targets.

```

plan run_with_facts(TargetSpec $nodes) {
  # This collects facts on nodes and update the inventory
  run_plan(facts, nodes => $nodes)

  $centos_nodes = get_targets($nodes).filter |$n| { $n.facts['os']['name']
  == 'CentOS' }
  $ubuntu_nodes = get_targets($nodes).filter |$n| { $n.facts['os']['name']
  == 'Ubuntu' }
  run_task(centos_task, $centos_nodes)
  run_task(ubuntu_task, $ubuntu_nodes)
}

```

Collect facts from PuppetDB

When targets are running a Puppet agent and sending facts to PuppetDB, you can use the `puppetdb_fact` plan to collect facts for them. This example collects facts with the `puppetdb_fact` plan, and then uses those facts to decide which task to run on the targets. You must configure the PuppetDB client before you run it.

```

plan run_with_facts(TargetSpec $nodes) {
  # This collects facts on nodes and update the inventory
  run_plan(puppetdb_fact, nodes => $nodes)

  $centos_nodes = get_targets($nodes).filter |$n| { $n.facts['os']['name']
  == 'CentOS' }
  $ubuntu_nodes = get_targets($nodes).filter |$n| { $n.facts['os']['name']
  == 'Ubuntu' }
  run_task(centos_task, $centos_nodes)
  run_task(ubuntu_task, $ubuntu_nodes)
}

```

Collect general data from PuppetDB

You can use the `puppetdb_query` function in plans to make direct queries to PuppetDB. For example you can discover nodes from PuppetDB and then run tasks on them. You'll have to configure the PuppetDB client before running it. You can learn how to [structure pql queries here](#), and find [pql reference and examples here](#)

```
plan pdb_discover {
  $result = puppetdb_query("inventory[certname] { app_role ==
'web_server' }")
  # extract the certnames into an array
  $names = $result.map |$r| { $r["certname"] }
  # wrap in url. You can skip this if the default transport is pcp
  $nodes = $names.map |$n| { "pcp://${n}" }
  run_task('my_task', $nodes)
}
```

Plan logging

Plan run information can be captured in log files or printed to a terminal session using the following methods.

Outputting section to the terminal

Print message strings to STDOUT using the plan function `out::message`. This function always prints messages regardless of the log level and doesn't log them to the log file.

Puppet log functions

To generate log messages from a plan, use the Puppet log function that corresponds to the level you want to track: `error`, `warn`, `notice`, `info`, or `debug`. Configure the log level for both log files and console logging in `bolt.yaml`. The default log level for the console is `warn` and for log files is `notice`. Use the `--debug` flag to set the console log level to `debug` for a single run.

Default action logging

Bolt logs actions that a plan takes on targets through the `upload_file`, `run_command`, `run_script`, or `run_task` functions. By default it logs a `notice` level message when an action starts and another when it completes. If you pass a description to the function, that is used in place of the generic log message.

```
run_task(my_task, $targets, "Better description", param1 => "val")
```

If your plan contains many small actions you may want to suppress these messages and use explicit calls to the Puppet log functions instead. This can be accomplished by wrapping actions in a `without_default_logging` block which causes the action messages to be logged at `info` level instead of `notice`. For example to loop over a series of nodes without logging each action.

```
plan deploy( TargetSpec $nodes) {
  without_default_logging() || {
    get_targets($nodes).each |$node| {
      run_task(deploy, $node)
    }
  }
}
```

To avoid complications with parser ambiguity, always call `without_default_logging` with `()` and empty block args `||`.

```
without_default_logging() || { run_command('echo hi', $nodes) }
```


not

```
without_default_logging { run_command('echo hi', $nodes) }
```

Example plans

Check out some example plans for inspiration writing your own.

Resource	Description	Level
facts module	Contains tasks and plans to discover facts about target systems.	Getting started
facts plan	Gathers facts using the facts task and sets the facts in inventory.	Getting started
facts::info plan	Uses the facts task to discover facts and map relevant fact values to targets.	Getting started
reboot module	Contains tasks and plans for managing system reboots.	Intermediate
reboot plan	Restarts a target system and waits for it to become available again.	Intermediate
Introducing Masterless Puppet with Bolt	Blog post explaining how plans can be used to deploy a load-balanced web server.	Advanced
profiles::nginx_install plan	Shows an example plan for deploying Nginx and HAProxy.	Advanced

- **Getting started** resources show simple use cases such as running a task and manipulating the results.
- **Intermediate** resources show more advanced features in the plan language.
- **Advanced** resources show more complex use cases such as applying puppet code blocks and using external modules.

Reviewing jobs

You can review jobs—on-demand Puppet run, task, plan, or scheduled—from the console or the command line.

Review jobs from the console

View a list of recent jobs and drill down to see useful details about each one.

Job list page

The **Job list** page provides a quick glance at key information about jobs run from the console. It also gathers information for jobs run from the orchestrator command-line interface.

Jobs are organized by type: **Puppet run**, **Task**, **Plan** (a plan combines multiple tasks and runs them with a single Bolt command) and **Scheduled**. And they are sorted by the date and time of the job's most recent status change; when the job started, stopped, completed or is scheduled to run.

Puppet run, Task, and Plan jobs have one of the following statuses:

- In progress
- Succeeded
- Finished with failures
- Stopping

- Stopped

View the Jobs list page

1. From the console sidebar, click **Jobs**.
2. To view the details of a job, click its **Job ID**.

The **Job ID** is a link only if you have the correct permissions to run the related job.

3. To view the profile of the user who ran the job, click the user name link.

A UUID appears for user records that have been deleted.

Related information

[User permissions](#) on page 230

The table below lists the available object types and their permissions, as well as an explanation of the permission and how to use it.

[Using Bolt with orchestrator](#) on page 429

Bolt enables running a series of tasks — called *plans* — to help you automate the manual work of maintaining your infrastructure. When you pair Bolt with PE, you get advanced automation with the management and logging capabilities of PE

Job details page

The **Job** details page captures the job start time, duration, user who ran the job, and node run results status: Succeeded, Failed, or Skipped.

View the Job details page

1. Click the **Job ID** on the **Job** list page to view the **Job** details page.
2. To view the profile of the user who ran the job, click the user name link.

A UUID appears for user records that have been deleted.

3. Click the **Job Details** link for more information.
 - Puppet run details include the node target type, environment, concurrency (if set on a job run from the command line), and run mode.
 - Task run details include the node target type and any parameters and values set for the task.
 - Plan run details include plan output and links to the individual task run details.
4. Review the **Node run results** table for individual node run results.

This table is available for Puppet run and task jobs. For nodes that have the Puppet agent installed, you can click a node name link to view the node detail page.

Node run results table for Puppet run jobs

In the **Node run results** table, nodes are sorted by status and then by certname. Use the table to view the number of total resources affected on each node in the job and track the event types for each node.

Event type	Description
Corrective change	Puppet found an inconsistency between the last applied catalog and a property's configuration and corrected the property to match the catalog.
Failure	A property was out of sync. Puppet tried to make changes, but was unsuccessful. To learn more about a failed node, click the node link and use the Reports page to drill down through its events.
Intentional change	Puppet applied catalog changes to a property.

Event type	Description
Skipped resource	A prerequisite for this resource was not met. This prerequisite is either one of the resource's dependencies or a timing limitation set with the schedule metaparameter. The resource might be in sync or out of sync; Puppet doesn't know yet.

Additionally, you can use the **Job node status** filter to filter nodes in this table based on their status. For example, if a job has completed and there were failures, you can filter to view only failed nodes, so you know which nodes to investigate further.

Nodes can have the following statuses:

- Succeeded
- Failed
- Failed - error...
- Skipped
- In progress (Used only for running jobs.)
- Queued

To view a node's run report, click the link in its **Report** column.

To export the node run results to a CSV file, click **Export data**. The CSV includes the same information as the table.

Node run results table for task jobs

The node run results returned by the orchestrator vary depending on the task type. Output is shown in either standard output (stdout) or structured output. For example, a `exec` task that runs the `ls` command on your nodes shows results in stdout output, but a package upgrade task most likely shows results in structured output.

If output is available for a task that completes successfully, the results table includes the number of nodes affected and any parameters specified in the task. Some tasks can successfully complete but have no output. In such cases, the orchestrator returns a `No output for this node` message.

When tasks fail, orchestrator returns a structured error that describes the failure.

Tip: Links to the node details screen are available only for nodes that have the Puppet agent installed.

Plan output and event log

You can find the output and result of plan job runs by navigating to the **Plan** tab in the **Job** page. Click on the **Job ID** for the plan you want to view to see the main job output, plus the event log broken down by step. Since plans include multiple steps, tasks, or scripts, you can view the output and result of each item separately.

Rerunning jobs from the details page

On the Job details page, you can run a job again, using the same settings specified in the original job. You must have the correct permissions to run Puppet or tasks to rerun a job.

To run a job again, click the **Run again** button. This action sets up a new job using the same settings from the original job. The new job recreates the PQL query or node list target and runs on the same nodes.

Jobs with application targets run from the orchestrator command line cannot be rerun from the console, as the console doesn't support these targets. Similarly, tasks run as part of a plan cannot be run from the console (a plan combines multiple tasks and runs them with a single Bolt command).

Additionally, the `whole_environment` target is deprecated, so jobs with that target cannot be rerun from the console or command line interface.

Review jobs from the command line

Use the `puppet job show` command to view running or completed jobs.

Before you begin

The first time you run a command, you need to authenticate. See the [orchestrator installation instructions](#) for information about setting RBAC permissions and token authorization.

If you're running this command from a managed or non-managed workstation, you must specify the full path to the command. For example, `c:\Program Files\Puppet Labs\Client\bin\puppet-task run`.

Install the tasks you want to use.

Make sure you have permission to run the tasks on all nodes.

Make sure you have permission to run the tasks.

Make sure you have access to the nodes you want to target.

Tip: You can add network devices to a node list when you have installed modules for device transports in your production environment. You can find such modules in [Puppet Forge](#).

Make sure you have permissions to run tasks and queries.

Make sure you have permissions to run jobs and queries.

Additionally, you can view a list of jobs or view the details of jobs that have previously run or are in progress from the **Job details** page in the console.

1. Log into your Puppet master or client tools workstation.

2. Run one of the following commands:

- To check the status of a running or completed job, run: `puppet job show <job ID>`

The command returns the following:

- The status of the job (running, finished, or failed).
- The job type: Puppet run, task, or plan task (a plan combines multiple tasks and runs them with a single Bolt command).
- The job description, if set.
- The start and finish time.
- The elapsed time or duration of the run.
- The user who ran the job.
- The environment the job ran in, if you set the environment.
- Whether no-op mode was enforced.
- The number of nodes in the job.
- The target specified for the job.
- To view a list of running and completed jobs, up to 50 maximum (the concurrency limit), ordered by timestamp, run: `puppet job show`

Viewing jobs triggered without the `puppet job` command

When you run any orchestrator jobs through the console or the orchestrator API, the `puppet job show` command also prints those jobs.

You can use the `puppet job show` command with or without a job ID.

```
puppet job show
```

ID	STATUS	JOB TYPE	TIMESTAMP	TARGET
3435	failed	echo	04/04/2018 06:07:18 PM	
192-168-2-171.rfc...	3434	finished	puppet run	
04/04/2018 05:57:27 PM	911bcc40-4550-44e...			

```

3433 finished package                                04/04/2018 05:53:14 PM
911bcc40-4550-44e...
3432 finished echo                                    04/04/2018 05:38:36 PM
bronze-10,bronze-...

```

*environment not enforced on a per node basis

The `enforce_environment` option

The `enforce_environment` option is an option on the `/command/` API endpoint. By default `enforce_environment` is set to `true`, which means agent nodes are forced to run in the same environment in which their configured applications are defined. If set to `false`, agent nodes run in whatever environment they are classified in or assigned to. API jobs might not have an environment specified. If a job has no environment specified, agent node run ordering **is not** specified by applications.

Puppet orchestrator API v1 endpoints

Use this API to gather details about the orchestrator jobs you run.

Related information

[API index](#) on page 33

APIs allow you to interact with Puppet and Puppet Enterprise (PE) applications from your own code or application integration hooks.

Puppet orchestrator API: forming requests

Instructions on interacting with this API.

By default, the orchestrator service listens on port 8143, and all endpoints are relative to the `/orchestrator/v1` path. So, for example, the full URL for the `jobs` endpoint on localhost would be `https://localhost:8143/orchestrator/v1/jobs`.

Tip: The orchestrator API accepts well-formed HTTP(S) requests.

Authenticating to the orchestrator API with a token

You need to authenticate requests to the orchestrators's API. You can do this using user authentication tokens.

For detailed information about authentication tokens, see [token-based authentication](#). The following example shows how to use a token in an API request.

To use the `jobs` endpoint of the orchestrator API to get a list of all jobs that exist in the orchestrator, along with their associated metadata, you'd first generate a token with the `puppet-access` tool. You'd then copy that token and replace `<TOKEN>` with that string in the following request:

```
curl -k -X GET https://<HOSTNAME>:<PORT>/orchestrator/v1/jobs -H 'X-
Authentication:<TOKEN>'
```

Example token usage: deploy an environment

If you want to deploy an environment with the orchestrator's API, you can form a request with the token you generated earlier. For example:

```
curl -k -H 'X-Authentication:<TOKEN>' https://<HOSTNAME>:<PORT>/
orchestrator/v1/command/deploy -X POST -d '{"environment":"production"}' -H
"Content-Type: application/json"
```

This returns a JSON structure that includes a link to the new job started by the orchestrator.

```
{
  "job" : {
    "id" : "https://orchestrator.vm:8143/orchestrator/v1/jobs/81",
    "name" : "81"
  }
}
```

You can make an additional request to get more information about the job. For example:

```
curl -k -X GET https://<HOSTNAME>:<PORT>/orchestrator/v1/jobs/81 -H 'X-
Authentication:<TOKEN>'
```

Related information

[Token-based authentication](#) on page 242

Puppet Enterprise users generate tokens to authenticate their access to certain PE command-line tools and API endpoints. Authentication tokens are tied to the permissions granted to the user through RBAC, and provide the user with the appropriate access to HTTP requests.

Puppet orchestrator API: command endpoint

Use the `/command` endpoint to start and stop orchestrator jobs for tasks and plans.

POST /command/deploy

Run the orchestrator across all nodes in an environment.

Request format

The request body must be a JSON object using these keys:

Key	Definition
environment	The environment to deploy. This key is required.
scope	Object, required unless <code>target</code> is specified. The PuppetDB query, a list of nodes, a classifier node group id, or an application/application instance to deploy.
description	String, a description of the job.
noop	Boolean, whether to run the agent in no-op mode. The default is <code>false</code> .
no_noop	Boolean, whether to run the agent in enforcement mode. Defaults to <code>false</code> . This flag overrides <code>noop = true</code> if set in the agent's <code>puppet.conf</code> , and cannot be set to true at the same time as the <code>noop</code> flag.
concurrency	Integer, the maximum number of nodes to run at one time. The default, if unspecified, is unlimited.
enforce_environment	Boolean, whether to force agents to run in the same environment in which their assigned applications are defined. This key is required to be <code>false</code> if <code>environment</code> is an empty string
debug	Boolean, whether to use the <code>--debug</code> flag on Puppet agent runs.
trace	Boolean, whether to use the <code>--trace</code> flag on Puppet agent runs.

Key	Definition
<code>evaltrace</code>	Boolean, whether to use the <code>--evaltrace</code> flag on Puppet agent runs.
<code>target</code>	String, required unless <code>scope</code> is specified. The name of an application or application instance to deploy. If an application is specified, all instances of that application are deployed. If this key is left blank or unspecified without a scope, the entire environment is deployed. This key is deprecated.

For example, to deploy the `node1.example.com` environment in no-op mode, the following request is valid:

```
{
  "environment" : "",
  "noop" : true,
  "scope" : {
    "nodes" : [ "node1.example.com" ]
  }
}
```

Scope

Scope is a JSON object containing exactly one of these keys:

Key	Definition
<code>application</code>	The name of an application or application instance to deploy. If an application type is specified, all instances of that application are deployed.
<code>nodes</code>	A list of node names to target.
<code>query</code>	A PuppetDB or PQL query to use to discover nodes. The target is built from <code>certname</code> values collected at the top level of the query.
<code>node_group</code>	A classifier node group ID. The ID must correspond to a node group that has defined rules. It is not sufficient for parent groups of the node group in question to define rules. The user must also have permissions to view the node group.

To deploy an application instance in the `production` environment:

```
{
  "environment" : "production",
  "scope" : {
    "application" : "Wordpress_app[demo]"
  }
}
```

To deploy a list of nodes:

```
{
  "environment" : "production",
  "scope" : {
    "nodes" : [ "node1.example.com", "node2.example.com" ]
  }
}
```

```
}
```

To deploy a list of nodes with the `certname` value matching a regex:

```
{
  "environment" : "production",
  "scope" : {
    "query" : ["from", "nodes", ["~", "certname", ".*"]]
  }
}
```

To deploy to the nodes defined by the "All Nodes" node group:

```
{
  "environment" : "production",
  "scope" : {
    "node_group" : "000000000-0000-4000-8000-0000000000000"
  }
}
```

Response format

If all node runs succeed, and the environment is successfully deployed, the server returns a 202 response.

The response is a JSON object containing a link to retrieve information about the status of the job and uses any one of these keys:

Key	Definition
<code>id</code>	An absolute URL that links to the newly created job.
<code>name</code>	The name of the newly created job.

For example:

```
{
  "job" : {
    "id" : "https://orchestrator.example.com:8143/orchestrator/v1/jobs/1234"
    "name" : "1234"
  }
}
```

Error responses

For this endpoint, the `kind` key of the error displays the conflict.

Key	Definition
<code>puppetlabs.orchestrator/unknown-environment</code>	If the environment does not exist, the server returns a 404 response.
<code>puppetlabs.orchestrator/empty-environment</code>	If the environment requested contains no applications or no nodes, the server returns a 400 response.
<code>puppetlabs.orchestrator/empty-target</code>	If the application instance specified to deploy does not exist or is empty, the server returns a 400 response.
<code>puppetlabs.orchestrator/dependency-cycle</code>	If the application code contains a cycle, the server returns a 400 response.
<code>puppetlabs.orchestrator/puppetdb-error</code>	If the orchestrator is unable to make a query to PuppetDB, the server returns a 400 response.

Key	Definition
<code>puppetlabs.orchestrator/query-error</code>	If a user does not have appropriate permissions to run a query, or if the query is invalid, the server returns a 400 response.

Related information

[Puppet orchestrator API: forming requests](#) on page 509

Instructions on interacting with this API.

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

POST /command/stop

Stop an orchestrator job that is currently in progress. Puppet agent runs that are in progress finish, but no new agent runs start. While agents are finishing, the server continues to produce events for the job.

The job transitions to status `stopped` when all pending agent runs have finished.

This command is *idempotent*: it can be issued against the same job any number of times.

Request format

The JSON body of the request must contain the ID of the job to stop. The job ID is the same value as the name property returned with the `deploy` command.

- `job`: the name of the job to stop.

For example:

```
{
  "job": "1234"
}
```

Response format

If the job is stopped successfully, the server returns a 202 response. The response uses these keys:

Key	Definition
<code>id</code>	An absolute URL that links to the newly created job.
<code>name</code>	The name of the newly created job.
<code>nodes</code>	A hash that shows all of the possible node statuses, and how many nodes are currently in that status.

For example:

```
{
  "job": {
    "id": "https://orchestrator.example.com:8143/orchestrator/v1/jobs/1234",
    "name": "1234",
    "nodes": {
      "new": 5,
      "running": 8,
      "failed": 3,
      "errored": 1,
      "skipped": 2,
      "finished": 5
    }
  }
}
```

```

    }
  }
}

```

Error responses

For this endpoint, the `kind` key of the error displays the conflict.

Key	Definition
<code>puppetlabs.orchestrator/validation-error</code>	If a job name is not valid, the server returns a 400 response.
<code>puppetlabs.orchestrator/unknown-job</code>	If a job name is unknown, the server returns a 404 response.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

POST /command/task

Run a permitted task job across a set of nodes.

Request format

The request body must be a JSON object and uses the following keys:

Key	Definition
<code>environment</code>	The environment to load the task from. The default is <code>production</code> .
<code>scope</code>	The PuppetDB query, list of nodes, or a node group ID. Application scopes are not allowed for task jobs. This key is required.
<code>description</code>	A description of the job.
<code>noop</code>	Whether to run the job in no-op mode. The default is <code>false</code> .
<code>task</code>	The task to run on the targets. This key is required.
<code>params</code>	The parameters to pass to the task. This key is required, but can be an empty object.

For example, to run the `package` task on `node1.example.com`, the following request is valid:

```

{
  "environment" : "test-env-1",
  "task" : "package",
  "params" : {
    "action" : "install",
    "name" : "httpd"
  },
  "scope" : {
    "nodes" : [ "node1.example.com" ]
  }
}

```

Scope

Scope is a JSON object containing exactly one of the following keys:

Key	Definition
<code>nodes</code>	An array of node names to target.
<code>query</code>	A PuppetDB or PQL query to use to discover nodes. The target is built from the <code>certname</code> values collected at the top level of the query.
<code>node_group</code>	A classifier node group ID. The ID must correspond to a node group that has defined rules. It is not sufficient for parent groups of the node group in question to define rules. The user must also have permissions to view the node group. Any nodes specified in the scope that the user does not have permissions to run the task on are excluded.

To deploy an application instance in the production environment, the following request is valid:

```
{
  "environment" : "production",
  "scope" : {
    "application" : "Wordpress_app[demo]"
  }
}
```

To run a task on a list of nodes, the following request is valid:

```
{
  "environment" : "production",
  "scope" : {
    "nodes" : ["node1.example.com", "node2.example.com"]
  }
}
```

To run a task on a list of nodes with the `certname` value matching a regex, the following request is valid:

```
{
  "environment" : "production",
  "scope" : {
    "query" : ["from", "nodes", ["~", "certname", ".*"]]
  }
}
```

To deploy to the nodes defined by the "All Nodes" node group the following request is valid:

```
{
  "environment" : "production",
  "scope" : {
    "node_group" : "000000000-0000-4000-8000-000000000000"
  }
}
```

Response format

If the task starts successfully, the response will have a 202 status.

The response will be a JSON object containing a link to retrieve information about the status of the job. The keys of this object are:

- `id`: an absolute URL that links to the newly created job.
- `name`: the name of the newly created job. For example:

```
{
  "job" : {
    "id" : "https://orchestrator.example.com:8143/orchestrator/v1/
jobs/1234",
    "name" : "1234"
  }
}
```

-

Error responses

For this endpoint, the `kind` key of the error displays the conflict.

Key	Definition
<code>puppetlabs.orchestrator/unknown-environment</code>	If the environment does not exist, the server returns a 404 response.
<code>puppetlabs.orchestrator/empty-target</code>	If the application instance specified to deploy does not exist or is empty, the server returns a 400 response.
<code>puppetlabs.orchestrator/puppetdb-error</code>	If the orchestrator is unable to make a query to PuppetDB, the server returns a 400 response.
<code>puppetlabs.orchestrator/query-error</code>	If a user does not have appropriate permissions to run a query, or if the query is invalid, the server returns a 400 response.
<code>puppetlabs.orchestrator/not-permitted</code>	This error occurs when a user does not have permission to run the task on the requested nodes. Server returns a 403 response.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

POST /command/schedule_task

Schedule a task to run at a future date and time.

Request format

The request body must be a JSON object and uses the following keys:

Key	Definition
<code>task</code>	The task to run on the targets. Required.
<code>params</code>	The parameters to pass to the task. Required.
<code>scope</code>	The PuppetDB query, list of nodes, or a node group ID. Application scopes are not allowed for task jobs. Required.

Key	Definition
<code>scheduled_time</code>	The ISO-8601 timestamp the determines when to run the scheduled job. If timestamp is in the past, a 400 error is thrown. Required.
<code>description</code>	A description of the job.
<code>environment</code>	The environment to load the task from. The default is <code>production</code> .
<code>noop</code>	Whether to run the job in no-op mode. The default is <code>false</code> .

For example, to run the `package` task on `node1.example.com`, the following request is valid:

```
{
  "environment" : "test-env-1",
  "task" : "package",
  "params" : {
    "action" : "install",
    "package" : "httpd"
  },
  "scope" : {
    "nodes" : ["node1.example.com"]
  },
  "scheduled_time" : "2027-05-05T19:50:08Z"
}
```

Response format

If the task schedules successfully the server returns 202.

The response is a JSON object containing a link to retrieve information about the status of the job. The keys of this object are

- `id`: an absolute URL that links to the newly created job.
- `name`: the name of the newly created job.

For example:

```
{
  "scheduled_job" : {
    "id" : "https://orchestrator.example.com:8143/orchestrator/v1/scheduled_jobs/2",
    "name" : "1234"
  }
}
```

Error responses

For this endpoint, the `kind` key of the error displays the conflict.

Key	Definition
<code>puppetlabs.orchestrator/invalid-time</code>	If the <code>scheduled_time</code> provided is in the past, the server returns a 400 response.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

POST /command/plan_run

Run a plan via the plan executor.

Request format

The request body must be a JSON object. The following keys are available:

- `plan_name`: String, the name of the plan to run.
- `params`: Hash, the parameters the job will use.
- `environment`: String, environment to load the plan from. The default is `production`.
- `description`: String, description of the job.

To start the canary plan, the following request is valid:

```
{
  "plan_name" : "canary",
  "description" : "Start the canary plan on node1 and node2",
  "params" : {
    "nodes" : [ "node1.example.com", "node2.example.com" ],
    "command" : "whoami",
    "canary" : 1
  }
}
```

Response format

If the plan starts successfully, the response will have status 202.

The response will be a JSON object containing the generated plan job name.

For example:

```
{
  "name" : "1234"
}
```

Error responses

See the [error response documentation](#) for the general format of error response. For this endpoint, the `kind` key of the error displays the conflict.

- `puppetlabs.orchestrator/validation-error`: if the plan name is not valid the server returns a 400 response.
- `puppetlabs.orchestrator/not-permitted`: this error occurs if a user makes a request they lack permissions for. The server returns a 403 response.

Puppet orchestrator API: events endpoint

Use the `/events` endpoint to learn about events that occurred during an orchestrator job.

GET /jobs/:job-id/events

Retrieve all of the events that occurred during a given job.

Parameters

The request accepts this query parameter:

Parameter	Definition
start	Start the list of events with the <i>nth</i> event.

For example:

```
https://orchestrator.example.com:8143/orchestrator/v1/jobs/352/events?
start=1272
```

Response format

The response is a JSON object that details the events in a job, and uses these keys:

Key	Definition
next-events	A link to the next event in the job.
items	A list of all events related to the job.
id	The job ID.
type	The current status of the event. See event-types.
timestamp	The time when the event was created.
details	Information about the event.
message	A message about the given event.

For example:

```
{
  "next-events" : {
    "id" : "https://orchestrator.example.com:8143/orchestrator/v1/jobs/352/
events?start=1272"
  },
  "items" : [ {
    "id" : "1267",
    "type" : "node_running",
    "timestamp" : "2016-05-05T19:50:08Z",
    "details" : {
      "node" : "puppet-agent.example.com",
      "detail" : {
        "noop" : false
      }
    }
  },
  "message" : "Started puppet run on puppet-agent.example.com ..."
}]
}
```

Event types

The response format for each event contains one of these event types, which is determined by the status of the event.

Event type	Definition
node_errored	Created when there was an error running Puppet on a node.
node_failed	Created when Puppet failed to run on a node.
node_finished	Created when puppet ran successfully on a node.

Event type	Definition
node_running	Created when Puppet starts running on a node.
node_skipped	Created when a Puppet run is skipped on a node (for example, if a dependency fails).
job_aborted	Created when a job is aborted without completing.

Error responses

For this endpoint, the `kind` key of the error displays the conflict.

Key	Definition
<code>puppetlabs.orchestrator/validation-error</code>	If the <code>start</code> parameter or the <code>job-id</code> in the request are not integers, the server returns a 400 response.
<code>puppetlabs.orchestrator/unknown-job</code>	If the job does not exist, the server returns a 404 response.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

GET plan_job/:job-id/events

Retrieve all of the events that occurred during a given plan job.

Parameters

The request accepts the following query parameter:

- `start`: The lowest database ID of an event that should be shown.

For example:

`https://orchestrator.example.com:8143/orchestrator/v1/plan_jobs/352/events?start=1272`

Response format

The response is a JSON object that details the events in a plan job. The following keys are used:

- `next-events`: a link to the next event in the job.
 - `id`: the url of the next event.
 - `event`: the next event id.
- `items`: a list of all events related to the job.
 - `id`: the id of the event.
 - `type`: the type of event. See `event-types`.
 - `timestamp`: the time when the event was created.
 - `details`: details of the event, differs based on the type of the event. See the potential values section for more detailed information.

For example:

```
{
  "next-events" : {
    "id" : "https://orchestrator.example.com:8143/orchestrator/v1/
plan_jobs/352/events?start=1272",
    "event" : "1272"
  }
}
```



```

    },
    "items" : [ {
      "id" : "1267",
      "type" : "task_start",
      "timestamp" : "2016-05-05T19:50:08Z",
      "details" : {
        "job-id" : "8765"
      }
    },
    {
      "id" : "1268",
      "type" : "plan_finished",
      "timestamp" : "2016-05-05T19:50:14Z",
      "details" : {
        "plan-id" : "1234",
        "result" : {
          "Plan output"
        }
      }
    }
  ]
}

```

Event types

The response format for each event will contain one of the following event types, which is determined by the status of the event.

- `task_start`: created when a task run is started
- `script_start`: created when a script run as part of a plan is started
- `command_start`: created when a command run as part of a plan is started
- `upload_start`: created when a file upload as part of a plan is started
- `wait_start`: created when a `wait_until_available()` call as part of a plan is started
- `out_message`: created when `out::message` is called as part of a plan. `details` will contain a message key with a value containing the message sent with `out::message`
- `plan_finished`: created when a plan job successfully finishes
- `plan_failed`: created when a plan job fails

"Sub" plans

If a plan contains the `run_plan()` function it will begin execution of another "sub" plan during the execution of the plan job. "Sub" plans will not receive their own plan job, they will execute as part of the original plan job. There are specific events for "sub" plans that indicate a plan within a plan has started or finished:

- `plan_start`: created when a new plan was kicked off from the current plan job using the `run_plan()` function
- `plan_end`: created when a plan started using `run_plan()` has finished

Note: `plan_end` indicates the end of a "sub" plan (i.e. the plan job has not finished yet, but the "sub" plan has), while `plan_finished` indicates the end of a plan job.

Potential values in `details`

task and plan actions

Includes the `job-id` if the event is a plan action or task (i.e. `task_start`, `command_start`, etc).

```

{
  "type" : "task_start",
  "timestamp" : "2019-09-30T22:22:32Z",
  "details" : {

```

```

      "job-id" : 69
    },
    "id" : "80"
  }

```

plan_finished or plan_failed

If the event type is plan_finished or plan_failed, details will include plan-id and result.

```

{
  "type" : "plan_finished",
  "timestamp" : "2019-09-30T22:22:33Z",
  "details" : {
    "result" : "plan result",
    "plan-id" : "9"
  },
  "id" : "81"
}

```

out_message

If the event type is out_message, details will include message.

```

{
  "type" : "out_message",
  "timestamp" : "2019-09-30T22:22:32Z",
  "details" : {
    "message" : "message sent from a plan"
  }
}

```

plan_start or plan_end

If the event type is plan_start or plan_end, details will include the plan that ran (plan) and plan_end will include duration.

plan_start:

```

{
  "type" : "plan_start",
  "timestamp" : "2019-09-30T22:22:31Z",
  "details" : {
    "plan" : "test::sub_plan"
  },
  "id" : "76"
}

```

plan_end:

```

{
  "type" : "plan_end",
  "timestamp" : "2019-09-30T22:22:32Z",
  "details" : {
    "plan" : "test::sub_plan",
    "duration" : 0.647551
  }
}

```

Error responses

See the [error response documentation](#) for the general format of error responses. For the endpoint, the kind key of the error displays the conflict.

- `puppetlabs.orchestrator/validation-error`: if the `start` parameter or the `job-id` in the request are not integers, the server returns a 400 response.
- `puppetlabs.orchestrator/unknown-job`: if the plan job does not exist, the server returns a 404 response.

Puppet orchestrator API: inventory endpoint

Use the `/inventory` endpoint to discover which nodes can be reached by the orchestrator.

GET /inventory

List all nodes that are connected to the PCP broker.

Response format

The response is a JSON object containing a list of records for connected nodes, using these keys:

Key	Definition
<code>name</code>	The name of the connected node.
<code>connected</code>	The connection status is either <code>true</code> or <code>false</code> .
<code>broker</code>	The PCP broker the node is connected to.
<code>timestamp</code>	The time when the connection was made.

For example:

```
{
  "items" : [
    {
      "name" : "foo.example.com",
      "connected" : true,
      "broker" : "pcp://broker1.example.com/server",
      "timestamp": "2016-010-22T13:36:41.449Z"
    },
    {
      "name" : "bar.example.com",
      "connected" : true,
      "broker" : "pcp://broker2.example.com/server",
      "timestamp" : "2016-010-22T13:39:16.377Z"
    }
  ]
}
```

Error responses

For this endpoint, the `kind` key of the error displays the conflict.

The server returns a 500 response if the PCP broker can't be reached.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

GET /inventory/:node

Return information about whether the requested node is connected to the PCP broker.

Response format

The response is a JSON object indicating whether the queried node is connected, using these keys:

Key	Definition
name	The name of the connected node.
connected	The connection status is either <code>true</code> or <code>false</code> .
broker	The PCP broker the node is connected to.
timestamp	The time when the connection was made.

For example:

```
{
  "name" : "foo.example.com",
  "connected" : true,
  "broker" : "pcp://broker.example.com/server",
  "timestamp" : "2017-03-29T21:48:09.633Z"
}
```

Error responses

For this endpoint, the `kind` key of the error displays the conflict.

The server returns a 500 response if the PCP broker can't be reached.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

POST /inventory

Check if the given list of nodes is connected to the PCP broker.

Request format

The request body is a JSON object specifying a list of nodes to check. For example:

```
{
  "nodes" : [
    "foo.example.com",
    "bar.example.com",
    "baz.example.com"
  ]
}
```

Response format

The response is a JSON object with a record for each node in the request, using these keys:

Key	Definition
name	The name of the connected node.
connected	The connection status is either <code>true</code> or <code>false</code> .
broker	The PCP broker the node is connected to.
timestamp	The time when the connection was made.

For example:

```
{
```

```

"items" : [
  {
    "name" : "foo.example.com",
    "connected" : true,
    "broker" : "pcp://broker.example.com/server",
    "timestamp" : "2017-07-14T15:57:33.640Z"
  },
  {
    "name" : "bar.example.com",
    "connected" : false
  },
  {
    "name" : "baz.example.com",
    "connected" : true,
    "broker" : "pcp://broker.example.com/server",
    "timestamp" : "2017-07-14T15:41:19.242Z"
  }
]
}

```

Error responses

For this endpoint, the `kind` key of the error displays the conflict.

The server returns a 500 response if the PCP broker can't be reached.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

Puppet orchestrator API: jobs endpoint

Use the `/jobs` endpoint to examine orchestrator jobs and their details.

GET /jobs

List all of the jobs known to the orchestrator.

Parameters

The request accepts the following query parameters:

Parameter	Definition
<code>limit</code>	Return only the most recent <i>n</i> number of jobs.
<code>offset</code>	Return results offset <i>n</i> records into the result set.
<code>order_by</code>	Return results ordered by a column. Ordering is available on <code>owner</code> , <code>timestamp</code> , <code>environment</code> , <code>name</code> , and <code>state</code> . Orderings requested on <code>owner</code> are applied to the <code>login</code> subfield of <code>owner</code> .
<code>order</code>	Indicates whether results are returned in ascending or descending order. Valid values are <code>"asc"</code> and <code>"desc"</code> . Defaults to <code>"asc"</code>

For example:

```

https://orchestrator.example.com:8143/orchestrator/v1/jobs/352/jobs?
limit=20&offset=20

```

Response format

The response is a JSON object that lists orchestrator jobs and associated details, and uses these keys:

Key	Definition
items	Contains an array of all the known jobs.
id	An absolute URL to the given job.
name	The name of the given job.
state	The current state of the job: <code>new</code> , <code>ready</code> , <code>running</code> , <code>stopping</code> , <code>stopped</code> , <code>finished</code> , or <code>failed</code> .
command	The command that created that job.
options	All of the options used to create that job. The schema of options might vary based on the command.
owner	The subject id and login for the user that requested the job.
description	(deprecated) A user-provided description of the job. For future compatibility, use the description in <code>options</code> .
timestamp	The time when the job state last changed.
environment	(deprecated) The environment that the job operates in.
node_count	The number of nodes the job runs on.
node_states	A JSON map containing the counts of nodes involved with the job by current node state. Unrepresented states are not displayed. This field is <code>null</code> when no nodes exist for a job.
nodes	A link to get more information about the nodes participating in a given job.
report	A link to the report for a given job.
events	A link to the events for a given job.

For example:

```
{
  "items" : [
    {
      "id" : "https://orchestrator.example.com:8143/orchestrator/v1/
jobs/1234",
      "name": "1234",
      "state" : "finished",
      "command" : "deploy",
      "node_count" : 5,
      "node_states" : {
        "finished": 2,
        "errored": 1,
        "failed": 1,
        "running": 1
      },
      "options" : {
        "concurrency" : null,
        "noop" : false,
        "trace" : false,
```

```

      "debug" : false,
      "scope" : { },
      "enforce_environment" : true,
      "environment" : "production",
      "evaltrace" : false,
      "target" : null,
      "description" : "deploy the web app",
    },
    "owner" : {
      "id" : "751a8f7e-b53a-4ccd-9f4f-e93db6aa38ec",
      "login" : "brian"
    },
    "description" : "deploy the web app",
    "timestamp" : "2016-05-20T16:45:31Z",
    "environment" : { "name" : "production" },
    "report" : {
      "id" : "https://localhost:8143/orchestrator/v1/jobs/375/report"
    },
    "events" : {
      "id" : "https://localhost:8143/orchestrator/v1/jobs/375/events"
    },
    "nodes" : {
      "id" : "https://localhost:8143/orchestrator/v1/jobs/375/nodes"
    }
  },
  ...
],
"pagination": {
  "limit": 20,
  "offset": 0,
  "total": 42
}
}
...

```

Error responses

For this endpoint, the `kind` key of the error displays the conflict.

Key	Definition
<code>puppetlabs.orchestrator/validation-error</code>	If the <code>limit</code> parameter is not an integer, the server returns a 400 response.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

GET /jobs/:job-id

List all details of a given job.

Response format

The response is a JSON object that lists all details of a given job, and uses these keys:

Key	Definition
<code>id</code>	An absolute URL to the given job.
<code>name</code>	The name of the given job.

Key	Definition
state	The current state of the job: new, ready, running, stopping, stopped, finished, or failed.
command	The command that created that job.
options	All of the options used to create that job.
owner	The subject id and login for the user that requested the job.
description	(deprecated) A user-provided description of the job.
timestamp	The time when the job state last changed.
environment	The environment that the job operates in.
node_count	The number of nodes the job runs on.
nodes	A link to get more information about the nodes participating in a given job.
report	A link to the report for a given job.
events	A link to the events for a given job.
status	The various enter and exit times for a given job state.

For example:

```
{
  "id" : "https://orchestrator.example.com:8143/orchestrator/v1/jobs/1234",
  "name" : "1234",
  "command" : "deploy",
  "options" : {
    "concurrency" : null,
    "noop" : false,
    "trace" : false,
    "debug" : false,
    "scope" : {
      "application" : "Wordpress_app" },
    "enforce_environment" : true,
    "environment" : "production",
    "evaltrace" : false,
    "target" : null
  },
  "node_count" : 5,
  "owner" : {
    "id" : "751a8f7e-b53a-4ccd-9f4f-e93db6aa38ec",
    "login" : "admin"
  },
  "description" : "deploy the web app",
  "timestamp" : "2016-05-20T16:45:31Z",
  "environment" : {
    "name" : "production"
  },
  "status" : [ {
    "state" : "new",
    "enter_time" : "2016-04-11T18:44:31Z",
    "exit_time" : "2016-04-11T18:44:31Z"
  }, {
    "state" : "ready",
    "enter_time" : "2016-04-11T18:44:31Z",
    "exit_time" : "2016-04-11T18:44:31Z"
  } ]
}
```



```

    }, {
      "state" : "running",
      "enter_time" : "2016-04-11T18:44:31Z",
      "exit_time" : "2016-04-11T18:45:31Z"
    }, {
      "state" : "finished",
      "enter_time" : "2016-04-11T18:45:31Z",
      "exit_time" : null
    } ],
    "nodes" : { "id" : "https://orchestrator.example.com:8143/orchestrator/v1/jobs/1234/nodes" },
    "report" : { "id" : "https://orchestrator.example.com:8143/orchestrator/v1/jobs/1234/report" }
  }
}

```

Error responses

For this endpoint, the kind key of the error displays the conflict.

Key	Definition
puppetlabs.orchestrator/validation-error	If the job-id in the request is not an integer, the server returns a 400 response.
puppetlabs.orchestrator/unknown-job	If the job does not exist, the server returns a 404 response.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

GET /jobs/:job-id/nodes

List all of the nodes associated with a given job.

Response format

The response is a JSON object that details the nodes associated with a job.

`next-events` is an object containing information about where to find events about the job. It has the following keys:

Key	Definition
id	The url of the next event.
event	The next event id.

`items`: a list of all the nodes associated with the job, where each node has the following keys:

Key	Definition
timestamp	(deprecated) The time of the most recent activity on the node. Prefer <code>start_timestamp</code> and <code>finish_timestamp</code> .
start_timestamp	The time the node starting running or <code>nil</code> if the node hasn't started running or was skipped.

Key	Definition
finish_timestamp	The time the node finished running or nil if the node hasn't finished running or was skipped.
duration	The duration of the puppet run in seconds if the node has finished running, the duration in seconds that has passed after the node started running if it is currently running, or nil if the node hasn't started running or was skipped
state	The current state of the job.
transaction_uuid	(deprecated) The ID used to identify the nodes last report.
name	The hostname of the node.
details	information about the last event and state of a given node.
result	The result of a successful, failed, errored node-run. The schema of this varies.

For example:

```
{
  "next-events": {
    "id": "https://orchestrator.example.com:8143/orchestrator/v1/jobs/3/
events?start=10",
    "event": "10"
  },
  "items" : [ {
    "timestamp" : "2015-07-13T20:37:01Z",
    "start_timestamp" : "2015-07-13T20:36:13Z",
    "finish_timestamp" : "2015-07-13T20:37:01Z",
    "duration" : 48.0,
    "state" : "state",
    "transaction_uuid" : :uuid,
    "name" : "wss6c3w9wngpycg.example.com",
    "details" : {
      "message": "Message of latest event"
    },
    "result": {
      "output_1": "success",
      "output_2": [1, 1, 2, 3,]
    }
  }, {
    ...
  } ]
}
```

Results

The result field is available after a node finishes, fails, or errors and contains the contents of the details of the corresponding event. In task jobs this is the result of executing the task. For puppet jobs it contains metrics from the puppet run.

For example:

An error when running a task:

```
"result" : {
```

```

    "msg" : "Running tasks is not supported for agents older than version
5.1.0",
    "kind" : "puppetlabs.orchestrator/execution-failure",
    "details" : {
      "node" : "copper-6"
    }
  }
}

```

Raw stdout from a task:

```

"result" : {
  "output" : "test\n"
}

```

Structured output from a task

```

"result" : {
  "status" : "up to date",
  "version" : "5.0.0.201.g879fc5a-1.e17"
}

```

Error output from a task

```

"result" : {
  "error" : "Invalid task name 'package::status'"
}

```

Puppet run results:

```

"result" : {
  "hash" : "d7ec44e176bb4b2e8a816157ebbae23b065b68cc",
  "noop" : {
    "noop" : false,
    "no_noop" : false
  },
  "status" : "unchanged",
  "metrics" : {
    "corrective_change" : 0,
    "out_of_sync" : 0,
    "restarted" : 0,
    "skipped" : 0,
    "total" : 347,
    "changed" : 0,
    "scheduled" : 0,
    "failed_to_restart" : 0,
    "failed" : 0
  },
  "environment" : "production",
  "configuration_version" : "1502024081"
}

```

Details

The details field contains information based on the last event and current state of the node and can be empty. In some cases it can duplicate data from the results key for historical reasons.

If the node state is `finished` or `failed` the details hash can include a message and a report-url. (deprecated) for jobs started with the run command it also duplicates some information from the result.

```

{
  "items" : [ {

```

```

    "timestamp" : "2015-07-13T20:37:01Z",
    "state" : "finished",
    "transaction_uuid" : :uuid,
    "name" : "wss6c3w9wnngpycg.example.com",
    "details" : {
      "report-url" : "https://peconsole.example.com/#/cm/report/
a15bf509dd7c40705e4e1c24d0935e2e8a1591df",
      "message": "Finished puppet run on wss6c3w9wnngpycg.example.com -
Success!"
    }
  }, {
    "result" : {
      "metrics" : {
        "total" : 82,
        "failed" : 0,
        "changed" : 51,
        "skipped" : 0,
        "restarted" : 2,
        "scheduled" : 0,
        "out_of_sync" : 51,
        "failed_to_restart" : 0
      }
    }, {
      ...
    } ]
  }
}

```

If the node state is `skipped` or `errored`, the service includes a `:detail` key that explains why a node is in that state.

```

{
  "items" : [ {
    "timestamp" : "2015-07-13T20:37:01Z",
    "state" : "failed",
    "transaction_uuid" : :uuid,
    "name" : "wss6c3w9wnngpycg.example.com",
    "details" : {
      "message": "Error running puppet on wss6c3w9wnngpycg.example.com:
java.net.Exception: Something went wrong"
    }
  }, {
    ...
  } ]
}

```

If the node state is `running`, the service returns the `run-time` (in seconds).

```

{
  "items" : [ {
    "timestamp" : "2015-07-13T20:37:01Z",
    "state" : "running",
    "transaction_uuid" : :uuid,
    "name" : "wss6c3w9wnngpycg.example.com",
    "details" : {
      "run-time": 30,
      "message": "Started puppet run on wss6c3w9wnngpycg.example.com..."
    }
  }, {
    ...
  } ]
}

```

Error responses

For this endpoint, the `kind` key of the error displays the conflict.

Key	Definition
<code>puppetlabs.orchestrator/validation-error</code>	If the <code>job-id</code> in the request is not an integer, the server returns a 400 response.
<code>puppetlabs.orchestrator/unknown-job</code>	If the job does not exist, the server returns a 404 response.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

GET /jobs/:job-id/report

Returns a report for a given job.

Response format

The response is a JSON object that reports the status of a job, and uses these keys:

Key	Definition
<code>items</code>	An array of all the reports associated with a given job.
<code>node</code>	The hostname of a node.
<code>state</code>	The current state of the job.
<code>timestamp</code>	The time when the job was created.
<code>events</code>	Any events associated with that node during the job.

For example:

```
{
  "items" : [ {
    "node" : "wss6c3w9wngpycg.example.com",
    "state" : "running",
    "timestamp" : "2015-07-13T20:37:01Z",
    "events" : [ ]
  }, {
    ...
  } ]
}
```

Error responses

For this endpoint, the `kind` key of the error displays the conflict.

Key	Definition
<code>puppetlabs.orchestrator/validation-error</code>	If the <code>job-id</code> in the request is not an integer, the server returns a 400 response.
<code>hpuppetlabs.orchestrator/unknown-job</code>	If the job does not exist, the server returns a 404 response.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

Puppet orchestrator API: scheduled jobs endpoint

Use the `/scheduled_jobs` endpoint to gather information about orchestrator jobs scheduled to run.

GET `/scheduled_jobs`

List scheduled jobs in ascending order.

Parameters

The request accepts the following query parameters:

Parameter	Definition
<code>limit</code>	Return only the most recent <i>n</i> number of jobs.
<code>offset</code>	Return results offset <i>n</i> records into the result set.

For example:

```
https://orchestrator.example.com:8143/orchestrator/v1/scheduled_jobs?
limit=20&offset=20
```

Response format

The response is a JSON object that contains a list of the known jobs and information about the pagination.

Key	Definition
<code>items</code>	Contains an array of all the scheduled jobs.
<code>id</code>	An absolute URL to the given job.
<code>name</code>	The ID of the scheduled job
<code>type</code>	The type of scheduled job (currently only <code>task</code>)
<code>task</code>	The name of the task associated with the scheduled job
<code>scope</code>	The specification of the targets for the task.
<code>environment</code>	The environment that the job operates in.
<code>owner</code>	The specification for the user that requested the job.
<code>description</code>	A user-provided description of the job.
<code>scheduled_time</code>	An ISO8601 specification of when the scheduled job runs.
<code>noop</code>	True if the job runs in no-operation mode, false otherwise.
<code>pagination</code>	Contains the information about the limit, offset and total number of items.
<code>limit</code>	A restricted number of items for the request to return.
<code>offset</code>	A number offset from the start of the collection (zero based).

Key	Definition
total	The total number of items in the collection, ignoring limit and offset.

For example:

```
{
  "items": [
    {
      "id": "https://orchestrator.example.com:8143/orchestrator/v1/
scheduled_jobs/10",
      "name": "10",
      "type": "task",
      "scope": {"nodes": ["foo.example.com", "bar.example.com"]},
      "environment": "production",
      "owner": {
        "id": "751a8f7e-b53a-4ccd-9f4f-e93db6aa38ec",
        "login": "fred"
      },
      "description": "front face the nebaclouser",
      "scheduled_time": "2027-05-05T19:50:08.000Z",
      "noop": false,
    },
    {
      "id": "https://orchestrator.example.com:8143/orchestrator/v1/
scheduled_jobs/9",
      "name": "9",
      "type": "task",
      "scope": {"nodes": ["east.example.com", "west.example.com"]},
      "environment": "production",
      "owner": {
        "id": "751a8f7e-b53a-4ccd-9f4f-e93db6aa38ec",
        "login": "fred"
      },
      "description": "rear face the cranfitouser",
      "scheduled_time": "2027-05-05T19:55:08.000Z",
      "noop": false,
    }
  ],
  "pagination": {
    "limit": 2,
    "offset": 5,
    "total": 15
  }
}
```

Error responses

For this endpoint, the kind key of the error displays the conflict.

Key	Definition
puppetlabs.orchestrator/validation-error	If the limit or offset parameter is not an integer, the server returns a 400 response.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

DELETE /scheduled_jobs/:job-id

Delete a scheduled job.

- Response 204
- Response 403

- Body

```
{
  "kind": "puppetlabs.orchestrator/not-permitted",
  "msg": "Not authorized to delete job {id}"
}
```

Puppet orchestrator API: plans endpoint

Use the /plans endpoint to see all known plans in your environments.

GET /plans

List all known plans in a given environment.

Parameters

The request accepts the following query parameters:

Parameter	Definition
environment	Return the plans in a particular environment. Defaults to production.

Response format

The response is a JSON object that lists the known plans and where to find more information about them. It uses the following keys:

Key	Definition
items	Contains an array of all known plans for the specified environment.
environment	A map containing the name key with the environment name and a code_id key indicating the code id the plans are listed from. Note: code_id is always null.

Each item above has the following keys:

Key	Definition
id	An absolute URL to retrieve plan details.
name	The full name of the plan.
permitted	A boolean indicating if the user making the request is permitted to use the plan.

For example:

```
{
  "environment": {
    "name": "production",
    "code_id": null
  },
  "items": [
    {
      "id": "https://orchestrator.example.com:8143/orchestrator/v1/plans/
profile/firewall",
      "name": "profile::firewall",
      "permitted": true
    },
    {
      "id": "https://orchestrator.example.com:8143/orchestrator/v1/plans/
profile/rolling_update",
      "name": "profile::rolling_update",
      "permitted": true
    },
    {
      "id": "https://orchestrator.example.com:8143/orchestrator/v1/plans/
canary/random",
      "name": "canary::random",
      "permitted": false
    }
  ]
}
```

Error responses

For this endpoint, the kind key of the error displays the conflict.

Error	Definition
puppetlabs.orchestrator/validation-error	If the environment parameter is not a legal environment name, the server returns a 400 response.
puppetlabs.orchestrator/unknown-environment	If the specified environment doesn't exist, the server returns a 404 response.

GET /plans/:module/:planname

Return data about the specified plan, including metadata.

Parameters

Parameter	Definition
environment	Return the plan from a particular environment. Defaults to production. Note: code_id is always null.

Response format

The response is a JSON object that includes information about the specified plan. The following keys are used:

Key	Definition
id	An absolute URL to retrieve plan details.

Key	Definition
name	The full name of the plan.
environment	A map containing a name key with the environment name and a code_id key indicating the code id the plan is being listed from.
metadata	Currently always an empty object.
permitted	A boolean indicating if the user is permitted to use the plan or not.

For example:

```
{
  "id": "https://orchestrator.example.com:8143/orchestrator/v1/plans/
package/install",
  "name": "canary::random",
  "environment": {
    "name": "production",
    "code_id": null
  },
  "metadata": {},
  "permitted": true
}
```

Error responses

Error	Definition
puppetlabs.orchestrator/validation-error	If the environment parameter is not a legal environment name, or the module or plan name is invalid, the server returns a 400 response.
puppetlabs.orchestrator/unknown-environment	If the specified environment doesn't exist, the server returns a 404 response.
puppetlabs.orchestrator/unknown-plan	If the specified plan doesn't exist within that environment, the server returns a 404 response.

Puppet orchestrator API: plan jobs endpoint

Use the `/plan_jobs` endpoint to view details about plan jobs you have run.

GET `/plan_jobs`

List the known plan jobs sorted by name and in descending order.

Parameters

The request accepts the following query parameters:

Parameter	Definition
limit	Return only the most recent <i>n</i> number of jobs.
offset	Return results offset <i>n</i> records into the result set.

Response format

The response is a JSON object that contains a list of the known plan jobs, and information about the pagination.

Key	Definition
items	An array of all the plan jobs.
id	An absolute URL to the given plan job.
name	The ID of the plan job.
state	The current state of the plan job: running, success, or failure
options	Information about the plan job: description, plan_name, and any parameters.
description	The user-provided description for the plan job.
plan_name	The name of the plan that was run, for example <code>package::install</code> .
parameters	The parameters passed to the plan for the job.
result	The output from the plan job.
owner	The subject ID and login for the user that requested the job.
created_timestamp	The time the plan job was created.
finished_timestamp	The time the plan job finished.
events	A link to the events for a given plan job.
status	A hash of jobs that ran as part of the plan job, with associated lists of states and their enter and exit times.
pagination	Contains the information about the limit, offset and total number of items.
limit	The number of items the request was limited to.
offset	The offset from the start of the collection (zero based).
total	The total number of items in the collection, ignoring limit and offset.

For example:

```
{
  "items" : [ {
    "finished_timestamp" : null,
    "name" : "37",
    "events" : {
      "id" : "https://localhost:50310/orchestrator/v1/plan_jobs/37/events"
    },
    "state" : "running",
    "id" : "https://localhost:50310/orchestrator/v1/plan_jobs/37",
    "created_timestamp" : "YYYY-MM-DDT20:22:08Z",
    "options" : {
      "description" : "Testing myplan",
      "plan_name" : "myplan",
      "parameters" : {
        "nodes" : [ "localhost" ]
      }
    },
    "owner" : {
      "email" : "",
      "is_revoked" : false,

```

```

      "last_login" : "YYYY-MM-DDT20:22:06.327Z",
      "is_remote" : false,
      "login" : "admin",
      "is_superuser" : true,
      "id" : "42bf351c-f9ec-40af-84ad-e976fec7f4bd",
      "role_ids" : [ 1 ],
      "display_name" : "Administrator",
      "is_group" : false
    },
    "result" : null
  }, {
    "finished_timestamp" : null,
    "name" : "36",
    "events" : {
      "id" : "https://localhost:50310/orchestrator/v1/plan_jobs/36/events"
    },
    "state" : "running",
    "id" : "https://localhost:50310/orchestrator/v1/plan_jobs/36",
    "created_timestamp" : "YYYY-MM-DDT20:22:08Z",
    "options" : {
      "description" : "Testing myplan",
      "plan_name" : "myplan",
      "parameters" : {
        "nodes" : [ "localhost" ]
      }
    },
    "owner" : {
      "email" : "",
      "is_revoked" : false,
      "last_login" : "YYYY-MM-DDT20:22:06.327Z",
      "is_remote" : false,
      "login" : "admin",
      "is_superuser" : true,
      "id" : "42bf351c-f9ec-40af-84ad-e976fec7f4bd",
      "role_ids" : [ 1 ],
      "display_name" : "Administrator",
      "is_group" : false
    },
    "result" : null
  }, {
    "finished_timestamp" : null,
    "name" : "35",
    "events" : {
      "id" : "https://localhost:50310/orchestrator/v1/plan_jobs/35/events"
    },
    "state" : "running",
    "id" : "https://localhost:50310/orchestrator/v1/plan_jobs/35",
    "created_timestamp" : "YYYY-MM-DDT20:22:07Z",
    "options" : {
      "description" : "Testing myplan",
      "plan_name" : "myplan",
      "parameters" : {
        "nodes" : [ "localhost" ]
      }
    },
    "owner" : {
      "email" : "",
      "is_revoked" : false,
      "last_login" : "YYYY-MM-DDT20:22:06.327Z",
      "is_remote" : false,
      "login" : "admin",
      "is_superuser" : true,
      "id" : "42bf351c-f9ec-40af-84ad-e976fec7f4bd",
      "role_ids" : [ 1 ],

```

```

      "display_name" : "Administrator",
      "is_group" : false
    },
    "result" : null
  }, {
    "finished_timestamp" : null,
    "name" : "34",
    "events" : {
      "id" : "https://localhost:50310/orchestrator/v1/plan_jobs/34/events"
    },
    "state" : "running",
    "id" : "https://localhost:50310/orchestrator/v1/plan_jobs/34",
    "created_timestamp" : "YYYY-MM-DDT20:22:07Z",
    "options" : {
      "description" : "Testing myplan",
      "plan_name" : "myplan",
      "parameters" : {
        "nodes" : [ "localhost" ]
      }
    },
    "owner" : {
      "email" : "",
      "is_revoked" : false,
      "last_login" : "YYYY-MM-DDT20:22:06.327Z",
      "is_remote" : false,
      "login" : "admin",
      "is_superuser" : true,
      "id" : "42bf351c-f9ec-40af-84ad-e976fec7f4bd",
      "role_ids" : [ 1 ],
      "display_name" : "Administrator",
      "is_group" : false
    },
    "result" : null
  }, {
    "finished_timestamp" : null,
    "name" : "33",
    "events" : {
      "id" : "https://localhost:50310/orchestrator/v1/plan_jobs/33/events"
    },
    "state" : "running",
    "id" : "https://localhost:50310/orchestrator/v1/plan_jobs/33",
    "created_timestamp" : "YYYY-MM-DDT20:22:07Z",
    "options" : {
      "description" : "Testing myplan",
      "plan_name" : "myplan",
      "parameters" : {
        "nodes" : [ "localhost" ]
      }
    },
    "owner" : {
      "email" : "",
      "is_revoked" : false,
      "last_login" : "YYYY-MM-DDT20:22:06.327Z",
      "is_remote" : false,
      "login" : "admin",
      "is_superuser" : true,
      "id" : "42bf351c-f9ec-40af-84ad-e976fec7f4bd",
      "role_ids" : [ 1 ],
      "display_name" : "Administrator",
      "is_group" : false
    },
    "result" : null
  } ],
  "pagination" : {

```

```

    "limit" : 5,
    "offset" : 3,
    "total" : 40
  }
}

```

GET /plan_jobs/:job-id

List all the details of a given plan job.

Response format

The response is a JSON object that lists all details of a given plan job. The following keys are used:

Key	Definition
id	An absolute URL to the given plan job.
name	The ID of the plan job.
state	The current state of the plan job: running, success, or failure
options	Information about the plan job: description, plan_name, and any parameters.
description	The user-provided description for the plan job.
plan_name	The name of the plan that was run, for example <code>package::install</code> .
parameters	The parameters passed to the plan for the job.
result	The output from the plan job.
owner	The subject ID and login for the user that requested the job.
timestamp	The time when the plan job state last changed.
created_timestamp	The time the plan job was created.
finished_timestamp	The time the plan job finished.
events	A link to the events for a given plan job.
status	A hash of jobs that ran as part of the plan job, with associated lists of states and their enter and exit times.

For example:

```

{
  "id": "https://orchestrator.example.com:8143/orchestrator/v1/
plan_jobs/1234",
  "name": "1234",
  "state": "success",
  "options": {
    "description": "This is a plan run",
    "plan_name": "package::install",
    "parameters": {
      "foo": "bar"
    }
  },
  "result": {
    "output": "test\n"
  },
}

```

```

"owner": {
  "email": "",
  "is_revoked": false,
  "last_login": "YYYY-MM-DDT17:06:48.170Z",
  "is_remote": false,
  "login": "admin",
  "is_superuser": true,
  "id": "42bf351c-f9ec-40af-84ad-e976fec7f4bd",
  "role_ids": [
    1
  ],
  "display_name": "Administrator",
  "is_group": false
},
"timestamp": "YYYY-MM-DDT16:45:31Z",
"status": {
  "1": [
    {
      "state": "running",
      "enter_time": "YYYY-MM-DDT18:44:31Z",
      "exit_time": "YYYY-MM-DDT18:45:31Z"
    },
    {
      "state": "finished",
      "enter_time": "YYYY-MM-DDT18:45:31Z",
      "exit_time": null
    }
  ],
  "2": [
    {
      "state": "running",
      "enter_time": "YYYY-MM-DDT18:44:31Z",
      "exit_time": "YYYY-MM-DDT18:45:31Z"
    },
    {
      "state": "failed",
      "enter_time": "YYYY-MM-DDT18:45:31Z",
      "exit_time": null
    }
  ]
},
"events": {
  "id": "https://localhost:8143/orchestrator/v1/plan_jobs/1234/events"
}
}

```

Error responses

For this endpoint, the kind key of the error displays the conflict.

- `puppetlabs.orchestrator/validation-error`: if the `job-id` in the request is not an integer, the server returns a 400 response.
- `puppetlabs.orchestrator/unknown-job`: if the plan job does not exist, the server returns a 404 response.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

Puppet orchestrator API: tasks endpoint

Use the `/tasks` endpoint to view details about the tasks pre-installed by PE and those you've installed.

GET /tasks

Lists all tasks in a given environment.

Parameters

The request accepts this query parameter:

Parameter	Definition
environment	Returns the tasks in the specified environment. If unspecified, defaults to <code>production</code> .

Response format

The response is a JSON object that lists each known task with a link to additional information, and uses these keys:

Key	Definition
environment	A map containing a <code>name</code> key specifying the environment's name and a <code>code_id</code> key indicating the code ID where the task is listed.
items	Contains an array of all known tasks.
id	An absolute URL where the task's details are listed.
name	The full name of the task.

```
{
  "environment": {
    "name": "production",
    "code_id": "urn:puppet:code-
id:1:a86dal66c30f871823f9b2ea224796e834840676;production"
  },
  "items": [
    {
      "id": "https://orchestrator.example.com:8143/orchestrator/v1/tasks/
package/install",
      "name": "package::install"
    },
    {
      "id": "https://orchestrator.example.com:8143/orchestrator/v1/tasks/
package/upgrade",
      "name": "package::upgrade"
    },
    {
      "id": "https://orchestrator.example.com:8143/orchestrator/v1/tasks/
exec/init",
      "name": "exec"
    }
  ]
}
```


Error responses

For this endpoint, the `kind` key of the error displays the conflict.

Key	Definition
<code>puppetlabs.orchestrator/validation-error</code>	If the <code>environment</code> parameter is not a legal environment name, the server returns a 400 response.
<code>puppetlabs.orchestrator/unknown-environment</code>	If the specified environment doesn't exist, the server returns a 404 response.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

GET /tasks/:module/:taskname

Returns data about a specified task, including metadata and file information. For the default task in a module, `:taskname` is `init`.

Parameters

The request accepts this query parameter:

Parameter	Definition
<code>environment</code>	Returns the tasks in the specified environment. If unspecified, defaults to <code>production</code> .

Response format

The response is a JSON object that includes information about the specified task, and uses these keys:

Key	Definition
<code>id</code>	An absolute URL where the task's details are listed.
<code>name</code>	The full name of the task.
<code>environment</code>	A map containing a <code>name</code> key specifying the environment's name and a <code>code_id</code> key indicating the code ID where the task is listed.
<code>metadata</code>	A map containing the contents of the <code><task>.json</code> file.
<code>files</code>	An array of the files in the task.
<code>filename</code>	The base name of the file.
<code>uri</code>	A map containing <code>path</code> and <code>params</code> fields to construct a URL to download the file content. The client determines which host to download the file from.
<code>sha256</code>	The SHA-256 hash of the file content, in lowercase hexadecimal form.
<code>size_bytes</code>	The size of the file content in bytes.

For example:

```
{
```

```

    "id": "https://orchestrator.example.com:8143/orchestrator/v1/tasks/
package/install",
    "name": "package::install",
    "environment": {
      "name": "production",
      "code_id": "urn:puppet:code-
id:1:a86dal66c30f871823f9b2ea224796e834840676;production"
    },
    "metadata": {
      "description": "Install a package",
      "supports_noop": true,
      "input_method": "stdin",
      "parameters": {
        "name": {
          "description": "The package to install",
          "type": "String[1]"
        },
        "provider": {
          "description": "The provider to use to install the package",
          "type": "Optional[String[1]]"
        },
        "version": {
          "description": "The version of the package to install, defaults to
latest",
          "type": "Optional[String[1]]"
        }
      }
    },
    "files": [
      {
        "filename": "install",
        "uri": {
          "path": "/package/tasks/install",
          "params": {
            "environment": "production"
          }
        },
        "sha256":
"a9089b5b9720dca38a49db6f164cf8a053a7ea528711325dalc23de94672980f",
        "size_bytes": 693
      }
    ]
  }
}

```

Error responses

For this endpoint, the kind key of the error displays the conflict.

Key	Definition
puppetlabs.orchestrator/validation-error	If the environment parameter is not a legal environment name, or the module or taskname is invalid, the server returns a 400 response.
puppetlabs.orchestrator/unknown-environment	If the specified environment doesn't exist, the server returns a 404 response.
puppetlabs.orchestrator/unknown-task	If the specified task doesn't exist within the specified environment, the server returns a 404 response.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

Puppet orchestrator API: root endpoint

Use the `/orchestrator` endpoint to return metadata about the orchestrator API.

GET /orchestrator

A request to the root of the Puppet orchestrator returns metadata about the orchestrator API, along with a list of links to application management resources.

Response format

Responses use the following format:

```
{
  "info" : {
    "title" : "Application Management API (EXPERIMENTAL)",
    "description" : "Multi-purpose API for performing application management
operations",
    "warning" : "This version of the API is experimental, and might change
in backwards-incompatible ways in the future",
    "version" : "0.1",
    "license" : {
      "name" : "Puppet Enterprise License",
      "url" : "https://puppetlabs.com/puppet-enterprise-components-licenses"
    }
  },
  "status" : {
    "name" : "status",
    "id" : "https://orchestrator.example.com:8143/orchestrator/v1/status"
  },
  "collections" : [ {
    "name" : "environments",
    "id" : "https://orchestrator.example.com:8143/orchestrator/v1/
environments"
  }, {
    "name" : "jobs",
    "id" : "https://orchestrator.example.com:8143/orchestrator/v1/jobs"
  } ],
  "commands" : [ {
    "name" : "deploy",
    "id" : "https://orchestrator.example.com:8143/orchestrator/v1/command/
deploy"
  }, {
    "name" : "stop",
    "id" : "https://orchestrator.example.com:8143/orchestrator/v1/command/
stop"
  } ]
}
```

Error responses

For this endpoint, the server returns a 500 response.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

Puppet orchestrator API: usage endpoint

Use the `/usage` endpoint to view details about the active nodes in your deployment.

GET /usage

List the daily node usage of the orchestrator and nodes that are present in PuppetDB.

Parameters

start_date

The first day to include in the query. Format as YYYY-MM-DD.

end_date

The last day to include in the query. This date must be on or after the start date and formatted as YYYY-MM-DD.

Response format

The response is a JSON object indicating the total number of active nodes and subtotals of the nodes with and without an agent installed.

The following keys are used:

items

Contains an array of entries from most recent to least recent.

date

The date in ISO 8601 date format YYYY-MM-DD.

total_nodes

The total number of nodes used on the given day.

nodes_with_agent

The number of active nodes in PuppetDB as of UTC midnight.

nodes_without_agent

The number of unique nodes that don't have an agent that were used since UTC midnight.

pagination

An optional object that includes information about the original query.

start_date

The starting day for the request (if specified).

end_date

The last day requested (if specified).

```
{
  "items": [
    {
      "date": "2018-06-08",
      "total_nodes": 100,
      "nodes_with_agent": 95,
      "nodes_without_agent": 5
    }, {
      "date": "2018-06-07",
      "total_nodes": 100,
      "nodes_with_agent": 95,
      "nodes_without_agent": 5
    }
  ]
}
```

```

    }, {
      "date": "2018-06-06",
      "total_nodes": 100,
      "nodes_with_agent": 95,
      "nodes_without_agent": 5
    }, {
      "date": "2018-06-05",
      "total_nodes": 100,
      "nodes_with_agent": 95,
      "nodes_without_agent": 5
    }
  ],
  "pagination": {
    "start_date": "2018-06-01",
    "end_date": "2018-06-30"
  }
}

```

Error Responses

For this endpoint, the `kind` key of the error displays the conflict.

Related information

[Puppet orchestrator API: error responses](#) on page 549

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

Puppet orchestrator API: error responses

From time to time, you might encounter an error using the orchestrator API. In such cases, you receive an error response.

Every error response from the Puppet orchestrator is a JSON response. Each response is an object that contains the following keys:

Key	Definition
<code>kind</code>	The kind of error encountered.
<code>msg</code>	The message associated with the error.
<code>details</code>	A hash with more information about the error.

For example, if an environment does not exist for a given request, an error is raised similar to the following:

```

{
  "kind" : "puppetlabs.orchestrator/unknown-environment",
  "msg" : "Unknown environment doesnotexist",
  "details" : {
    "environment" : "doesnotexist"
  }
}

```

Managing and deploying Puppet code

Puppet Enterprise (PE) includes built-in tools for managing and deploying your Puppet infrastructure. Code Manager and r10k are code management tools that automatically install modules, create and maintain environments, and deploy new code to your masters, all based on version control of your Puppet code and data.

Code management tools use Git version control to track, maintain, and deploy your Puppet modules, manifests, and data. This allows you to more easily maintain, update, review, and deploy Puppet code and data for multiple environments.

Code Manager automates the deployment of your Puppet code and data. You make code and data changes on your workstation, push changes to your Git repository, and then Code Manager creates environments, installs modules, and deploys the new code to your masters, without interrupting agent runs.

If you are already using r10k to manage your Puppet code, we suggest that you upgrade to Code Manager. Code Manager works in concert with r10k, so when you switch to Code Manager, you no longer interact directly with r10k.

If you're using r10k and aren't ready to switch to Code Manager yet, you can continue using r10k alone. You push your code changes to your version control repo, deploy environments from the command line, and r10k creates environments and installs the modules for each one.

Both Code Manager and the r10k code management tool are built into PE, so you don't have to install anything. To begin managing your code with either of these tools, you perform the following tasks:

1. Create a control repository with Git for maintaining your environments and code.

The control repository is the Git repository that code management uses to maintain and deploy your Puppet code and data and to create environments in your Puppet infrastructure. As you update your control repo, code management keeps each of your environments updated.

2. Set up a Puppetfile to manage content in your environment.

This file specifies which modules and data to install in your environment, including what version of that content to install, and where to download the content from.

3. Configure Code Manager (recommended) or r10k.

Configure code management in the console's master profile. If you need to customize your configuration further, you can do so by adding keys to Hiera.

4. Deploy environments. With Code Manager, either set up a deployment trigger (recommended), or plan to trigger your deployment from the command line, which is useful for initial configuration. If you are using r10k alone, run it from the command line whenever you want to deploy.

The following sections go into detail about setting up and using Code Manager and r10k.

Managing environments with a control repository

To manage your Puppet code and data with either Code Manager or r10k, you need a Git version control repository. This control repository is where code management stores code and data to deploy your environments.

How the control repository works

Code management relies on version control to track, maintain, and deploy your Puppet code and data. The control repository (or repo) is the Git repository that code management uses to manage environments in your infrastructure. As you update code and data in your control repo, code management keeps each of your environments updated.

Code management creates and maintains your environments based on the branches in your control repo. For example, if your control repo has a production branch, a development branch, and a testing branch, code management creates a production environment, a development environment, and a testing environment, each with its own version of your Puppet code and data.

Environments are created in `/etc/puppetlabs/code/environments` on the master. To learn more about environments in Puppet, read the documentation about [environments](#).

To create a control repo that includes the standard recommended structure, code examples, and configuration scripts, base your control repo on the Puppet control repo template. This template covers most customer situations. If you cannot access the internet or cannot use modules directly from the Forge because of your organization's security rules, create an empty control repo and add the files you need to it.

Note: For Windows systems, be sure your version control is configured to use CRLF line endings. See your version control system for instructions on how to do this.

At minimum, a control repo comprises:

- A Git remote repository. The remote is where your control repo is stored on your Git host.
- A default branch named `production`, rather than the usual Git default of `master`.
- A Puppetfile to manage your environment content.
- An `environment.conf` file that modifies the `$modulepath` setting to allow environment-specific modules and settings.



CAUTION: Enabling code management means that Puppet manages the environment directories and **existing environments are not preserved**. Environments with the same name as the new one are overwritten. Environments not represented in the control repo are erased. If you were using environments before, commit any necessary files or code to the appropriate new control repo branch, or back them up somewhere *before* you start configuring code management.

Create a control repo from the Puppet template

To create a control repo that includes a standard recommended structure, code examples, and configuration scripts, base your control repo on the Puppet control repo template. This template covers most customer situations.

To base your control repo on the Puppet control repository template, you copy the control repo template to your development workstation, set your own remote Git repository as the default source, and then push the template contents to that source.

The control repo template contains the files needed to get you started with a functioning control repo, including:

- An `environment.conf` file to implement a `site-modules/` directory for roles, profiles, and custom modules.
- `config_version` scripts to notify you which control repo version was applied to the agents.
- Basic code examples for setting up roles and profiles.
- An example `hieradata` directory that matches the default hierarchy in PE .

1. Generate a private SSH key to allow access to the control repository.

This SSH key cannot require a password.

- a) Generate the key pair:

```
ssh-keygen -t rsa -b 2048 -P '' -f /etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa
```

- b) Set appropriate permissions so that the `pe-puppet` user can access the key:

```
puppet infrastructure configure
```

Your keys are now located in `/etc/puppetlabs/puppetserver/ssh/`:

- Private key: `/etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa`
- Public key: `/etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa.pub`

Configure your `host` to use the SSH public key you generated. The process to do this is different for every `host`. Usually, you create a user or service account, and then assign the SSH public key to it.

Code management needs read access to your control repository, as well as any module repositories referenced in the `.`

See the your `host` docs for detailed instructions on adding SSH keys to your `server`. Commonly used `hosts` include:

- [GitHub](#)
- [BitBucket Server](#)
- [GitLab](#)

2. Create a repository in your `account`, with the name you want your control repo to have.

Steps for creating a repository vary, depending on what `host` you are using (, [GitLab](#), [Bitbucket](#), or another provider). See your `host`'s documentation for complete instructions.

For example, on `:`

- a) Click `+` at the top of the page, and choose **New repository**.
- b) Select the account **Owner** for the repository.
- c) Name the repository (for example, `control-repo`).
- d) Note the repository's SSH URL for later use.

3. From the command line, clone the Puppet control-repo template.

```
git clone https://github.com/puppetlabs/control-repo.git
```

4. Remove the template repository as your default source.

```
git remote remove origin
```

5. Add the control repository you created as the default source.

```
git remote add origin <URL OF YOUR GIT REPOSITORY>
```

You now have a control repository based on the `Puppetcontrol-repo` template. When you make changes to this repo on your workstation, push those changes to the remote copy of the control repo on your Git server, so that code management can deploy your infrastructure changes.

After you've set up your control repo, create a `for` managing your environment content with code management.

If you already have a `,` you can now configure code management. Code management configuration steps differ, depending on whether you're using (recommended) or `.` For important information about the function and limitations of each of these tools, along with configuration instructions, see the `and` pages.

Create an empty control repo

If you can't use the control repo template because, for example, you cannot access the internet or use modules directly from the Forge because of your security rules, create an empty control repo and then add the files you need.

To start with an empty control repo, you create a new repo on your Git host and then copy it to your workstation. You make some changes to your repo, including adding a configuration file that allows code management tools to find modules in both your site and environment-specific module directories. When you're done making changes, push your changes to your repository on your Git host.

1. Generate a private SSH key to allow access to the control repository.

This SSH key cannot require a password.

- a) Generate the key pair:

```
ssh-keygen -t rsa -b 2048 -P '' -f /etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa
```

- b) Set appropriate permissions so that the `pe-puppet` user can access the key:

```
puppet infrastructure configure
```

Your keys are now located in `/etc/puppetlabs/puppetserver/ssh`:

- Private key: `/etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa`
- Public key: `/etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa.pub`

Configure your host to use the SSH public key you generated. The process to do this is different for every host. Usually, you create a user or service account, and then assign the SSH public key to it.

Code management needs read access to your control repository, as well as any module repositories referenced in the `.`

See the your host docs for detailed instructions on adding SSH keys to your server. Commonly used hosts include:

- [GitHub](#)
- [BitBucket Server](#)
- [GitLab](#)

2. Create a repository in your account, with the name you want your control repo to have.

Steps for creating a repository vary, depending on what host you are using (, GitLab, Bitbucket, or another provider). See your host's documentation for complete instructions.

For example, on :

- a) Click + at the top of the page, and choose **New repository**.
 - b) Select the account **Owner** for the repository.
 - c) Name the repository (for example, `control-repo`).
 - d) Note the repository's SSH URL for later use.
3. Clone the empty repository to your workstation by running `git clone <REPOSITORY URL>`
 4. Create a file named `environment.conf` in the main directory of your control repo.
 5. In your text editor, open `environment.conf` file, and add the line below to set the modulepath. Save and close the file.

```
modulepath=site-modules:modules:$basemodulepath
```

6. Add and commit your change to the repository by running:

- a) `git add environment.conf`
- b) `git commit -m "add environment.conf"`

The `environment.conf` file allows code management tools to find modules in both your site and environment-specific module directories. For details about this file, see the [environment.conf](#) documentation.

7. Rename the master branch to production by running `git branch -m master production`

Important: The default branch of a control repo must be production.

8. Push your repository's production branch from your workstation to your host by running `git push -u origin production`

When you make changes to this repo on your workstation, push those changes to the remote copy of the control repo on your Git server, so that code management can deploy your infrastructure changes.

After you've set up your control repo, create a for managing your environment content with code management.

If you already have a , you can now configure code management. Code management configuration steps differ, depending on whether you're using (recommended) or . For important information about the function and limitations of each of these tools, along with configuration instructions, see the and pages.

Add an environment

Create new environments by creating branches based on the production branch of your control repository.

Before you begin

Make sure you have:

- Configured either [Code Manager](#) or [r10k](#).
- Created a [Puppetfile](#) in the default (usually 'production') branch of your control repo.
- Selected your code management deployment method (such as the [puppet-code](#) command or a [webhook](#)).

Remember: If you are using multiple control repos, do not duplicate branch names unless you use a source prefix. For more information about source prefixes, see the documentation about configuring [sources](#).

1. Create a new branch: `git branch <NAME-OF-NEW-BRANCH>`
2. Check out the new branch: `git checkout <NAME-OF-NEW-BRANCH>`
3. Edit the Puppetfile to track the modules and data needed in your new environment, and save your changes.
4. Commit your changes: `git commit -m "a commit message summarizing your change"`
5. Push your changes: `git push origin <NAME-OF-NEW-BRANCH>`
6. Deploy your environments as you normally would, either on the command line or with a Code Manager webhook.

Delete an environment with code management

To delete an environment with Code Manager or r10k, delete the corresponding branch from your control repository.

1. On the production branch of your control repo directory, on the command line, delete the environment's corresponding remote branch by running `git push origin --delete <BRANCH-TO-DELETE>`
2. Delete the local branch by running `git branch -d <BRANCH-TO-DELETE>`
3. Deploy your environments as you normally would, either on the command line or with a Code Manager webhook.

Note: If you use Code Manager to deploy environments with the webhook, deleting a branch from your control repository does not immediately delete that environment from the master's live code directories. Code Manager deletes the environment when it next deploys changes to any other environment. Alternately, to delete the environment immediately, deploy all environments manually, run `puppet-code deploy --all --wait`

Managing environment content with a Puppetfile

A Puppetfile specifies detailed information about each environment's Puppet code and data, including where to get that code and data from, where to install it, and whether to update it.

Both Code Manager and r10k use a Puppetfile to install and manage the content of your environments.

The Puppetfile

The Puppetfile specifies the modules and data that you want in each environment. The Puppetfile can specify what version of modules you want, how the modules and data are to be loaded, and where they are placed in the environment.

A Puppetfile is a formatted text file that specifies the modules and data that you want brought into your control repo. Typically, a Puppetfile controls content such as:

- Modules from the Forge
- Modules from Git repositories
- Data from Git repositories

For each environment that you want to manage content in, you need a Puppetfile. Create a base Puppetfile in your default environment (usually `production`). As you create new branches based on your default branch, each environment inherits this base Puppetfile. You can then edit each environment's Puppetfile as needed.

Managing modules with a Puppetfile

With code management, install and manage your modules only with a Puppetfile.

Almost all Puppet manifests are kept in modules, collections of Puppet code and data with a specific directory structure. By default, Code Manager and r10k install content in a modules directory (`./modules`) in the same directory the Puppetfile is in. For example, declaring the `puppetlabs-apache` module in the Puppetfile normally installs the module into `./modules/apache`. To learn more about modules, see the [module](#) documentation.

Important:

With Code Manager and r10k, **do not** use the `puppet module` command to install or manage modules. Instead, code management depends on the Puppetfile in each of your environments to install, update, and manage your modules. If you've installed modules to the live code directory with `puppet module install`, Code Manager deletes them.

The Puppetfile does NOT include Forge module dependency resolution. You must make sure that you have every module needed for all of your specified modules to run. In addition, Forge module symlinks are unsupported; when you install modules with r10k or Code Manager, symlinks are not installed.

Including your own modules

If you develop your own modules, maintain them in version control and include them in your Puppetfile as you would declare any module from a repository. If you have content in your control repo's module directory that is *not* listed in your Puppetfile, code management purges it. (The control repo module directory is, by default, `./modules` relative to the location of the Puppetfile.)

Deploying code

When you install or update a module, you must trigger Code Manager or r10k to deploy the new or updated code to your environments.

With Code Manager, you can deploy code on the command line or using a webhook:

- [Triggering Code Manager on the command line](#) on page 572
- [Triggering Code Manager with a webhook](#) on page 577

With r10k, you can deploy code using the command line:

- [Deploying environments with r10k](#) on page 603

Creating a Puppetfile

Your Puppetfile manages the content you want to maintain in that environment.

In a Puppetfile, you can declare:

- Modules from the Forge.
- Modules from a Git repository.
- Data or other non-module content (such as Hieradata) from a Git repository.

You can declare any or all of this content as needed for each environment. Each module or repository is specified with a `mod` directive, along with the name of the content and other information the Puppetfile needs so that it can correctly install and update your modules and data.

It's best to create your first Puppetfile in your production branch. Then, as you create branches based on your production branch, edit each branch's Puppetfile as needed.

Create a Puppetfile

Create a Puppetfile that manages the content maintained in your environment.

Before you begin

Set up a control repo, with `production` as the default branch. To learn more about control repositories, see the [control repository](#) documentation.

Create a Puppetfile in your production branch, and then edit it to declare the content in your production environment with the `mod` directive.

1. On your production branch, in the root directory, create a file named `Puppetfile`.
2. In a text editor, for example Visual Studio Code (VS Code), edit the Puppetfile, declaring modules and data content for your environment. Note that Puppet has an [extension](#) for VS Code that supports syntax highlighting of the Puppet language.

You can declare modules from the Forge or you can declare Git repositories in your Puppetfile. See the related topics about declaring content in the Puppetfile for details and code examples.

After creating your Puppetfile, configure Code Manager or r10k.

Change the Puppetfile module installation directory

If needed, you can change the directory to which the Puppetfile installs all modules.

To specify a module installation path other than the default modules directory (`./modules`), use the `moduledir` directive.

This directive applies to *all* content declared in the Puppetfile. You must specify this as a relative path at the top of the Puppetfile, **before** you list any modules.

To change the installation paths for only certain modules or data, declare those content sources as Git repositories and set the `install_path` option. This option overrides the `moduledir` directive. See the related topic about how to declare content as a Git repo for instructions.

Add the `moduledir` directive at the top of the Puppetfile, specifying your module installation directory relative to the location of the Puppetfile.

```
moduledir 'thirdparty'
```

Declare Forge modules in the Puppetfile

Declare Forge modules in your Puppetfile, specifying the version and whether or not code management keeps the module updated.

Specify modules by their full name. You can specify the most recent version of a module, with or without updates, or you can specify a specific version of a module.

Note: Some existing Puppetfiles contain a `forge` setting that provides legacy compatibility with `librarian-puppet`. This setting is non-operational for Code Manager and r10k. To configure how Forge modules are downloaded, specify `forge_settings` in Hiera instead. See the topics about configuring the Forge settings for Code Manager or r10k for details.

In your Puppetfile, specify the modules to install with the `mod` directive. For each module, pass the module name as a string, and optionally, specify what version of the module you want to track.

If you specify no options, code management installs the latest version and keeps the module at that version. To keep the module updated, specify `:latest`. To install a specific version of the module and keep it at that version, specify the version number as a string.

```
mod 'puppetlabs/apache'
mod 'puppetlabs/ntp', :latest
mod 'puppetlabs/stdlib', '0.10.0'
```

This example:

- Installs the latest version of the `apache` module, but does not update it.
- Installs the latest version of the `ntp` module, and updates it when environments are deployed.
- Installs version 0.10.0 of the `stdlib` module, and does not update it.

Declare Git repositories in the Puppetfile

List the modules, data, or other non-module content that you want to install from a Git repository.

To specify any environment content as a Git repository, use the `mod` directive with with the `:git` option. This is useful for modules you don't get from the Forge, such as your own modules, as well as data or other non-module content.

To install content and keep it updated to the master branch, declare the content name and specify the repository with the `:git` directive. Optionally, specify an `:install_path` for the content.

```
mod 'apache',
  :git => 'https://github.com/puppetlabs/puppetlabs-apache'
mod 'site_data',
  :git => 'git@git.example.com:site_data.git',
  :install_path => 'hieradata'
```

This example installs the `apache` module and keeps that module updated with the master branch of the listed repository. It also installs site data content from a Git repository into the environment's `./hieradata/site_data` subdirectory.

Note: Content is installed in the modules directory and treated as a module, unless you use the `:install_path` option. Use this option with non-module content to keep your data separate from your modules.

Specify installation paths for repositories

You can set individual installation paths for any of the repositories that you declare in the Puppetfile.

The `:install_path` option allows you to separate non-module content in your directory structure or to set specific installation paths for individual modules. When you set this option for a specific repository, it overrides the `moduledir` setting.

To install the content into a subdirectory in the environment, specify the directory with the `install_path` option. To install into the root of the environment, specify an empty value.

```
mod 'site_data_1',
  :git => 'git@git.example.com:site_data_1.git',
  :install_path => 'hieradata'
mod 'site_data_2',
  :git => 'git@git.example.com:site_data_2.git',
  :install_path => ''
```

This example installs site data content from the `site_data_1` repository into `./hieradata/site_data` and content from `site_data_2` into `./site_data` subdirectory.

Declare module or data content with SSH private key authentication

To declare content protected by SSH private keys, declare the content as a repository, and then configure the private key setting in your code management tool.

1. Declare your repository content, specifying the Git repo by the SSH URL.

```
mod 'myco/privatemod',
  :git => 'git@git.example.com:myco/privatemod.git'
```

2. Configure the correct private key by setting Code Manager or `r10k` parameters in Hiera:

- To set a key for all Git operations, use the private key setting under `git-settings`.
- To set a private key for an individual remote, set the private key in the `repositories` hash in `git-settings` for each specific remote.

Keep repository content at a specific version

The Puppetfile can maintain repository content at a specific version.

To specify a particular repository version, declare the version you want to track with the following options. Setting these options maintains the repository at that version and deploys any updates made to that particular version.

- `ref`: Specifies the Git reference to check out. This option can reference either a tag, a commit, or a branch.
- `tag`: Specifies the repo by a certain tag value.
- `commit`: Specifies the repo by a certain commit.
- `branch`: Specifies a certain branch of the repo.
- `default_branch`: Specifies a default branch to use for deployment if the specified `ref`, `tag`, `commit`, or `branch` cannot be deployed. You must also specify one of the other version options. This is useful if you are tracking a relative branch of the control repo.

In the Puppetfile, declare the content, specifying the repository and version you want to track.

To install `puppetlabs/apache` and specify the '0.9.0' tag, use the `tag` option.

```
mod 'apache',
  :git => 'https://github.com/puppetlabs/puppetlabs-apache',
  :tag => '0.9.0'
```

To install `puppetlabs/apache` and use the `branch` option to track the 'proxy_match' branch.

```
mod 'apache',
  :git => 'https://github.com/puppetlabs/puppetlabs-apache',
  :branch => 'proxy_match'
```

To install `puppetlabs/apache` and use the `commit` option to track the '8df51aa' commit.

```
mod 'apache',
  :git => 'https://github.com/puppetlabs/puppetlabs-apache',
```

```
:commit => '8df51aa'
```

Declare content from a relative control repo branch

The `branch` option also has a special `:control_branch` option, which allows you to deploy content from a control repo branch relative to the location of the Puppetfile.

Normally, `branch` tracks a specific named branch of a repo, such as `testing`. If you set it to `:control_branch`, it instead tracks whatever control repo branch the Puppetfile is in. For example, if your Puppetfile is in the production branch, content from the production branch is deployed; if a duplicate Puppetfile is located in testing, content from testing is deployed. This means that as you create new branches, you don't have to edit the inherited Puppetfile as extensively.

To track a branch within the control repo, declare the content with the `:branch` option set to `:control_branch`.

```
mod 'hieradata',
  :git      => 'git@git.example.com:organization/hieradata.git',
  :branch   => :control_branch
```

Set a default branch for content deployment

Set a `default_branch` option to specify what branch code management deploys content from if the given option for your repository cannot be deployed.

If you specified a `ref`, `tag`, `commit`, or `branch` option for your repository, and it cannot be resolved and deployed, code management can instead deploy the `default_branch`. This is mostly useful when you set `branch` to the `:control_branch` value.

If your specified content cannot be resolved and you have not set a default branch, or if the default branch cannot be resolved, code management logs an error and does not deploy or update the content.

In the Puppetfile, in the content declaration, set the `default_branch` option to the branch you want to deploy if your specified option fails.

```
# Track control branch and fall-back to master if no matching branch.
mod 'hieradata',
  :git      => 'git@git.example.com:organization/hieradata.git',
  :branch   => :control_branch,
  :default_branch => 'master'
```

Managing code with Code Manager

Code Manager automates the management and deployment of your Puppet code. Push code updates to your source control repo, and then Puppet syncs the code to your masters, so that all your servers start running the new code at the same time, without interrupting agent runs.

How Code Manager works

Code Manager uses `r10k` and the file sync service to stage, commit, and sync your code, automatically managing your environments and modules.

First, create a control repository with branches for each environment that you want to create (such as production, development, or testing). Create a Puppetfile for each of your environments, specifying exactly which modules to install in each environment. This allows Code Manager to create directory environments, based on the branches you've set up. When you push code to your control repo, you trigger Code Manager to pull that new code into a staging code directory (`/etc/puppetlabs/code-staging`). File sync then picks up those changes, pauses Puppet Server to avoid conflicts, and then syncs the new code to the live code directories on your masters.

For more information about using environments in Puppet, see [About Environments](#).

Understanding file sync and the staging directory

To sync your code across multiple masters and to make sure that code stays consistent, Code Manager relies on file sync and two different code directories: the staging directory and the live code directory.

Without Code Manager or file sync, Puppet code lives in the `codedir`, or live code directory, at `/etc/puppetlabs/code`. But the file sync service looks for code in a code staging directory (`/etc/puppetlabs/code-staging`), and then syncs that to the live `codedir`.

Code Manager moves new code from source control into the staging directory, and then file sync moves it into the live code directory. This means you no longer write code to the `codedir`; if you do, the next time Code Manager deploys code from source control, it overwrites your changes.

For more detailed information about how file sync works, see the related topic about file sync.



CAUTION:

Don't directly modify code in the staging directory. Code Manager overwrites it with updates from the control repo. Similarly, don't modify code in the live code directory, because file sync overwrites that with code from the staging directory.

Related information

[About file sync](#) on page 606

File sync helps Code Manager keep your Puppet code synchronized across multiple masters.

Environment isolation metadata and Code Manager

Both your live and staging code directories contain metadata files that are generated by file sync to provide environment isolation for your resource types.

These files, which have a `.pp` extension, ensure that each environment uses the correct version of the resource type. Do not delete or modify these files. Do not use expressions from these files in regular manifests.

These files are generated as Code Manager deploys new code in your environments. If you are new to Code Manager, these files are generated when you first deploy your environments. If you already use Code Manager, the files are generated as you make and deploy changes to your existing environments.

For more details about these files and how they isolate resource types in multiple environments, see [environment isolation](#).

Moving from r10k to Code Manager

Switching from r10k to Code Manager can improve automation of your code management and deployments, but some r10k users might not be ready to switch yet.

Code Manager uses r10k in the background to improve automation of your code management and deployment. However, switching to Code Manager means you can no longer use r10k manually. Because of this, some r10k users might not want to move to Code Manager yet. Specifically, be aware of the following restrictions:

- If you use Code Manager, you **cannot** deploy code manually with r10k. If you depend on the ability to deploy modules directly from r10k, using the `r10k deploy module` command, continue to use your current r10k workflow.
- Code Manager must deploy all control repos to the same directory. If you are using r10k to deploy control repos to different directories, continue to use your current r10k workflow.
- Code Manager does not support system `.SSH` configuration or other shellgit options.
- Code Manager does not support post-deploy scripts.

If you rely on any of the above configurations, or any other r10k configuration that Code Manager doesn't yet support, continue to use your current r10k workflow. If you are using a custom script to deploy code, carefully assess whether Code Manager meets your needs.

Related information

[Managing code with r10k](#) on page 594

r10k is a code management tool that allows you to manage your environment configurations (such as production, testing, and development) in a source control repository.

Configuring Code Manager

To configure Code Manager, enable it in Puppet Enterprise (PE), set up authentication, and test the communication between the control repo and Code Manager.

When you finished configuration, you're ready to deploy environments with Code Manager.

You can enable and configure Code Manager either during or after r10k installation.

To enable Code Manager after a new installation or in an existing PE installation, set Code Manager parameters in the console. You can also configure Code Manager during a fresh PE installation, but only during a text-mode installation.

1. Enable and configure Code Manager, after installation or upgrade, by setting parameters in the master profile class in the PE console. Alternatively, enable during a fresh installation, by setting parameters in `pe.conf`
2. Test your control repo.
3. Set up authentication for Code Manager.
4. Test Code Manager.

Important: If you enable Code Manager, do not attempt to follow the workflows in the PE getting started guides. The `puppet module` command is not compatible with Code Manager.

Upgrading from r10k to Code Manager

If you are upgrading from r10k to Code Manager, you must first disable your old r10k installation.

If you are upgrading from r10k to Code Manager, check the following before enabling Code Manager:

- If you used r10k prior to PE 2015.3, you might have configured r10k in the console using the `pe_r10k` class. If so, you must remove the `pe_r10k` class in the console **before** configuring Code Manager.
- If you used any previous versions of r10k, disable any tools that might automatically run it. Most commonly, this is the `zack-r10k` module. Code Manager cannot install or update code properly if other tools are running r10k.

When you start using Code Manager, it runs r10k in the background. You can no longer directly interact with r10k or use the `zack-r10k` module.

Enable Code Manager

Usually, you enable Code Manager after PE is already installed. If you are automating your PE installation and using an existing control repo and SSH key, you can enable Code Manager during the PE installation process

Enable Code Manager after installation

To enable Code Manager after installing PE or in an existing installation, set parameters in the console.

1. In the console, set the following parameters in the `puppet_enterprise::profile::master` class in the PE Master node group.
 - `code_manager_auto_configure` to `true`: This enables and configures both Code Manager and file sync.
 - `r10k_remote`: This is the location of your control repository. Enter a string that is a valid URL for your Git control repository. For example: `"git@<YOUR.GIT.SERVER.COM>:puppet/control.git"`.

Note: Some Git providers, such as Bitbucket, might have additional requirements for enabling SSH access. See your provider's documentation for information.
 - `r10k_private_key`: Enter a string specifying the path to the SSH private key that permits the `pe-puppet` user to access your Git repositories. This file must be located on the master, owned by the `pe-puppet` user, and located in a directory that the `pe-puppet` user has permission to view. We recommend `/etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa`.

2. Run Puppet on all masters.

If you run Puppet for all your masters at the same time, such as with **Run Puppet** in the console, you might see errors like this your compilers' logs:

```
2015-11-20 08:14:38,308 ERROR [clojure-agent-send-off-pool-0]
[p.e.s.f.file-sync-client-core] File sync failure: Unable to get
latest-commits from server (https://master.example.com:8140/file-sync/v1/
latest-commits).
java.net.ConnectException: Connection refused
```

You can ignore these errors. They occur because Puppet Server is restarting while the compilers are trying to poll for new code. These errors generally stop as soon as the Puppet Server on the master has finished restarting.

Next, set up authentication.

Enable Code Manager during installation

To configure Code Manager during a fresh installation of PE, add parameters to the `pe.conf` file.

Use these parameters **only** with text-mode PE installation, not with web-based installation. Adding the listed parameters enables and configures file sync and Code Manager.

1. Add the following three parameters to `pe.conf` *before* installation, adding your specific URL and directory information:

```
"puppet_enterprise::profile::master::r10k_remote":
  "git@<YOUR.GIT.SERVER.COM>:puppet/control.git"

"puppet_enterprise::profile::master::r10k_private_key": "/etc/puppetlabs/
puppetserver/ssh/id-control_repo.rsa"

puppet_enterprise::profile::master::code_manager_auto_configure": true
```

These parameters specify the private key location and the control repo URL, and enables Code Manager and file sync.

- `puppet_enterprise::profile::master::r10k_private_key`

This setting accepts a string that specifies the path to the future location of the SSH private key used to access your Git repositories. The `pe-puppet` user **must** be able to access this location; we recommend `/etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa`.

This SSH private key file, which you created when you set up your control repository, must be located on the Puppet master and owned by the `pe-puppet` user. **After** PE installation is completed, place the key in the specified location. You do this in step 3 below.

- `"puppet_enterprise::profile::master::r10k_remote"`

This setting specifies the location of the control repository. It accepts a string that is a valid URL for your Git control repository.

- `puppet_enterprise::profile::master::code_manager_auto_configure`

This setting configures Code Manager, the Git control repository to use for storing code, and the private key for accessing your Git repos. Accepts the values `true` or `false`. Set it to `true` to enable and configure Code Manager. A `false` setting disables Code Manager and file sync.

2. Complete the installation, following the steps in the text-based installation instructions.
3. After installation is complete, place the SSH private key you created when you set up your control repository in the `r10k_private_key` location.

Next, set up authentication.

Related information

[Install using text install](#) on page 136

When you run the installer in text mode, you provide a configuration file (`pe.conf`) to the installer. The `pe.conf` file contains values for the parameters needed for installation.

[Configuration parameters and the pe.conf file](#) on page 138

A `pe.conf` file is a HOCON formatted file that declares parameters and values needed to install, upgrade, and configure PE.

Set up authentication for Code Manager

To securely deploy environments, Code Manager needs an authentication token for both authentication and authorization.

Before you begin

Configure the Puppet access command line tool.

To generate a token for Code Manager, first assign a user to the deployment role, and then request an authentication token.

Related information

[Configuring puppet-access](#) on page 244

The `puppet-access` command allows users to generate and manage authentication tokens from the command line of any workstation (Puppet-managed or not), without the need to SSH into the Puppet master.

Assign a user to the deployment role

To request an authentication token, you must first assign a user the correct permissions with role-based access control (RBAC).

1. In the console, create a deployment user. We recommend that you create a dedicated deployment user for Code Manager use.
2. Add the deployment user to the **Code Deployers** role. This role is automatically created on install, with default permissions for code deployment and token lifetime management.

Next, request the authentication token.

Related information

[Add a user to a user role](#) on page 236

When you add users to a role, the user gains the permissions that are applied to that role. A user can't do anything in PE until they have been assigned to a role.

[Assign a user group to a user role](#) on page 242

After you've imported a group, you can assign it a user role, which gives each group member the permissions associated with that role. You can add user groups to existing roles, or you can create a new role, and then add the group to the new role.

Request an authentication token for deployments

Request an authentication token for the deployment user to enable secure deployment of your code.

By default, authentication tokens have a five-minute lifetime. With the `Override default expiry` permission set, you can change the lifetime of the token to a duration better suited for a long-running, automated process.

Generate the authentication token using the `puppet-access` command.

1. From the command line on the master, run `puppet-access login --lifetime 180d`. This command both requests the token and sets the token lifetime to 180 days.

Tip: You can add flags to the request specifying additional settings such as the token file's location or the URL for your RBAC API. See [Configuration file settings for puppet-access](#).

2. Enter the username and password of the deployment user when prompted.

The generated token is stored in a file for later use. The default location for storing the token is `~/ .puppetlabs/ token`. To view the token, run `puppet-access show`. Next, test the connection to the control repo.

Related information

[Setting a token-specific lifetime](#) on page 246

Tokens have a default lifetime of five minutes, but you can set a different lifetime for your token when you generate it. This allows you to keep one token for multiple sessions.

[Generate a token for use by a service](#) on page 243

If you need to generate a token for use by a service and the token doesn't need to be saved, use the `--print` option.

Test the control repo

To make sure that Code Manager can connect to the control repo, test the connection to the repository.

From the command line, run `puppet-code deploy --dry-run`.

- If the control repo is set up properly, this command fetches and displays a list of the environments in the control repo.
- If an environment is not set up properly or causes an error, it does not appear in the returned list.

Test Code Manager

To test whether Code Manager deploys your environments correctly, trigger a single environment deployment on the command line.

Deploy a single environment

Test Code Manager by deploying a single test environment.

This deploys the test environment, and then returns deployment results with the SHA (a checksum for the content stored) for the control repo commit.

From the command line, deploy one environment by running `puppet-code deploy my_test_environment --wait`

Check to make sure the environment was deployed. If so, you've set up Code Manager correctly.

If the deployment does not work as you expect, check over the configuration steps, or refer to the troubleshooting guide for help.

After Code Manager is fully enabled and configured, you can trigger it to deploy your environments.

There are several ways to trigger deployments, depending on your needs.

- Manually, on the command line.
- Automatically, with a webhook.
- Automatically, with a custom script that hits the deploys endpoint.

Code Manager console settings

After Code Manager is configured, you can adjust some settings in the master profile in the console.

These options are required for Code Manager to work, unless otherwise noted.

Setting	Description	Example
<code>code_manager_auto_configure</code>	Set to true to auto-configure Code Manager.	<code>true</code>
<code>r10k_remote</code>	The location of the Git control repository. Enter a string that is a valid URL for your control repository.	<code>'git@<YOUR.GIT.SERVER.COM>:puppet/control.git'</code>

Setting	Description	Example
<code>r10k_private_key</code>	Required when using the SSH protocol; optional in all other cases. Enter a string that is the path to the file containing the private key used to access all Git repositories.	<code>'/etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa'</code>
<code>r10k_proxy</code>	Optional. A proxy setting <code>r10k</code> Code Manager uses when accessing the Forge. If empty, no proxy settings are used.	<code>'http://proxy.example.com:3128'</code>

To further customize your Code Manager configuration with Hiera, see the related topic about customizing your configuration.

Customize Code Manager configuration in Hiera

To customize your Code Manager configuration, set parameters with Hiera.

Add Code Manager parameters in the `puppet_enterprise::master::code_manager` class with Hiera.

Important:

Do not edit the Code Manager configuration file manually. Puppet manages this configuration file automatically and undoes any manual changes you make.

Related information

[Configure settings with Hiera](#) on page 190

Hiera is a hierarchy-based method of configuration management that relies on a “defaults, with overrides” system. When you add a parameter or setting to your Hiera data, Hiera searches through the data in the order it was written to find the value you want to change. Once found, it overrides the default value with the new parameter or setting. You can use Hiera to manage your PE configuration settings.

Add Code Manager parameters in Hiera

To customize your Code Manager configuration, add parameters to your control repository hierarchy in the `hieradata/common.yaml` file.

1. Add the parameter to `hieradata/common.yaml` in the format:

```
puppet_enterprise::master::code_manager::<parameter>

puppet_enterprise::master::code_manager::deploy_pool_size: 4
puppet_enterprise::master::code_manager::timeouts_deploy: 300
```

The first parameter in this example increases the size of the default worker pool, and the second reduces the maximum time allowed for deploying a single environment.

2. Run Puppet on the master to apply changes.

Configuring post-environment hooks

Configure post-environment hooks to trigger custom actions after your environment deployment.

To configure list of hooks to run after an environment has been deployed, specify the Code Manager `post_environment_hook` setting in Hiera.

This parameter accepts an array of hashes, with the following keys:

- `url`
- `use-client-ssl`

For example, to set up Code Manager to update classes in the console after deploying code to your environments.

```
puppet_enterprise::master::code_manager::post_environment_hooks:
  - url: 'https://console.yourorg.com:4433/classifier-api/v1/update-classes'

  use-client-ssl: true
```

url

This setting specifies an HTTP URL to send a request to, with the result of the environment deploy. The URL receives a POST with a JSON body with the structure:

```
{
  "deploy-signature": "482f8d3adc76b5197306c5d4c8aa32aa8315694b",
  "file-sync": {
    "environment-commit": "6939889b679fdb1449545c44f26aa06174d25c21",
    "code-commit": "ce5f7158615759151f77391c7b2b8b497aaebce1"
  },
  "environment": "production",
  "id": 3,
  "status": "complete"
}
```

use-client-ssl

Specifies whether the client SSL configuration is used for HTTPS connections. If the hook destination is a server using the Puppet CA, set this to `true`; otherwise, set it to `false`. Defaults to `false`.

Configuring garbage collection

By default, Code Manager retains environment deployments in memory for one hour, but you can adjust this by configuring garbage collection.

To configure the frequency of Code Manager garbage collection, specify the `deploy_ttl` parameter in Hiera. By default, deployments are kept for one hour.

Specify the time as a string using any of the following suffixes:

- `d` - days
- `h` - hours
- `m` - minutes
- `s` - seconds
- `ms` - milliseconds

For example, a value of `30d` would configure Code Manager to keep the deployment in memory for 30 days, and a value of `48h` would maintain the deployment in memory for 48 hours.

If the value of `deploy_ttl` is less than the combined values of `timeouts_fetch`, `timeouts_sync`, and `timeouts_deploy`, all completed deployments are retained indefinitely, which might significantly slow the performance of Code Manager over time.

Configuring Forge settings

To configure how Code Manager downloads modules from the Forge, specify `forge_settings` in Hiera.

This parameter configures where modules should be installed from, and sets a proxy for all interactions. The `forge_settings` parameter accepts a hash with the following values:

- `baseurl`
- `proxy`

baseurl

Indicates where modules should be installed from. Defaults to `https://forgeapi.puppetlabs.com`.

```
puppet_enterprise::master::code_manager::forge_settings:
  baseurl: 'https://private-forge.mysite'
```

proxy

Sets the proxy for all interactions.

This setting overrides the global `proxy` setting on operations only. You can set an unauthenticated proxy or an authenticated proxy with either Basic or Digest authentication. See the global `proxy` setting for more information and examples.

```
proxy: 'http://proxy.example.com:3128'
```

Configuring Git settings

To configure Code Manager to use a private key, a proxy, or multiple repositories with Git, specify the `git_settings` parameter.

The `git_settings` parameter accepts a hash of the following settings:

- `private-key`
- `proxy`
- `repositories`

Note: The `git_settings` hash cannot be used with the default Code Manager settings for `r10k_remote` and `r10k_private_key`. To avoid errors, remove the following settings from the `puppet_enterprise::profile::master` class:

- `r10k_private_key`
- `r10k_remote`
- `code_manager_auto_configure`

private-key

Specifies the file containing the default private key used to access control repositories, such as `/etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa`. This file must have read permissions for the `pe-puppet` user. The SSH key cannot require a password. This setting is required, but by default, the value is supplied by the master profile in the console.

proxy

Sets a proxy specifically for operations that use an HTTP(S) transport.

This setting overrides the global `proxy` setting on operations only. You can set an unauthenticated proxy or an authenticated proxy with either Basic or Digest authentication. To set a proxy for a specific repository only, set `proxy` in the `repositories` subsetting of `git_settings`. See the global `proxy` setting for more information and examples.

```
proxy: 'http://proxy.example.com:3128'
```

repositories

Specifies a list of repositories and their respective private keys. Use this setting if you want to use multiple control repos.

To use multiple control repos, the `sources` setting and the `repositories` setting must match. Accepts the following settings:

- `remote`: The repository to which these settings apply.
- `private-key`: The file containing the private key to use for this repository. This file must have read permissions for the `pe-puppet` user.
- `proxy`: The proxy setting allows you to set or override the global proxy setting for a single, specific repository. See the global proxy setting for more information and examples.

The `repositories` setting accepts a hash in the following format:

```
repositories:
  - remote: "jfkennedy@gitserver.puppetlabs.net:repositories/repo_one.git"
    private-key: "/test_keys/jfkennedy"
  - remote: "whaft@gitserver.puppetlabs.net:repositories/repo_two.git"
    private-key: "/test_keys/whaft"
  - remote: "https://git.example.com/git_repos/environments.git"
    proxy: "https://proxy.example.com:3128"
```

Configuring proxies

To configure proxy servers, use the `proxy` setting. You can set a global proxy for all HTTP(S) operations, for all Git or Forge operations, or for a specific Git repository only.

proxy

To set a proxy for all operations occurring over an HTTP(S) transport, set the global `proxy` setting. You can also set an authenticated proxy with either Basic or Digest authentication.

To override this setting for or operations only, set the `proxy` subsetting under the `git_settings` or `forge_settings` parameters. To override for a specific repository, set a proxy in the `repositories` list of `git_settings`. To override this setting with no proxy for , , or a particular repository, set that specific `proxy` setting to an empty string.

In this example, the first proxy is set up without authentication, and the second proxy uses authentication:

```
proxy: 'http://proxy.example.com:3128'
proxy: 'http://user:password@proxy.example.com:3128'
```

Configuring sources

If you are managing more than one repository with Code Manager, specify a map of your source repositories.

Use the `source` parameter to specify a map of sources. Configure this setting if you are managing more than just Puppet environments, such as when you are also managing Hiera data in its own control repository.

To use multiple control repos, the `sources` setting and the `repositories` setting must match.

If `sources` is set, you cannot use the global `remote` setting. If you are using multiple sources, use the `prefix` option to prevent collisions.

The `sources` setting accepts a hash with the following subsettings:

- `remote`
- `prefix`

```
myorg:
  remote: "git://git-server.site/myorg/main-modules"
  prefix: true
mysource:
  remote: "git://git-server.site/mysource/main-modules"
  prefix: "testing"
```


remote

Specifies the location from which you fetch the source repository. The remote must be able to be fetched without any interactive input. That is, you cannot be prompted for usernames or passwords in order to fetch the remote.

Accepts a string that is any valid URL that r10k can clone, such as `git://git-server.site/my-org/main-modules`.

prefix

Specifies a string to prefix to environment names. Alternatively, if `prefix` is set to `true`, the source's name is used. This prevents collisions when multiple sources are deployed into the same directory.

For example, with two "main-modules" environments set up in the same base directory, the `prefix` setting differentiates the environments: the first is named "myorg-main-modules", and the second is "testing-main-modules".

```
myorg:
  remote: "git://git-server.site/myorg/main-modules"
  prefix: true
mysource:
  remote: "git://git-server.site/mysource/main-modules"
  prefix: "testing"
```

Code Manager parameters

Code Manager parameters

Parameter	Description	Value	Default
<code>remote</code>	The location of the Git control repository. Either the <code>remote</code> or <code>sources</code> parameters must be specified. If <code>remote</code> is already specified in the master profile, that value is used here. If both the <code>sources</code> and <code>remote</code> keys are specified, the <code>sources</code> key overrides <code>remote</code> .	A string that is a valid SSH URL for your Git remote.	No default.
<code>authenticate_webhook</code>	Turns on RBAC authentication for the <code>/v1/webhook</code> endpoint.	Boolean.	<code>true</code>
<code>cachedir</code>	The path to the location where Code Manager caches Git repositories.	A valid file path.	<code>/opt/puppetlabs/server/data/code-manager/cache</code> .
<code>certname</code>	The certname of the Puppet signed certs to use for SSL.	A valid certname.	Defaults to <code>\$\$::clientcert</code> .
<code>data</code>	The path to the directory where Code Manager stores internal file content.	A valid file path.	<code>/opt/puppetlabs/server/data/code-manager</code>
<code>deploy_pool_size</code>	Specifies the number of threads in the worker pool; this dictates how many deploy processes can run in parallel.	An integer.	<code>2</code>

Parameter	Description	Value	Default
deploy_ttl	Specifies the length of time completed deployments are retained before garbage collection. For usage details, see configuring garbage collection.	The time as a string, using any of the following suffixes: <ul style="list-style-type: none"> • d - days • h - hours • m - minutes • s - seconds • ms - milliseconds 	1h (one hour)
hostcrl	The path to the SSL CRL.	A valid file path.	<code>\$puppet_enterprise::params::hostcrl</code>
localcacert	The path to the SSL CA cert.	A valid file path.	<code>\$puppet_enterprise::params::localcacert</code>
post_environment_hooks	<p>A list of hooks to run after an environment has been deployed. Specifies:</p> <ul style="list-style-type: none"> • an HTTP URL to send deployment results to, • Whether to use client SSL for HTTPS connections <p>For usage details, see configuring post-environment hooks.</p>	An array of hashes that can include the keys: <ul style="list-style-type: none"> • url • use-client-ssl 	No default.
timeouts_deploy	Maximum execution time (in seconds) for deploying a single environment.	An integer	600
timeouts_fetch	Maximum execution time (in seconds) for updating the control repo state.	An integer.	30
timeouts_hook	Maximum time (in seconds) to wait for a single post-environment hook URL to respond. Used for both the socket connect timeout and the read timeout, so the longest total timeout would be twice the specified value.	An integer.	30
timeouts_shutdown	Maximum time (in seconds) to wait for in-progress deploys to complete when shutting down the service.	An integer.	610

Parameter	Description	Value	Default
timeouts_sync	Maximum time (in seconds) that a request sent with a <code>wait</code> parameter must wait for all compilers to receive deployed code before timing out.	An integer.	60
webserver_ssl_host	The host that Code Manager listens on.	An IP address	0.0.0.0
webserver_ssl_port	The port that Code Manager listens on. Port 8170 must be open if you're using Code Manager.	A port number.	8170

r10k specific parameters

These parameters are specific to r10k, which Code Manager uses in the background.

Parameter	Description	Value	Default
environmentdir	The single directory to which Code Manager deploys all sources. If <code>file_sync_auto_commit</code> is set to <code>true</code> , this defaults to <code>/etc/puppetlabs/code-staging/environments</code> . See <code>file_sync_auto_commit</code> .	Directory	<code>/etc/puppetlabs/code-staging/environments</code>
forge_settings	Contains settings for downloading modules from the Forge. See Configuring Forge settings for usage details.	Accepts a hash of: <ul style="list-style-type: none"> <code>baseurl</code> <code>proxy</code> 	No default.
invalid_branches	Specifies, for all sources, how branch names that cannot be cleanly mapped to Puppet environments are handled.	<ul style="list-style-type: none"> <code>'correct'</code>: Replaces non-word characters with underscores and gives no warning. <code>'error'</code>: Ignores branches with non-word characters and issues an error. 	<code>'error'</code>

Parameter	Description	Value	Default
git_settings	<p>Configures settings for Git:</p> <ul style="list-style-type: none"> Specifies the file containing the default private key used to access control repositories. Sets or overrides the global proxy setting specifically for Git operations that use an HTTP(S) transport. Specifies a list of repositories and their respective private keys. <p>See Configuring Git settings for usage details.</p>	<p>Accepts a hash of:</p> <ul style="list-style-type: none"> private-key proxy repositories 	Default private-key value as set in console. No other default settings.
proxy	<p>Configures a proxy server to use for all operations that occur over an HTTP(S) transport.</p> <p>See Configuring proxies for usage details.</p>	<p>Accepts:</p> <ul style="list-style-type: none"> A proxy server. An authenticated proxy with Basic or Digest authentication. An empty value. 	No default.
sources	<p>Specifies a map of sources to be passed to r10k. Use if you are managing more than just Puppet environments.</p> <p>See Configuring sources for usage details.</p>	<p>A hash of:</p> <ul style="list-style-type: none"> remote prefix 	No default.

Triggering Code Manager on the command line

The `puppet-code` command allows you to trigger Code Manager from the command line to deploy your environments.

Installing and configuring puppet-code

PE automatically installs and configures the `puppet-code` command on your masters as part of the included PE client tools package. You can also set up `puppet-code` on an agent node or on a workstation, customize configuration for different users, or change the configuration settings.

The global configuration settings for Linux and macOS systems are in a JSON file at `/etc/puppetlabs/client-tools/puppet-code.conf`. The default configuration file looks something like:

```
{
  "cacert": "/etc/puppetlabs/puppet/ssl/certs/ca.pem",
  "token-file": "~/.puppetlabs/token",
  "service-url": "https://<PUPPET MASTER HOSTNAME>:8170/code-manager"
}
```

Important:

On a PE-managed machine, Puppet manages this file for you. Do not manually edit this file, because Puppet overwrites your new values the next time it runs.

Additionally, you can set up `puppet-code` on an agent node or on a workstation not managed by PE. And on any machine, you can set up user-specific config files. For instructions, see the related topic about advanced configuration of `puppet-code`.

You can also override existing configuration settings on a case-by-case basis on the command line. When you deploy environments with the `puppet-code deploy` command, you can specify either an alternative config file or particular config settings directly in the command. For examples, see the related topic about deploying environments with `puppet code`.

Windows paths

The global `puppet-code` configuration file on Windows systems is located at `C:\ProgramData\PuppetLabs\client-tools\puppet-code.conf`.

The default configuration file looks something like:

```
{
  "cacert": "C:\\ProgramData\\PuppetLabs\\puppet\\etc\\ssl\\certs\\ca.pem",
  "token-file": "C:\\Users\\<username>\\.puppetlabs\\token",
  "service-url": "https://<PUPPET MASTER HOSTNAME>:8170/code-manager"
}
```

Configuration precedence and puppet-code

There are several ways to configure `puppet-code`, but some configuration methods take precedence over others.

If no other configuration is specified, `puppet-code` uses the settings in the global configuration file. User-specific configuration files override the global configuration file.

If you specify a configuration file on the command line, Puppet temporarily uses that configuration file **only** and does not read the global or user-specific config files at all.

Finally, if you specify individual configuration options directly on the command line, those options temporarily take precedence over *any* configuration file settings.

Deploying environments with puppet-code

To deploy environments with the `puppet-code` command, use the `deploy` action, either with the name of a single environment or with the `--all` flag.

The `deploy` action deploys the environments, but returns only deployment *queuing* results by default. To view the results of the deployment itself, add the `--wait` flag.

The `--wait` flag deploys the specified environments, waits for file sync to complete code deployment to the live code directory and all compilers, and then returns results. In deployments that are geographically dispersed or have a large quantity of environments, completing code deployment can take up to several minutes.

The resulting message includes the deployment signature, which is the commit SHA of the control repo used to deploy the environment. The output also includes two other SHAs that indicate that file sync is aware that the environment has been newly deployed to the code staging directory.

To temporarily override default, global, and user-specific configuration settings, specify the following configuration options on the command line:

- `--cacert`
- `--token-file, -t`
- `--service-url`

Alternately, you can specify a custom `puppet-code.conf` configuration file by using the `--config-file` option.

Running puppet-code on Windows

If you're running these commands from a managed or non-managed Windows workstation, you must specify the full path to the command.

```
C:\Program Files\Puppet Labs\Client\bin\puppet code deploy mytestenvironment
--wait
```

Deploy environments on the command line

To deploy environments, use the `puppet-code deploy` command, specifying the environments to deploy.

To deploy environments, on the command line, run `puppet-code deploy`, specifying the environment.

Specify the environment by name. To deploy all environments, use the `--all` flag.

Optionally, you can specify the `--wait` flag to return results after the deployment is finished. Without the `--wait` flag, the command returns only queuing results.

```
puppet-code deploy myenvironment --wait
```

```
puppet-code deploy --all --wait
```

Both of these commands deploy the specified environments, and then return deployment results with a control repo commit SHA for each environment.

Related information

[Reference: puppet-code command](#) on page 575

The `puppet-code` command accepts options, actions, and `deploy` action options.

Deploy with a custom configuration file

You can deploy environments with a custom configuration file that you specify on the command line.

To deploy all environments using the configuration settings in a specified config file, run the command `puppet-code deploy` command with a `--config-file` flag specifying the location of the config file.

```
puppet-code --config-file ~/.puppetlabs/myconfigfile/puppet code.conf deploy
--all
```

Deploy with command-line configuration settings

You can override an existing configuration setting on a per-use basis by specifying that setting on the command line.

Specify the setting, which is used on this deployment only, on the command line.

```
puppet-code --service-url "https://puppet.example.com:8170/code-manager"
deploy mytestenvironment
```

This example deploys 'mytestenvironment' using global or user-specific config settings (if set), except for `--service-url`, for which it uses the value specified on the command line ("https://puppet.example.com:8170/code-manager").

Advanced puppet-code configuration

You can set up the `puppet-code` command on an agent node or on a workstation not managed by PE. And on any machine, you can set up user-specific config files.

You can set up the `puppet-code` command on an agent node or on a workstation not managed by PE. And on any machine, you can set up user-specific config files.

The `puppet-code.conf` file is a JSON configuration file for the `puppet-code` command. For Linux or Mac OS X operating systems, it looks something like:

```
{
  "cacert": "/etc/puppetlabs/puppet/ssl/certs/ca.pem",
```

```
"token-file": "~/.puppetlabs/token",
"service-url": "https://<PUPPET MASTER HOSTNAME>:8170/code-manager"
}
```

For Windows systems, use the entire Windows path, such as:

```
{
"cacert": "C:\\ProgramData\\PuppetLabs\\puppet\\etc\\ssl\\certs\\ca.pem",
"token-file": "C:\\Users\\<username>\\.puppetlabs\\token",
"service-url": "https://<PUPPET MASTER HOSTNAME>:8170/code-manager"
}
```

Related information

[Installing PE client tools](#) on page 169

PE client tools are a set of command line tools that let you access Puppet Enterprise services from a workstation that might or might not be managed by Puppet.

Configure puppet-code on agents and workstations

To use puppet-code on an agent node or on a workstation that is not managed by PE, install the client tools package and configure puppet-code on that machine.

Before you begin

Download and install the client tools package. See the client tools documentation for instructions.

Create a config file called puppet-code.conf in the client tools directory.

- For Linux and Mac OS X systems, the default client tools directory is /etc/puppetlabs/client-tools
- For Windows systems, the default client tools directory is C:\ProgramData\PuppetLabs\client-tools

Configure puppet-code for different users

On any machine, whether it is a master, an agent, or a workstation not managed by PE, you can set up specific puppet-code configurations for specific users.

Before you begin

If PE is **not** installed on the workstation you are configuring, see instructions for configuring puppet-code on agents and workstations.

1. Create a puppet-code.conf file in the user's client tools directory.
 - For Linux or Mac OS X systems, place the file in the user's ~/.puppetlabs/client-tools/
 - For Windows systems, place the file in the default user config file location: C:\Users\<username>\.puppetlabs\ssl\certs\ca.pem
2. In the user's puppet-code.conf file, specify the cacert, token-file, and service-url settings in JSON format.

Reference: puppet-code command

The puppet-code command accepts options, actions, and deploy action options.

Usage: puppet-code [global options] <action> [action options]

Global puppet-code options

Global options set the behavior of the puppet-code command on the command line.

Option	Description	Accepted arguments
--help, -h	Prints usage information for puppet-code.	none

Option	Description	Accepted arguments
<code>--version, -V</code>	Prints the application version.	none
<code>--log-level, -l</code>	Sets the log verbosity. It accepts one log level as an argument.	log levels: none, trace, debug, info, warn, error, fatal.
<code>--config-file, -c</code>	Sets a <code>puppet-code.conf</code> file that takes precedence over all other existing <code>puppet-code.conf</code> files.	A path to a <code>puppet-code.conf</code> file.
<code>--cacert</code>	Sets a Puppet CA certificate that overrides the <code>cacert</code> setting in any configuration files.	A path to the location of a CA Certificate.
<code>--token-file, -t</code>	Sets an authentication token that overrides the <code>token-file</code> setting in any configuration files.	A path to the location of the authentication token.
<code>--service-url</code>	Sets a base URL for the Code Manager service, overriding <code>service-url</code> settings in any configuration files.	A valid URL to the service.

puppet-code actions

The `puppet-code` command can perform print, deploy, and status actions.

Action	Result	Arguments	Options
<code>puppet-code print-config</code>	Prints out the resolved <code>puppet-code</code> configuration.	none	none
<code>puppet-code deploy</code>	Runs remote code deployments with the Code Manager service. By default, returns only deployment queuing results.	Accepts either the name of a single environment or the <code>--all</code> flag.	<code>--wait, -w</code> ; <code>--all</code> ; <code>--dry-run</code> ; <code>--format, -F</code>
<code>puppet-code status</code>	Checks whether Code Manager and file sync are responding. By default, details are returned at the info level.	Accepts log levels none, trace, info, warn, error, fatal.	none

puppet-code deploy action options

Modify the `puppet-code deploy` action with action options.

Option	Description
<code>--wait, -w</code>	Causes <code>puppet-code deploy</code> to: <ol style="list-style-type: none"> 1. Start a deployment, 2. Wait for the deployment to complete, 3. Wait for file sync to deploy the code to all compilers, and 4. Return the deployment signature with control repo commit SHAs for each environment. <p>The return output also includes two other SHAs that indicate that file sync is aware that the environment has been newly deployed to the code-staging directory.</p>
<code>--all</code>	Tells <code>puppet-code deploy</code> to start deployments for all Code Manager environments.
<code>--dry-run</code>	Tests the connections to each configured remote and, if successfully connected, returns a consolidated list of the environments from all remotes. The <code>--dry-run</code> flag implies both <code>--all</code> and <code>--wait</code> .
<code>--format, -F</code>	Specifies the output format. The default and only supported value is 'json'.

puppet-code.conf configuration settings

Temporarily specify `puppet-code.conf` configuration settings on the command line.

Setting	Controls	Linux and Mac OS X default	Windows default
<code>cacert</code>	Specifies the path to the Puppet CA certificate to use when connecting to the Code Manager service over SSL.	<code>/etc/puppetlabs/ puppet/ssl/ certs/ca.pem</code>	<code>C:\ProgramData \PuppetLabs\puppet \etc\ssl\certs \ca.pem</code>
<code>token-file</code>	Specifies the location of the file containing the authentication token for Code Manager.	<code>~/.puppetlabs/ token</code>	<code>C:\Users \<username> \.puppetlabs\token</code>
<code>service-url</code>	Specifies the base URL of the Code Manager service.	<code>https:// <PUPPET MASTER HOSTNAME>:8170/ code-manager</code>	<code>https:// <PUPPET MASTER HOSTNAME>:8170/ code-manager</code>

Triggering Code Manager with a webhook

To deploy your code, you can trigger Code Manager by hitting a web endpoint, either through a webhook or a custom script. The webhook is the simplest way to trigger Code Manager.

Custom scripts are a good alternative if you have requirements such as existing continuous integration systems (including Continuous Delivery for Puppet Enterprise (PE)), privately hosted Git repos, or custom notifications. For information about writing a script to trigger Code Manager, see the related topic about creating custom scripts.

Code Manager supports webhooks for GitHub, Bitbucket Server (formerly Stash), Bitbucket, and Team Foundation Server. The webhook must only be used by the control repository. It can't be used by any other repository (for example, other internal component module repositories).

Important: Code Manager webhooks are not compatible with Continuous Delivery for PE. If your organization uses Continuous Delivery for PE, you must use a method other than webhooks to deploy environments.

Creating a Code Manager webhook

To set up the webhook to trigger environment deployments, you must create a custom URL, and then set up the webhook with your Git host.

Creating a custom URL for the Code Manager webhook

To trigger deployments with a webhook, you'll need a custom URL to enable communication between your Git host and Code Manager.

Code Manager supports webhooks for GitHub, Bitbucket Server (formerly Stash), Bitbucket, GitLab (Push events only), and Team Foundation Server (TFS). To use the GitHub webhook with the Puppet signed cert, disable SSL verification.

To create the custom URL for your webhook, use the following elements:

- The name of the Puppet master server (for example, `code-manager.example.com`).
- The Code Manager port (for example, 8170).
- The endpoint (`/code-manager/v1/webhook/`).
- Any relevant query parameters (for example, `type=github`).
- The complete authentication token you generated earlier (`token=<TOKEN>`), passed with the `token` query parameter.

For example, the URL for a GitHub webhook might look like this:

```
https://code-manager.example.com:8170/code-manager/v1/webhook?
type=github&token=<TOKEN>
```

The URL for a Stash webhook might look something like this:

```
https://code-manager.example.com:8170/code-manager/v1/webhook?
type=stash&prefix=dev&token=<TOKEN>
```

With the complete token attached, a GitHub URL looks something like this:

```
https://code-manager.example.com:8170/code-manager/v1/webhook?
type=github&token=eyJhbGciOiJSUzUxMiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJkZXBsbn3kiLCJpYXQiOiJlY0E0
HGRC0r2J87Pj2sDsyz-
EMK-7sZalBTswy2e3uSv1Z6ulXaIQQd69PQnSSBExQotExDgXZInyQa_le2gwTo4smjUaBd6_lnPYr6GJ4hjB4-
fT8dNnVuvZaE5WPkKt-
sNJKJwE9LiEd4W42aCYfse1KNgPuXR5m77SRsUy86ymthVPKHqgexEyuS7LGeQJvyQE1qEe jSdbiLg6zn1JXhg1V
NrE17oxrrNkU0ZxioUgDeqGycwvNIaMazztM9NyD-
dWmZc4dKJsqm0su0CRkMSWcYPPaeIcsYFI7XSaeC65N4RLIKhUfwIxxE-
uODEhcl3mTr9rwZGnVMu3WrY7t6w1bDM9FomPejGM2aJoZ05PinAIYvX3oH5QJ9fam0pVLb-
mI3bvKkm2wKAgpc4Dh1ut9sqLWkG8-
AwMA4r_oEvLwFzf8clzk34zNyPG7BvlnPle99HjQues690L-
fknSdFiXyRZeRThvZop0SWJzvUSR49etmk-
OxnMbQE4tCBWZr_khEG5jUDzeKt3PIiXdxmUaaEPHzo6Vl9XIY5
```

Code Manager webhook query parameters

The following query parameters are permitted in the Code Manager webhook.

The `token` query is mandatory, unless you disable `authenticate_webhook` in the Code Manager configuration.

- `type`: Required. Specifies which type of post body to expect. Accepts:
 - `github`: GitHub
 - `gitlab`: GitLab
 - `stash`: Bitbucket Server (Stash)
 - `bitbucket`: Bitbucket
 - `tfs-git`: Team Foundation Server (resource version 1.0 is supported)
- `prefix`: Specifies a prefix for converting branch names to environments.
- `token`: Specifies the entire PE authorization token.

Setting up the Code Manager webhook on your Git host

Enter the custom URL you created for Code Manager into your Git server's webhook form as the payload URL.

The content type for webhooks is JSON.

Exactly how you set up your webhook varies, depending on where your Git repos are hosted. For example, in your GitHub repo, click on **Settings > Webhooks & services** to enter the payload URL and enter `application/json` as the content type.

Tip: On Bitbucket Server, the server configuration menu has settings for both "hooks" and "webhooks." To set up Code Manager, use the webhooks configuration. For proper webhook function with Bitbucket Server, make sure you are using the Bitbucket Server 5.4 or later and the latest fix version of PE.

After you've set up your webhook, your Code Manager setup is complete. When you commit new code and push it to your control repo, the webhook triggers Code Manager, and your code is deployed.

Testing and troubleshooting a Code Manager webhook

To test and troubleshoot your webhook, review your Git host logs or check the Code Manager troubleshooting guide.

Each of the major repository hosting services (such as GitHub or Bitbucket) provides a way to review the logs for your webhook runs, so check their documentation for instructions.

For other issues, check the Code Manager troubleshooting for some common problems and troubleshooting tips.

Triggering Code Manager with custom scripts

Custom scripts are a good way to trigger deployments if you have requirements such as existing continuous integration systems, privately hosted Git repos, or custom notifications.

Alternatively, a webhook provides a simpler way to trigger deployments automatically.

To create a script that triggers Code Manager to deploy your environments, you can use either the `puppet-code` command or a `curl` statement that hits the endpoints. We recommend the `puppet-code` command.

Either way, after you have composed your script, you can trigger deployment of new code into your environments. Commit new code, push it to your control repo, and run your script to trigger Code Manager to deploy your code.

All of these instructions assume that you have enabled and configured Code Manager.

Related information

[Request an authentication token for deployments](#) on page 563

Request an authentication token for the deployment user to enable secure deployment of your code.

[Code Manager API](#) on page 584

Use Code Manager endpoints to deploy code and query environment deployment status on your masters without direct shell access.

Deploying environments with the `puppet-code` command

The `puppet-code` command allows you to trigger environment deployments from the command line. You can use this command in your custom scripts.

For example, to deploy all environments and then return deployment results with commit SHAs for each of your environments, incorporate this command into your script:

```
puppet-code deploy --all --wait
```

Related information

[Triggering Code Manager on the command line](#) on page 572

The `puppet-code` command allows you to trigger Code Manager from the command line to deploy your environments.

Deploying environments with a `curl` command

To trigger Code Manager code deployments with a custom script, compose a `curl` command to hit the Code Manager/`deploys` endpoint.

To deploy environments with a `curl` command in your custom script, compose a `curl` command to hit the `/v1/deploys` endpoint.

Specify either the `"deploy-all"` key, to deploy all configured environments, or the `"environments"` key, to deploy a specified list of environments. By default, Code Manager returns queuing results immediately after accepting the deployment request.

If you prefer to get complete deployment results after Code Manager finishes processing the deployments, specify the optional `"wait"` key with your request. In deployments that are geographically dispersed or have a large number of environments, completing code deployment can take up to several minutes.

For complete information about Code Manager endpoints, request formats, and responses, see the Code Manager API documentation.

You must include a custom URL for Code Manager and a PE authentication token in any request to a Code Manager endpoint.

Creating a custom URL for Code Manager scripts

To trigger deployments with a `curl` command in a custom script, you'll need a custom Code Manager URL. This URL is composed of:

- Your Puppet master server name (for example, `master.example.com`)
- The Code Manager port (for example, 8170)
- The endpoint you want to hit (for example, `/code-manager/v1/deploys/`)
- The authentication token you generated earlier (`token=<TOKEN>`)

A typical URL for use with a `curl` command might look something like this:

```
https://master.example.com:8170/code-manager/v1/deploys&token=<TOKEN>
```

Attaching an authentication token

You must attach a PE authentication token to any request to Code Manager endpoints. You can either:

- Specify the path to the token with `cat ~/.puppetlabs/token` in the body of your request. For example:

```
$ curl --cacert $(puppet agent --configprint cacert) -X POST -H 'Content-Type: application/json' -H "X-Authentication: `cat ~/.puppetlabs/token`"
```

```
https://localhost:8170/code-manager/v1/deployments -d '{"environments":
["production", "testing"]}'
```

- Attach the entire token to your Code Manager URL using the token query parameter. For example:

```
https://master.example.com:8170/code-manager/v1/webhook?
type=github&token=<TOKEN>
```

If you are using a `curl` command to hit the `/webhook` endpoint directly, you must attach the entire token.

Checking deployment status with a curl command

You can check the status of a deployment by hitting the status endpoint.

To use a `curl` command in your custom script, compose a `curl` command to hit the status endpoint. You must incorporate a custom URL for Code Manager in the script.

Troubleshooting Code Manager

Code Manager requires coordination between multiple components, including `r10k` and the file sync service. If you have issues with Code Manager, check that these components are functioning.

Code Manager logs

Code Manager logs to the Puppet Server log. By default, this log is in `/var/log/puppetlabs/puppetserver/puppetserver.log`. For more information about working with the logs, see the [Puppet Server logs](#) documentation.

Check Code Manager status

Check the status of Code Manager and file sync if your deployments are not working as expected, or if you need to verify that Code Manager is enabled before running a dependent command.

The command `puppet-code status` verifies that Code Manager and file sync are responding. The command returns the same information as the Code Manager status endpoint. By default, the command returns details at the `info` level; `critical` and `debug` aren't supported.

The following table shows errors that might appear in the `puppet-code status` output.

Error	Cause
Code Manager couldn't connect to the server	<code>pe-puppetserver</code> process isn't running
Code Manager reports invalid configuration	Invalid configuration at <code>/etc/puppetlabs/puppetserver/conf.d/code-manager.conf</code>
File sync storage service reports unknown status	Status callback timed out

Test the connection to the control repository

The control repository controls the creation of environments, and ensures that the correct versions of all the necessary modules are installed. The master server must be able to access and clone the control repo as the `pe-puppet` user.

To make sure that Code Manager can connect to the control repo, run:

```
puppet-code deploy --dry-run
```

If the connection is set up correctly, this command returns a list of all environments in the control repo or repos. If the command completes successfully, the SSH key has the correct permissions, the Git URL for the repository is correct, and the `pe-puppet` user can perform the operations involved.

If the connection is not working as expected, Code Manager reports an `Unable to determine current branches for Git source` error.

The unsuccessful command also returns a path on the master that you can use for debugging the SSH key and Git URL.

Check the Puppetfile for errors

Check the Puppetfile for syntax errors and verify that every module in the Puppetfile can be installed from the listed source. To do this, you need a copy of the Puppetfile in a temporary directory.

Create a temporary directory `/var/tmp/test-puppetfile` on the master for testing purposes, and place a copy of the Puppetfile into the temporary directory.

You can then check the syntax and listed sources in your Puppetfile.

Check Puppetfile syntax

To check the Puppetfile syntax, run `r10k puppetfile check` from within the temporary directory.

If you have Puppetfile syntax errors, correct the syntax and test again. When the syntax is correct, the command prints "Syntax OK".

Check the sources listed in the Puppetfile

To test the configuration of all sources listed in your Puppetfile, perform a test installation. This test installs the modules listed in your Puppetfile into a modules directory in the temporary directory.

In the temporary directory, run the following command:

```
sudo -H -u pe-puppet bash -c \
  '/opt/puppetlabs/puppet/bin/r10k puppetfile install'
```

This installs all modules listed in your Puppetfile, verifying that you can access all listed sources. Take note of **all** errors that occur. Issues with individual modules can cause issues for the entire environment. Errors with individual modules (such as Git URL syntax or version issues) show up as general errors for that module.

If you have modules from private Git repositories requiring an SSH key to clone the module, check that you are using the SSH Git URL and not the HTTPS Git URL.

After you've fixed errors, test again and fix any further errors, until all errors are fixed.

Run a deployment test

Manually run a full r10k deployment to check not only the Puppetfile syntax and listed host access, but also whether the deployment works through r10k.

This command attempts a full r10k deployment based on the `r10k.yaml` file that Code Manager uses. This test writes to the code staging directory only. This does not trigger a file sync and is to be used only for ad-hoc testing.

Run this deployment test with the following command:

```
sudo -H -u pe-puppet bash -c \
  '/opt/puppetlabs/puppet/bin/r10k deploy environment -c /opt/puppetlabs/
server/data/code-manager/r10k.yaml -p -v debug'
```

If this command completes successfully, the `/etc/puppetlabs/code-staging` directory is populated with directory-based environments and all of the necessary modules for every environment.

If the command fails, the error is likely in the Code Manager settings specific to r10k. The error messages indicate which settings are failing.

Monitor logs for webhook deployment trigger issues

Issues that occur when a Code Manager deployment is triggered by the webhook can be tricky to isolate. Monitor the logs for the deployment trigger to find the issue.

Deployments triggered by a webhook in Stash/Bitbucket, GitLab, or GitHub are governed by authentication and hit each service's `/v1/webhook` endpoint.

If you are using a GitLab version older than 8.5.0, Code Manager webhook authentication does not work because of the length of the authentication token. To use the webhook with GitLab, either disable authentication or update GitLab.

Note: If you disable webhook authentication, it is disabled **only** for the `/v1/webhook` endpoint. Other endpoints (such as `/v1/deploys`) are still controlled by authentication. There is no way to disable authentication on any other Code Manager endpoint.

To troubleshoot webhook issues, follow the Code Manager webhook instructions while monitoring the Puppet Server log for successes and errors. To do this, open a terminal window and run:

```
tail -f /var/log/puppetlabs/puppetserver/puppetserver.log
```

Watch the log closely for errors and information messages when you trigger the deployment. The `puppetserver.log` file is the only location these errors appear.

If you cannot determine the problem with your webhook in this step, manually deploy to the `/v1/deploys` endpoint, as described in the next section.

Monitor logs for /v1/deploys deployment trigger issues

Issues that occur when a Code Manager deployment is triggered by the `/v1/deploys` endpoint can be tricky to isolate. Monitor the logs for the deployment trigger to find the issue.

Before you trigger a deployment to the `/v1/deploys` endpoint, generate an authentication token. Then deploy one or more environments with the following command:

```
curl -k -X POST -H 'Content-Type: application/json' \
-H "X-Authentication: `cat ~/.puppetlabs/token`" \
https://<URL.OF.CODE.MANAGER.SERVER>:8170/code-manager/v1/deploys \
-d '{"environments": [<ENV_NAME>], "wait": true}'
```

This request waits for the deployment and sync to compilers to complete (`"wait": true`) and so returns errors from the deployment.

Alternatively, you can deploy **all** environments with the following `curl` command:

```
curl -k -X POST -H 'Content-Type: application/json' \
-H "X-Authentication: `cat ~/.puppetlabs/token`" \
https://<URL.OF.CODE.MANAGER.SERVER>:8170/code-manager/v1/deploys \
-d '{"deploy-all": true, "wait": true}'
```

Monitor the `console-services.log` file for any errors that arise from this `curl` command.

```
tail -f /var/log/puppetlabs/console-services/console-services.log
```

Code deployments time out

If your environments are heavily loaded, code deployments can take a long time, and the system can time out before deployment is complete.

If your deployments are timing out too soon, increase your `timeouts_deploy` key. You might also need to increase `timeouts_shutdown`, `timeouts_sync`, and `timeouts_wait`.

File sync stops when Code Manager tries to deploy code

Code Manager code deployment involves accessing many small files. If you store your Puppet code on network-attached storage, slow network or backend hardware can result in poor deployment performance.

Tune the network for many small files, or store Puppet code on local or direct-attached storage.

Classes are missing after deployment

After a successful code deployment, a class you added isn't showing in the console.

If your code deployment works, but a class you added isn't in the console:

1. Check on disk to verify that the environment folder exists.
2. Check your node group in the **Edit node group metadata** box to make sure it's using the correct environment.
3. Refresh classes.
4. Refresh your browser.

Code Manager API

Use Code Manager endpoints to deploy code and query environment deployment status on your masters without direct shell access.

Authentication

All Code Manager endpoint requests require token-based authentication. For `/deploys` endpoints, you can pass this token as either an HTTP header or as a query parameter. For the `/webhook` endpoint, you must attach the entire token with the `token` query parameter.

To generate an authentication token, from the command line, run `puppet-access login`. This command stores the token in `~/puppetlabs/token`. To learn more about authentication for Code Manager use, see the related topic about requesting an authentication token.

Pass this token as either an `X-Authentication` HTTP header or as the `token` query parameter with your request. If you pass the token as a query parameter, it might show up in access logs.

For example, an HTTP header for the `/deploys` endpoint looks something like:

Bash

```
curl --cacert $(puppet agent --configprint cacert) -X POST -H 'Content-Type: application/json' \
-H "X-Authentication: `puppet access show`" \
"https://localhost:8170/code-manager/v1/deploys" -d '{"environments": [{"production"}]}'
```

PowerShell

```
$pe_headers = @{
    "X-Authentication" = "AMilWlel2Ybd2Lqu-hdhHaLWckfvgSgl8AUgKdwzixwe" ;
}

$pe_uri = "https://puppet.winops2019.automationdemos.com:8170/code-manager/v1/deploys"

$pe_codemgr_hash = [ordered]@{
    environments = @("production")
}

# Generate JSON object from pe_codemgr_hash
$JSON = $pe_codemgr_hash | convertto-json -compress
Invoke-RestMethod -SkipCertificateCheck -ContentType "application/json" -uri
$pe_uri -Method POST -Body $JSON -Headers $pe_headers
```

Passing the token with the `token` query parameter might look like:

Bash

```
curl --cacert $(puppet agent --configprint cacert) -X POST -H 'Content-Type: application/json' \
  "https://localhost:8170/code-manager/v1/webhook?type=github&token=`cat ~/.puppetlabs/token`" \
  -d $GITHUB_PAYLOAD
```

PowerShell

```
# Common Data
$token = "AMilWlel2Ybd2Lqu-hdhHaLWckfvgSgl8AUgKdwzixwe"
$pe_uri = "https://puppet.winops2019.automationdemos.com:8170/code-manager"

# Example 1 - deploys

$pe_headers = @{
    "X-Authentication" = "$token" ;
}

$pe_codemgr_hash = [ordered]@{
    environments = @("production")
}

# Generate JSON object from pe_codemgr_hash
$JSON = $pe_codemgr_hash | convertto-json -compress
$pe_ex1_uri = "$pe_uri/v1/deploys"
Invoke-RestMethod -SkipCertificateCheck -ContentType "application/json" -uri
  $pe_ex1_uri -Method POST -Body $JSON -Headers $pe_headers

# Example 2 - Web hook

$pe_ex2_uri = "$pe_uri/v1/webhook?token=$token&type=github"
Write-Output "PE Example 2 URI: $pe_ex2_uri"
Invoke-RestMethod -SkipCertificateCheck -uri $pe_ex2_uri -Method POST -
  ContentType "application/json"
```

Related information

[Request an authentication token for deployments](#) on page 563

Request an authentication token for the deployment user to enable secure deployment of your code.

[API index](#) on page 33

APIs allow you to interact with Puppet and Puppet Enterprise (PE) applications from your own code or application integration hooks.

POST /v1/deploys

Use the Code Manager/deploys endpoint to queue code deployments.

Request format

The body of the POST /deploys request must be a JSON object that includes the authentication token and a URL containing:

- The name of the master server, such as `master.example.com`
- The Code Manager port, such as 8170.
- The endpoint.

The request must also include either:

- The "environments" key, with or without the "wait" key.
- The "deploy-all" key, with or without the "wait" key.

Key	Definition	Values
"environments"	Specifies the environments for which to queue code deployments. If not specified, the "deploy-all" key must be specified.	Accepts a comma- and space-separated list of strings specifying environments.
"deploy-all"	Specifies whether Code Manager deploys all environments or not. If specified true, Code Manager determines the most recent list of environments and queues a deploy for each one. If specified false or not specified, the "environments" key must be specified with a list of environments.	true, false
"wait"	Specifies whether Code Manager must wait for all deploys to finish or error before it returns a response. If specified true, Code Manager returns a response after all deploys have either finished or errored. If specified false or not specified, returns a response showing that deploys are queued. Specify together with the "deploy-all" or "environment" keys.	true, false
"dry-run"	Tests the Code Manager connection to each of the configured remotes and attempts to fetch a list of environments from each, reporting any connection errors. Specify together with the "deploy-all" or "environment" keys.	true, false

For example, to deploy the production and testing environments and return results after the deployments complete or fail, pass the "environments" key with the "wait" key:

```
curl --cacert $(puppet agent --configprint cacert) -X POST -H 'Content-Type: application/json' \
-H "X-Authentication: `cat ~/.puppetlabs/token`" \
"https://localhost:8170/code-manager/v1/deploys" \
-d '{"environments": ["production", "testing"], "wait": true}'
```

To deploy all configured environments and return only the queuing results, pass the "deploy-all" key without the "wait" key:

```
curl --cacert $(puppet agent --configprint cacert) -X POST -H 'Content-Type: application/json' \
-H "X-Authentication: `cat ~/.puppetlabs/token`" \
"https://localhost:8170/code-manager/v1/deploys" \
-d '{"deploy-all": true}'
```

Response format

The `POST /deploys` response contains a list of objects, each containing data about a queued or deployed environment.

If the request did not include a `"wait"` key, the response returns each environment's name and status.

If the `"wait"` key was included in the request, the response returns information about each deployment. This information includes a `deploy-signature`, which is the commit SHA of the control repo that Code Manager used to deploy the environment. The output also includes two other SHAs that indicate that file sync is aware that the environment has been deployed to the code staging directory.

The response shows any deployment failures, even after they have successfully deployed. The failures remain in the response until cleaned up by garbage collection.

Key	Description	Value
"environment"	The name of the environment queued or deployed.	A string specifying the name of an environment.
"id"	Identifies the queue order of the code deploy request.	An integer generated by Code Manager.
"status"	The status of the code deployment for that environment.	Can be one of the following: <ul style="list-style-type: none"> "new": The deploy request has been accepted but not yet queued. "complete": The deploy is complete and has been synced to the live code directory on masters. "failed": The deploy failed. "queued": The deploy is queued and waiting.
"deploy-signature"	The commit SHA of the control repo that Code Manager used to deploy code in that environment.	A Git SHA.
"file-sync"	Commit SHAs used internally by file sync to identify the code synced to the code staging directory	Git SHAs for the "environment-commit" and "code-commit" keys.

For example, for a `/deploys` request without the `"wait"` key, the response shows only `"new"` or `"queued"` status, because this response is returned as soon as Code Manager accepts the request.

```
[
  {
    "environment": "production",
    "id": 1,
    "status": "queued"
  },
  {
    "environment": "testing",
    "id": 2,
    "status": "queued"
  }
]
```

If you pass the "wait" key with your request, Code Manager doesn't return any response until the environment deployments have either failed or completed. For example:

```
[
  {
    "deploy-signature": "482f8d3adc76b5197306c5d4c8aa32aa8315694b",
    "file-sync": {
      "environment-commit": "6939889b679fdb1449545c44f26aa06174d25c21",
      "code-commit": "ce5f7158615759151f77391c7b2b8b497aaebce1"
    },
    "environment": "production",
    "id": 3, Code Manager
    "status": "complete"
  }
]
```

Error responses

If any deployments fail, the response includes an error object for each failed environment. Code Manager returns deployment results only if you passed the "wait" key; otherwise, it returns queue information.

Key	Definition	Value
"environment"	The name of the environment queued or deployed.	A string specifying the name of an environment.
"error"	Information about the deployment failure.	Contains keys with error details.
"corrected-env-name"	The name of the environment.	A string specifying the name.
"kind"	The kind of error encountered.	An error type.
"msg"	The error message.	An error message.
"id"	Identifies the queue order of the code deploy request.	An integer generated by Code Manager.
"status"	The status of the requested code deployment.	For errors, status is "failed".

For example, information for a failed deploy of an environment might look like:

```
{
  "environment": "test14",
  "error": {
    "details": {
      "corrected-env-name": "test14"
    },
    "kind": "puppetlabs.code-manager/deploy-failure",
    "msg": "Errors while deploying environment 'test14' (exit code:
1):\nERROR\t -> Authentication failed for Git remote \"https://github.com/
puppetlabs/puffppetlabs-apache\".\n"
  },
  "id": 52,
  "status": "failed"
}
```

POST /v1/webhook

Use the Code Manager `/webhook` endpoint to trigger code deploys based on your control repo updates.

Request format

The `POST /webhook` request consists of a specially formatted URL that specifies the webhook type, an optional branch prefix, and a PE authentication token.

You must pass the authentication token with the `token` query parameter. To use a GitHub webhook with the Puppet signed certificate, you must disable SSL verification.

In your URL, include:

- The name of the Puppet master server (for example, `master.example.com`)
- The Code Manager port (for example, 8170)
- The endpoint (`/v1/webhook`)
- The `type` parameter with a valid value, such as `gitlab`.
- The `prefix` parameter and value.
- The `token` parameter with the complete token.

Parameter	Definition	Values
type	Required. Specifies what type of POST body to expect.	<ul style="list-style-type: none"> github gitlab tfs-git bitbucket stash
prefix	Optional. Specifies a prefix to use in translating branch names to environments.	A string specifying a branch prefix.
token	Specifies the PE authorization token to use. Required unless you disable <code>webhook_authentication</code> in Code Manager configuration.	The complete authentication token.

For example, a GitHub webhook might look like this:

`https://master.example.com:8170/code-manager/v1/webhook?type=github&token=$TOKEN`

A Stash webhook with the optional `prefix` parameter specified might look like:

```
https://master.example.com/code-manager/v1/webhook?
type=stash&prefix=dev&token=$TOKEN
```

You must attach the complete authentication token to the request. A GitHub request with the entire token attached might look like this:

```
https://master.example.com:8170/code-manager/v1/webhook?
type=github&token=eyJhbGciOiJSUzUxMiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJkZXBsY3kiLCJpYXQiOjE0MTYyOTUwMjQyLWVudC54bGRC0r2J87Pj2sDsyz-
EMK-7sZalBTswy2e3uSv1Z6ulXaIQqd69PqnSSBExQotExDgXZInyQa_le2gwTo4smjUaBd6_lnPYr6GJ4hjB4-ft8dNnVuvZae5WPkKt-
sNJkJWE9LiEd4W42aCYfselKNgpUXR5m77SRsUy86ymthVPKHqqexEyus7LGeQJvyQE1qEejSdbiLg6zn1JXhg1WNrEl7oxrrNkU0ZxioUGDeqGycwvNIamazztM9NyD-
dWmZc4dKJsqm0su0CRkMSWCYPPaeIcsYFI7XSaeC65N4RLIKhUfwIxxE-uODEhc13mTr9rwzGNVMu3WrY7t6wlbdM9FomPejGM2aJoZ05PinAIYvX3oH5OJ9fam0pVLb-
```

```
mI3bvKkm2wKAgpc4Dh1ut9sqLWkG8-
AwMA4r_oEvLwFzf8clzk34zNyPG7BvlnPle99HjQues690L-
fknSdFiXyRZeRThvZop0SWJzvUSR49etmk-
OxnMbQE4tCBWZr_khEG5jUDzeKt3PIiXdxmUaaEPHzo6Vl9XIY5
```

Response format

When you trigger your webhook by pushing changes to your control repository, you can see the response in your Git provider interface. Code Manager does not give a command line response to typical webhook usage.

If you hit the webhook endpoint directly using a curl command with a properly formatted request, valid authentication token, and a valid value for the "type" parameter, Code Manager returns a `{ "status": "OK" }` response, whether or not code successfully deployed.

Error responses

When you trigger your webhook by pushing changes to your control repository, you can see any errors in your Git provider interface. Code Manager does not give a command line response to typical webhook usage.

If you hit the webhook endpoint directly using a curl command with a request that has an incorrect "type" value or no "type" value, you get an error response such as:

```
{ "kind": "puppetlabs.code-manager/unrecognized-webhook-
type", "msg": "Unrecognized webhook type: 'githubby'", "details": "Currently
supported valid webhook types include: github gitlab stash tfs-git
bitbucket" }
```

GET /v1/deploys/status

Use the Code Manager `/deploys/status` endpoint to return the status of the code deployments that Code Manager is processing for each environment.

Request format

The body of the GET `/deploys/status` request must include the authentication token and a URL containing:

- The name of the master server, such as `master.example.com`.
- The Code Manager port, such as 8170.
- The endpoint.

```
curl --cacert $(puppet agent --configprint cacert) -X GET -H 'Content-
Type: application/json' \-H "X-Authentication: `cat ~/.puppetlabs/token`"
\ "https://localhost:8170/code-manager/v1/deploys/status"
```

Response format

The `/deploys/status` endpoint responds with a list of the code deployment requests that Code Manager is processing.

The response contains three sections:

- `"deploys-status"`: Lists the status for each code deployment that Code Manager is processing, including failed deployments. Deployments can be `"queued"`, `"deploying"`, `"new"`, or `"failed"`. Environments that have been successfully deployed to either the code staging or live code directories are displayed in the `"file-sync-storage-status"` or `"file-sync-client-status"` sections, respectively.
- `"file-sync-storage-status"`: Lists all environments that Code Manager has successfully deployed to the code staging directory, but not yet synced to the live code directory.
- `"file-sync-client-status"`: Lists status for each master that Code Manager is deploying environments to, including whether the code in the master's staging directory has been synced to its live code directory.

The response can contain the following keys:

Key	Definition	Values
"queued"	The environment is queued for deployment.	For each environment: <ul style="list-style-type: none"> "environment" "queued-at"
"deploying"	The environment is in the process of being deployed.	For each environment: <ul style="list-style-type: none"> "environment" "queued-at"
"new"	Code Manager has accepted a request to deploy the environment, but has not yet queued it.	For each environment: <ul style="list-style-type: none"> "environment" "queued-at"
"failed"	The code deployment for the environment has failed.	For each environment: <ul style="list-style-type: none"> "environment" "error" "details" "corrected-env-name" "kind" "msg" "queued-at"
"environment"	The name of the environment.	A string that is the name of the environment.
"queued-at"	The date and time when the environment was queued.	A date and time stamp.
"deployed"	Lists information for each deployed environment.	For each environment: <ul style="list-style-type: none"> "environment" "queued-at"
"date"	The date and time the operation was completed.	A date and time stamp.
"deploy-signature"	The commit SHA of the control repo used to deploy the environment.	A Git SHA.
"all-synced"	Whether all requested code deployments have been synced to the live code directories on their respective masters.	true, false
"file-sync-clients"	List of all masters that Code Manager is deploying code to.	For each environment listed: <ul style="list-style-type: none"> "last_check_in_time" "synced-with-file-sync-storage" "deployed" "environment" "date" "deploy-signature"

Key	Definition	Values
"last_check_in_time"	The most recent time that the live code directory checked in for new code from the staging directory.	A date and time stamp.
"synced-with-file-sync-storage"	Whether the live code directory has been synced with the code staging directory on a given master.	true, false

For example, a complete response to a request might look like:

```
{
  "deploys-status":{
    "queued":[
      {
        "environment":"dev",
        "queued-at":"2018-05-15T17:42:34.988Z"
      }
    ],
    "deploying":[
      {
        "environment":"test",
        "queued-at":"2018-05-15T17:42:34.988Z"
      },
      {
        "environment":"prod",
        "queued-at":"2018-05-15T17:42:34.988Z"
      }
    ],
    "new":[

    ],
    "failed":[

    ]
  },
  "file-sync-storage-status":{
    "deployed":[
      {
        "environment":"prod",
        "date":"2018-05-10T21:44:24.000Z",
        "deploy-signature":"66d620604c9465b464a3dac4884f96c43748b2c5"
      },
      {
        "environment":"test",
        "date":"2018-05-10T21:44:25.000Z",
        "deploy-signature":"24ecc7bac8a4d727d6a3a2350b6fda53812ee86f"
      },
      {
        "environment":"dev",
        "date":"2018-05-10T21:44:21.000Z",
        "deploy-signature":"503a335c99a190501456194d13ff722194e55613"
      }
    ]
  },
  "file-sync-client-status":{
    "all-synced":false,
    "file-sync-clients":{
      "chihuahua":{
        "last_check_in_time":null,
        "synced-with-file-sync-storage":false,
        "deployed":[

```



```

    ]
  },
  "localhost": {
    "last_check_in_time": "2018-05-11T22:41:20.270Z",
    "syncd-with-file-sync-storage": true,
    "deployed": [
      {
        "environment": "prod",
        "date": "2018-05-11T22:40:48.000Z",
        "deploy-
signature": "66d620604c9465b464a3dac4884f96c43748b2c5"
      },
      {
        "environment": "test",
        "date": "2018-05-11T22:40:48.000Z",
        "deploy-
signature": "24ecc7bac8a4d727d6a3a2350b6fda53812ee86f"
      },
      {
        "environment": "dev",
        "date": "2018-05-11T22:40:50.000Z",
        "deploy-
signature": "503a335c99a190501456194d13ff722194e55613"
      }
    ]
  }
}
}
}
}

```

Error responses

If code deployment fails, the "deploys-status" section of the response provides an "error" key for the environment with the failure. The "error" key contains information about the failure.

Deployment failures can remain in the response even after the environment in question is successfully deployed. Old failures are removed from the "deploys-status" response during garbage collection.

Key	Definition	Value
"environment"	The name of the environment queued or deployed.	A string specifying the name of an environment.
"error"	Information about the deployment failure.	Contains keys with error details.
"corrected-env-name"	The name of the environment.	A string specifying the name.
"kind"	The kind of error encountered.	An error type.
"msg"	The error message.	An error message.
"queued-at"	The date and time when the environment was queued.	A date and time stamp.

For example, with a failure, the "deploys-status" response section might look something like:

```

[
  {
    "deploys-status": {
      "deploying": [],
      "failed": [
        {
          "environment": "test14",

```

```

      "error": {
        "details": {
          "corrected-env-name": "test14"
        },
        "kind": "puppetlabs.code-manager/deploy-failure",
        "msg": "Errors while deploying environment
'test14' (exit code: 1):\nERROR\t -> Authentication failed for Git remote
\"https://github.com/puppetlabs/puffppetlabs-apache\".\n"
      },
      "queued-at": "2018-06-01T21:28:18.292Z"
    }
  ],
  "new": [],
  "queued": []
},
...
}
]

```

Managing code with r10k

r10k is a code management tool that allows you to manage your environment configurations (such as production, testing, and development) in a source control repository.

Based on the code in your control repo branches, r10k creates environments on your master and installs and updates the modules you want in each environment. If you are running PE 2015.3 or later, we encourage you to use Code Manager, which uses r10k in the background to automate the deployment of your code. If you use Code Manager, you won't need to manage or interact with r10k manually. To learn more about Code Manager and begin setup, see the Code Manager page. However, if you're already using r10k and aren't ready to switch to Code Manager, you can continue using r10k alone.

To set up r10k to manage your environments:

1. Set up a control repository for your code.
2. Create a Puppetfile to manage the content for your environment.
3. Configure r10k. Optionally, you can also customize your r10k configuration in Hiera.
4. Run r10k to deploy your environments and modules.

We've also included a reference of r10k subcommands.

Configuring r10k

When performing a fresh text-mode installation of Puppet Enterprise (PE), you can configure r10k by adding parameters to the `pe.conf` file. In existing installations, configure r10k by adjusting parameters in the console.



CAUTION: Do not edit the r10k configuration file manually. Puppet manages this configuration file automatically and undoes any manual changes you make.

Related information

[Install using text install](#) on page 136

When you run the installer in text mode, you provide a configuration file (`pe.conf`) to the installer. The `pe.conf` file contains values for the parameters needed for installation.

[Configuration parameters and the pe.conf file](#) on page 138

A `pe.conf` file is a HOCON formatted file that declares parameters and values needed to install, upgrade, and configure PE.

Upgrading from previous versions of r10k

If you used r10k prior to PE version 2015.3, you might have configured it in the console using the `pe_r10k` class. We suggest configuring r10k in the master profile class, and then customizing your configuration as needed in Hiera.

This simplifies configuration, and makes it easier to move to Code Manager in the future.

To switch to master profile class configuration, remove the `pe_r10k` class in the console, and then configure r10k as described in the topic about configuring r10k after PE installation. You can then customize your configuration in Hiera if needed.

Note: If you were using earlier versions of r10k with the `zack-r10k` module, discontinue use of the module and switch to the master profile configuration as above.

Configure r10k during installation

To set up r10k during PE installation, add the r10k parameters to `pe.conf` *before* starting installation. This is the easiest way to set up r10k with a new PE installation.

Before you begin

Ensure that you have a Puppetfile and a control repo. You also need the SSH private key that you created when you made your control repo.

Restriction: This configuration method works only with text-based installation. If you are using the web-based installer, see the related topic about configuring r10k after PE installation.

1. Add `puppet_enterprise::profile::master::r10k_remote` to `pe.conf`.

This setting specifies the location of the control repository. It accepts a string that is a valid URL for your Git control repository.

```
"puppet_enterprise::profile::master::r10k_remote":
  "git@<YOUR.GIT.SERVER.COM>:puppet/control.git"
```

2. Add `puppet_enterprise::profile::master::r10k_private_key` to `pe.conf`.

This setting specifies the path to the file that contains the SSH private key used to access your Git repositories. This location for the SSH private key file, which you created when you set up your control repository, **must** be located on the Puppet master and owned by and accessible to the `pe-puppet` user. The setting accepts a string.r10k

```
"puppet_enterprise::profile::master::r10k_private_key": "/etc/puppetlabs/
puppetserver/ssh/id-control_repo.rsa"
```

3. Complete PE installation. The installer configures r10k for you. You can change the values for the remote and the private key as needed in the master profile settings in the console.
4. After PE installation is complete, place the SSH private key you created when you set up your control repository in the `r10k_private_key` location.
5. Run r10k. PE does not automatically run r10k.

Configure r10k after PE installation

To configure r10k in an existing PE, set r10k parameters in the console. You can also adjust r10k settings in the console.

Before you begin

Ensure that you have a Puppetfile and a control repo. You also need the SSH private key that you created when you made your control repo.

1. In the console, set the following parameters in the `puppet_enterprise::profile::master` class in the **PE Master** node group:

- `r10k_remote`

This is the location of your control repository. Enter a string that is a valid URL for your Git control repository, such as `"git@<YOUR.GIT.SERVER.COM>:puppet/control.git"`.

- `r10k_private_key`

This is the path to the private key that permits the `pe-puppet` user to access your Git repositories. This file must be located on the Puppet master, owned by the `pe-puppet` user, and in a directory that the `pe-puppet` user has permission to view. We recommend `/etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa`. Enter a string, such as `" /etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa"`.

2. Run Puppet on all of your masters.

You can now customize your `r10k` configuration in Hiera, if needed. After `r10k` is configured, you can deploy your environments from the command line. PE does not automatically run `r10k` at the end of installation.

Customizing r10k configuration

If you need a customized `r10k` configuration, you can set specific parameters with Hiera.

To customize your configuration, add keys to your control repository hierarchy in the `hieradata/common.yaml` file in the format `pe_r10k::<parameter>`.

In this example, the first parameter specifies where to store the cache of your Git repos, and the second sets where the source repository should be fetched from.

```
pe_r10k::cachedir: /var/cache/r10k
pe_r10k::remote: git://git-server.site/my-org/main-modules
```

Remember: After changing any of these settings, run `r10k` on the command line to deploy your environments. PE does not automatically run `r10k` at the end of installation.

Related information

[Configure settings with Hiera](#) on page 190

Hiera is a hierarchy-based method of configuration management that relies on a “defaults, with overrides” system. When you add a parameter or setting to your Hiera data, Hiera searches through the data in the order it was written to find the value you want to change. Once found, it overrides the default value with the new parameter or setting. You can use Hiera to manage your PE configuration settings.

Add r10k parameters in Hiera

To customize your `r10k` configuration, add parameters to your control repository hierarchy in the `hieradata/common.yaml` file.

1. Add the parameter in the `hieradata/common.yaml` file in the format `pe_r10k::<parameter>`.
2. Run Puppet on the master to apply changes.

Configuring Forge settings

To configure how `r10k` downloads modules from the Forge, specify `forge_settings` in Hiera.

This parameter configures where modules should be installed from, and sets a proxy for all interactions. The `forge_settings` parameter accepts a hash with the following values:

- `baseurl`
- `proxy`

baseurl

Indicates where modules should be installed from. Defaults to `https://forgeapi.puppetlabs.com`.

```
pe_r10k::forge_settings:
  baseurl: 'https://private-forge.mysite'
```

proxy

Sets the proxy for all interactions.

This setting overrides the global `proxy` setting on operations only. You can set an unauthenticated proxy or an authenticated proxy with either Basic or Digest authentication. See the global `proxy` setting for more information and examples.

```
proxy: 'http://proxy.example.com:3128'
```

Configuring purge levels

The `purge_levels` setting controls which unmanaged content r10k purges during a deployment.

This setting accepts an array of strings specifying what content r10k purges during code deployments. Available values are:

- `deployment`
- `environment`
- `puppetfile`
- `purge_whitelist`

The default value for this setting is `['deployment', 'puppetfile']`. With these values, r10k purges:

- Any content that is not managed by the sources declared in the `remote` or `sources` settings.
- Any content that is not declared in the Puppetfile, but is found in a directory managed by the Puppetfile.

```
deploy:
  purge_levels: [ 'deployment', 'environment', 'puppetfile' ]
```

deployment

After each deployment, in the configured `basedir`, r10k recursively removes content that is not managed by any of the sources declared in the `remote` or `sources` parameters.

Disabling this level of purging could cause the number of deployed environments to grow without bound, because deleting branches from a control repo would no longer cause the matching environment to be purged.

environment

After a given environment is deployed, r10k recursively removes content that is neither committed to the control repo branch that maps to that environment, nor declared in a Puppetfile committed to that branch.

With this purge level, r10k loads and parses the Puppetfile for the environment even if the `--puppetfile` flag is not set. This allows r10k to check whether or not content is declared in the Puppetfile. However, Puppetfile content is deployed only if the environment is new or the `--puppetfile` flag is set.

If r10k encounters an error while evaluating the Puppetfile or deploying its contents, no environment-level content is purged.

puppetfile

After Puppetfile content for a given environment is deployed, r10k recursively removes content in any directory managed by the Puppetfile, if that content is not also declared in that Puppetfile.

Directories considered to be managed by a Puppetfile include the configured `moduledir` (which defaults to "modules"), as well as any alternate directories specified as an `install_path` option to any Puppetfile content declarations.

purge_whitelist

The `purge_whitelist` setting exempts the specified filename patterns from being purged.

This setting affects environment level purging only. Any given value must be a list of shell style filename patterns in string format.

See the [Ruby documentation](#) for the `fnmatch` method for more details on valid patterns. Both the `FNM_PATHNAME` and `FNM_DOTMATCH` flags are in effect when `r10k` considers the whitelist.

Patterns are relative to the root of the environment being purged and, by default, **do not match recursively**. For example, a whitelist value of `*myfile*` would preserve only a matching file at the root of the environment. To preserve the file throughout the deployed environment, you would need to use a recursive pattern such as `**/*myfile*`.

Files matching a whitelist pattern might still be removed if they exist in a folder that is otherwise subject to purging. In this case, use an additional whitelist rule to preserve the containing folder.

```
deploy:
  purge_whitelist: [ 'custom.json', '**/*.xpp' ]
```

Locking r10k deployments

The `deploy: write_lock` setting allows you to temporarily disallow `r10k` code deploys without completely removing the `r10k` configuration.

This `deploy` subsetting is useful to prevent `r10k` deployments at certain times or to prevent deployments from interfering with a common set of code that might be touched by multiple `r10k` configurations.

```
deploy:
  write_lock: "Deploying code is disallowed until the next maintenance
window."
```

Configuring sources

If you are managing more than one repository with `r10k`, specify a map of your source repositories.

Use the `source` parameter to specify a map of sources. Configure this setting if you are managing more than just Puppet environments, such as when you are also managing Hiera data in its own control repository.

To use multiple control repos, the `sources` setting and the `repositories` setting must match.

If `sources` is set, you cannot use the global `remote` and `r10k_basedir` settings. Note that `r10k` raises an error if different sources collide in a single base directory. If you are using multiple sources, use the `prefix` option to prevent collisions.

This setting accepts a hash with the following subsettings:

- `remote`
- `basedir`
- `prefix`
- `invalid_branches`

```
myorg:
  remote: "git://git-server.site/myorg/main-modules"
  basedir: "/etc/puppetlabs/puppet/environments"
  prefix: true
  invalid_branches: 'error'
mysource:
  remote: "git://git-server.site/mysource/main-modules"
```

```

basedir: "/etc/puppetlabs/puppet/environments"
prefix: "testing"
invalid_branches: 'correct_and_warn'

```

remote

Specifies where the source repository should be fetched from. The remote must be able to be fetched without any interactive input. That is, you cannot be prompted for usernames or passwords in order to fetch the remote.

Accepts a string that is any valid URL that r10k can clone, such as:

```
git://git-server.site/my-org/main-modules
```

basedir

Specifies the path where environments are created for this source. This directory is entirely managed by r10k, and any contents that r10k did not put there are removed.

Note that the `basedir` setting **must match** the `environmentpath` in your `puppet.conf` file, or Puppet won't be able to access your new directory environments.

prefix

Allows environment names to be prefixed with this string. Alternatively, if `prefix` is set to `true`, the source's name is used. This prevents collisions when multiple sources are deployed into the same directory.

In the `sources` example above, two "main-modules" environments are set up in the same base directory. The `prefix` setting is used to differentiate the environments: the first is named "myorg-main-modules", and the second is "testing-main-modules".

ignore_branch_prefixes

Causes branches from a source which match any of the prefixes listed in the setting to be ignored. The match can be full or partial. On deploy, each branch in the repo has its name tested against all prefixes listed in `ignore_branch_prefixes` and, if the prefix is found, then an environment is not deployed for this branch.

The setting is a list of strings. In the following example, branches in "mysource" with the prefixes "test" and "dev" are not deployed.

```

---
sources:
  mysource:
    basedir: '/etc/puppet/environments'
    ignore_branch_prefixes:
      - 'test'
      - 'dev'

```

If `ignore_branch_prefixes` is not specified, then all branches are deployed.

invalid_branches

Specifies how branch names that cannot be cleanly mapped to Puppet environments are handled. This option accepts three values:

- 'correct_and_warn' is the default value and replaces non-word characters with underscores and issues a warning.
- 'correct' replaces non-word characters with underscores without warning.
- 'error' ignores branches with non-word characters and issues an error.

Configuring the r10k base directory

The r10k base directory specifies the path where environments are created for this source.

This directory is entirely managed by r10k, and any contents that r10k did not put there are removed. If `r10k_basedir` is not set, it uses the default `environmentpath` in your `puppet.conf` file. If `r10k_basedir` is set, you cannot set the `sources` parameter.

Note that the `r10k_basedir` setting **must match** the `environmentpath` in your `puppet.conf` file, or Puppet won't be able to access your new directory environments.

Accepts a string, such as: `/etc/puppetlabs/code/environments`.

Configuring r10k Git settings

To configure r10k to use a specific Git provider, a private key, a proxy, or multiple repositories with Git, specify the `git_settings` parameter.

The r10k `git_settings` parameter accepts a hash of the following settings:

- `private_key`
- `proxy`
- `repositories`
- `provider`

Contains settings specific to Git. Accepts a hash of values, such as:

```
pe_r10k::git_settings:
  provider: "rugged"
  private_key: "/etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa"
  username: "git"
```

private_key

Specifies the file containing the default private key used to access control repositories, such as `/etc/puppetlabs/puppetserver/ssh/id-control_repo.rsa`. This file must have read permissions for the `pe-puppet` user. The SSH key cannot require a password. This setting is required, but by default, the value is supplied by the master profile in the console.

proxy

Sets a proxy specifically for operations that use an HTTP(S) transport.

This setting overrides the global `proxy` setting on operations only. You can set an unauthenticated proxy or an authenticated proxy with either Basic or Digest authentication. To set a proxy for a specific repository only, set `proxy` in the `repositories` subsetting of `git_settings`. See the global `proxy` setting for more information and examples.

```
proxy: 'http://proxy.example.com:3128'
```

repositories

Specifies a list of repositories and their respective private keys. Use this setting if you want to use multiple control repos.

To use multiple control repos, the `sources` setting and the `repositories` setting must match. Accepts an array of hashes with the following keys:

- `remote`: The repository these settings should apply to.
- `private_key`: The file containing the private key to use for this repository. This file must have read permissions for the `pe-puppet` user. This setting overrides the global `private_key` setting.

- `proxy`: The `proxy` setting allows you to set or override the global proxy setting for a single, specific repository. See the global proxy setting for more information and examples.

```
pe_r10k::git_settings:
  repositories:
    - remote: "ssh://tessier-ashpool.freeside/protected-repo.git"
      private_key: "/etc/puppetlabs/r10k/ssh/id_rsa-protected-repo-deploy-
key"
    - remote: "https://git.example.com/my-repo.git"
      proxy: "https://proxy.example.com:3128"
```

username

Optionally, specify a username when the Git remote URL does not provide a username.

Configuring proxies

To configure proxy servers, use the `proxy` setting. You can set a global proxy for all HTTP(S) operations, for all Git or Forge operations, or for a specific Git repository only.

proxy

To set a proxy for all operations occurring over an HTTP(S) transport, set the global `proxy` setting. You can also set an authenticated proxy with either Basic or Digest authentication.

To override this setting for `or` operations only, set the `proxy` subsetting under the `git_settings` or `forge_settings` parameters. To override for a specific repository, set a proxy in the `repositories` list of `git_settings`. To override this setting with no proxy for `,` `,` or a particular repository, set that specific `proxy` setting to an empty string.

In this example, the first proxy is set up without authentication, and the second proxy uses authentication:

```
proxy: 'http://proxy.example.com:3128'
proxy: 'http://user:password@proxy.example.com:3128'
```

r10k proxy defaults and logging

By default, `r10k` looks for and uses the first environment variable it finds in this list:

- `HTTPS_PROXY`
- `https_proxy`
- `HTTP_PROXY`
- `http_proxy`

If no proxy setting is found in the environment, the global `proxy` setting defaults to no proxy.

The proxy server used is logged at the "debug" level when `r10k` runs.

Configuring postrun commands

To configure a command to be run after all deployment is complete, add the `postrun` parameter.

The `postrun` optional command to be run after `r10k` finishes deploying environments. The command must be an array of strings to be used as an argument vector. You can set this parameter only one time.

```
postrun: [ '/usr/bin/curl', '-F', 'deploy=done', 'http://my-app.site/
endpoint' ]
```

r10k parameters

The following parameters are available for r10k. Parameters are optional unless otherwise stated.

Parameter	Description	Value	Default
cachedir	Specifies the path to the directory where you want the cache of your Git repos to be stored.	A valid directory path	<code>/var/cache/r10k</code>
deploy	Controls how r10k code deployments behave. See Configuring purge levels and Locking deployments for usage details.	Accepts settings for: <ul style="list-style-type: none"> <code>purge_level</code> <code>write_lock</code> 	<ul style="list-style-type: none"> <code>purge_level</code>: <code>['deployment', 'puppetfile']</code> <code>write_lock</code>: Not set by default.
forge_settings	Contains settings for downloading modules from the Puppet Forge. See Configuring Forge settings for usage details.	Accepts a hash of: <ul style="list-style-type: none"> <code>baseurl</code> <code>proxy</code> 	No default.
git_settings	Configures Git settings: Git provider, private key, proxies, and repositories. See Configuring Git settings for usage details.	Accepts a hash of: <ul style="list-style-type: none"> <code>provider</code> <code>private_key</code> <code>proxy</code> <code>repositories</code> 	No default.
proxy	Configures a proxy server to use for all operations that occur over an HTTP(S) transport. See Configuring proxies for usage details.	Accepts a string specifying a URL to proxy server, without authentication, or with Basic or Digest authentication.	No default set.
postrun	An optional command to be run after r10k finishes deploying environments.	An array of strings to use as an argument vector.	No default set.
remote	Specifies where the source repository should be fetched from. The remote cannot require any prompts, such as for usernames or passwords. If <code>remote</code> is set, you cannot use <code>sources</code> .	Accepts a string that is any valid SSH URL that r10k can clone.	By default, uses the <code>r10k_remote</code> value set in the console.
r10k_basedir	Specifies the path where environments are created for this source. See Configuring the base directory for usage details.	A valid file path, which must match the value of <code>environmentpath</code> in your <code>puppet.conf</code> file.	Uses the value of the <code>environmentpath</code> in your <code>puppet.conf</code> file.

Parameter	Description	Value	Default
<code>sources</code>	Specifies a map of sources to be passed to <code>r10k</code> . Use if you are managing more than just Puppet environments. See Configuring sources for usage details.	A hash of: <ul style="list-style-type: none"> <code>basedir</code> <code>remote</code> <code>prefix</code> <code>ignore_branch_prefixes</code> 	No default.

Deploying environments with `r10k`

Deploy environments on the command line with the `r10k` command.

The first time you run `r10k`, deploy all environments and modules by running `r10k deploy environment`.

This command:

1. Checks your control repository to see what branches are present.
2. Maps those branches to the Puppet directory environments.
3. Clones your Git repo, and either creates (if this is your first run) or updates (if it is a subsequent run) your directory environments with the contents of your repo branches.

Note:

When running commands to deploy code on a master, `r10k` needs write access to your environment path. Run `r10k` as the `pe-puppet` user or as root. Running the user as root requires access control to the root user.

Updating environments

To update environments with `r10k`, use the `deploy environment` command.

Updating all environments

The `deploy environment` command updates all existing environments and recursively creates new environments.

The `deploy environment` command updates all existing environments and recursively creates new environments. This command updates modules only on the first deployment of a given environment. On subsequent updates, it updates only the environment itself.

From the command line, run `r10k deploy environment`

Updating all environments and modules

With the `--puppetfile` flag, the `deploy environment` command updates all environments and modules.

This command:

- Updates all sources.
- Creates new environments.
- Deletes old environments.
- Recursively updates all environment modules specified in each environment's Puppetfile.

This command does the maximum possible work, and is therefore the slowest method for `r10k` deployments. Usually, you should use the less resource-intensive commands for updating environments and modules.

From the command line, run `r10k deploy environment --puppetfile`

Updating a single environment

To update just one environment, specify that environment name with the `deploy environment` command.

This command updates modules only on the first deployment of the specified environment. On subsequent updates, it updates only the environment itself.

If you're actively developing on a given environment, this is the quickest way to deploy your changes.

On the command line, run `r10k deploy environment <MY_WORKING_ENVIRONMENT>`

Updating a single environment and its modules

To update both a single given environment and all of the modules contained in that environment's Puppetfile, add the `--puppetfile` flag to your command. This is useful if you want to make sure that a given environment is fully up to date.

On the command line, run `r10k deploy environment <MY_WORKING_ENVIRONMENT> --puppetfile`

Installing and updating modules

To update modules, use the `r10k deploy module` subcommand. This command installs or updates the modules you've specified in each environment's Puppetfile.

If the specified module is not described in a given environment's Puppetfile, that environment is skipped.

Updating specific modules across all environments

To update specific modules across all environments, specify the modules with the `deploy module` command.

You can specify a single module or multiple modules. This is useful when you're working on a module and want to update it across all environments.

For example, to update the `apache`, `jenkins`, and `java` modules, run `r10k deploy module apache jenkins java`

Updating one or more modules in a single environment

To update specific modules in a single environment, specify both the environment and the modules.

On the command line, run `r10k deploy module` with:

- The environment option `-e`.
- The name of the environment.
- The names of the modules you want to update in that environment.

For example, to install the `apache`, `jenkins`, and `java` modules in the `production` environment, run `r10k deploy module -e production apache jenkins java`

Viewing environments with r10k

Display information about your environments and modules with the `r10k deploy display` subcommand. This subcommand does not deploy environments, but only displays information about the environments and modules `r10k` is managing.

This command can return varying levels of detail about the environments.

- To display all environments being managed by `r10k`, use the `deploy display` command. From the command line, run `r10k deploy display`
- To display all managed environments and Puppetfile modules, use the Puppetfile flag, `-p`. From the command line, run `r10k deploy display -p`
- To get detailed information about module versions, use the `-p` (Puppetfile) and `--detail` flags. This provides both the expected and actual versions of the modules listed in each environment's Puppetfile. From the command line, run `r10k deploy display -p --detail`
- To get detailed information about the modules only in specific environments, limit your query with the environment names. This provides both the expected and actual versions of the modules listed in the Puppetfile in each specified environment. For example, to see details on the modules for environments called `production`, `vmwr`, and `webrefactor`, run:

```
r10k deploy display -p --detail production vmwr webrefactor
```

r10k command reference

r10k accepts several command line actions, with options and subcommands.

r10k command actions

r10k includes several command line actions.

r10k command	Action
deploy	Performs operations on environments. Accepts deploy subcommands listed below.
help	Displays the r10k help page in the terminal.
puppetfile	Performs operations on a Puppetfile. Accepts puppetfile subcommands.
version	Displays your r10k version in the terminal.

```
r10k deploy
```

r10k command options

r10k commands accept the following options.

r10k command options	Action
--color	Enables color coded log messages.
--help, -h	Shows help for this command.
--trace, -t	Displays stack traces on application crash.
--verbose, -v	Sets log verbosity. Valid values: fatal, error, warn, notice, info, debug, debug1, debug2.

r10k deploy subcommands

You can use the following subcommands with the r10k deploy command.

```
r10k deploy --display
```

deploy subcommand	Action
display	Displays a list of environments in your deployment.
display -p	Displays a list of modules in the Puppetfile.
display --detail	Displays detailed information.
display --fetch	Update the environment sources. Allows you to check for missing environments.
environment	Deploys environments and their specified modules.
environment -p	Updates modules specified in the environment's Puppetfile.
module	Deploys a module in all environments.
module -e	Updates all modules in a particular environment.
no-force	Prevents the overwriting of local changes to Git-based modules.

See the related topic about deploying environments with `r10k` for examples of `r10k deploy` command use.

r10k puppetfile subcommands

The `r10k puppetfile` command accepts several subcommands for managing modules.

```
r10k puppetfile install
```

These subcommands must be run as the user with write access to that environment's modules directory. These commands interact with the Puppetfile in the current working directory, so before running the subcommand, make sure you are in the directory of the Puppetfile you want to use.

puppetfile subcommand	Action
<code>check</code>	Verifies the Puppetfile syntax is correct.
<code>install</code>	Installs all modules from a Puppetfile.
<code>install --puppetfile</code>	Installs modules from a custom Puppetfile path.
<code>install --moduledir</code>	Installs modules from a Puppetfile to a custom module directory location.
<code>install --update_force</code>	Installs modules from a Puppetfile with a forced overwrite of local changes to Git-based modules.
<code>purge</code>	Purges unmanaged modules from a Puppetfile managed directory.

About file sync

File sync helps Code Manager keep your Puppet code synchronized across multiple masters.

When triggered by a web endpoint, file sync takes changes from your working directory on your master and deploys the code to a live code directory. File sync then automatically deploys that code onto all your compilers, ensuring that all masters in a multi-master configuration are kept in sync.

In addition, file sync ensures that your Puppet code is deployed only when it is ready. These deployments ensure that your agents' code won't change during a run. File sync also triggers an environment cache flush when the deployment has finished, to ensure that new agent runs happen against the newly deployed Puppet code.

File sync works with Code Manager, so you typically won't need to do anything with file sync directly. If you want to know more about how file sync works, or you need to work with file sync directly for testing, this page provides additional information.

File sync terms

There are a few terms that are helpful when you are working with file sync or tools that use file sync.

Live code directory

This is the directory that all of the Puppet environments, manifests, and modules are stored in. This directory is used by Puppet Server for catalog compilation. It corresponds to the Puppet Server `master-code-dir` setting and the `Puppet$codedir` setting. The default value is `/etc/puppetlabs/code`. In file sync configuration, you might see this referred to simply as `live-dir`. This directory exists on all of your masters.

Staging code directory

This is the directory where you stage your code changes before rolling them out to the live code dir. You can move files into this directory in your usual way. Then, when you trigger a file sync deployment, file sync moves the changes to the live code dir on all of your masters. This directory exists only on the master; compilers do not need a staging directory. The default value is `/etc/puppetlabs/code-staging`.

How file sync works

File sync helps distribute your code to all of your masters and agents.

By default, file sync is disabled and the staging directory is not created on the master. If you're upgrading from 2015.2 or earlier, file sync is disabled after the upgrade. You must enable file sync, and then run Puppet on all masters. This creates the staging directory on the master, which you can then populate with your Puppet code. File sync can then commit your code; that is, it can prepare the code for synchronization to the live code directory, and then your compilers. Normally, Code Manager triggers this commit automatically, but you can trigger a commit by hitting the file sync endpoint.

For example:

```
/opt/puppetlabs/puppet/bin/curl -s --request POST --header "Content-Type:
application/json" --data '{"commit-all": true}' --cert ${cert} --key ${key}
--cacert ${cacert} https://${fqdn}:8140/file-sync/v1/commit"
```

The above command is run from the master and contains the following variables:

- `fqdn`: Fully qualified domain name of the master.
- `cacert`: The Puppet CA's certificate (`/etc/puppetlabs/puppet/ssl/certs/ca.pem`).
- `cert`: The ssl cert for the master (`/etc/puppetlabs/ssl/certs/${fqdn}.pem`).
- `key`: The private key for the master (`/etc/puppetlabs/ssl/private_keys/${fqdn}.pem`).

This command commits all of the changes in the staging directory. After the commit, when any compilers check the file sync service for changes, they receive the new code and deploy it into their own code directories, where it is available for agents checking in to those masters. (By default, compilers check file sync every 5 seconds.)

Commits can be restricted to a specific environment and can include details such as a message, and information about the commit author.

Enabling or disabling file sync

File sync is normally enabled or disabled automatically along with Code Manager.

File sync's behavior is linked to that of Code Manager. Because Code Manager is disabled by default, file sync is also disabled. To enable file sync, enable Code Manager. You can enable and configure Code Manager either during or after PE installation.

The `file_sync_enabled` parameter in the `puppet_enterprise::profile::master` class in the console defaults to `automatic`, which means that file sync is enabled and disabled automatically with Code Manager. If you set this parameter to `true`, it forces file sync to be enabled even if Code Manager is disabled. The `file_sync_enabled` parameter doesn't appear in the class definitions --- you must add the parameter to the class in order to set it.

Resetting file sync

If file sync has entered a failure state, consumed all available disk space, or a repository has become irreparably corrupted, reset the service.

Resetting deletes the commit history for all repositories managed by file sync, which frees up disk space and returns the service to a "fresh install" state while preserving any code in the staging directory.

1. On the master, perform the appropriate action:

- If you use file sync with Code Manager, ensure that any code ready for deployment is present in the staging directory, and that the code most recently deployed is present in your control repository so that it can be re-synced.
- If you use file sync with r10k, perform an r10k deploy and ensure that the code most recently deployed is present in your control repository so that it can be re-synced.
- If you use file sync alone, ensure that any code ready for deployment is present in `/etc/puppetlabs/code-staging`.

2. Shut down the pe-puppetserver service: `puppet resource service pe-puppetserver ensure=stopped.`
3. Delete the data directory located at `/opt/puppetlabs/server/data/puppetserver/filesync/storage.`
4. Restart the pe-puppetserver service: `puppet resource service pe-puppetserver ensure=running.`
5. Perform the appropriate action:
 - If you use file sync with Code Manager, use Code Manager to deploy all environments.
 - If you use file sync alone or with r10k, perform a commit.

File sync is now reset. The service creates fresh repositories on each client and the storage server for the code it manages.

Checking your deployments

You can manually check information about file sync's deployments with curl commands that hit the `status` endpoint.

To manually check that your code has been successfully committed and deployed, you can hit a status endpoint on the master.

```
curl -k https://{fqdn}:8140/status/v1/services?level=debug
```

This returns output in JSON, so you can pipe it to `python -m json.tool` for better readability.

To check a list of file sync's clients, query the `file-sync-storage-service` section of the master by running a curl command.

This command returns a list of:

- All of the clients that file sync is aware of.
- When those clients last checked in.
- Which commit they have deployed .

```
curl -k https://{fqdn}:8140/status/v1/services/file-sync-client-service?level=debug
```

If your commit has been deployed, it is listed in the status listing for `latest_commit`.

Cautions

There are a few things to be aware of with file sync.

Always use the staging directory

Always make changes to your Puppet code in the staging directory. If you have edited code in the live code directory on the master or any compiled masters, *it's overwritten* by file sync on the next commit.

The `enable-forceful-sync` parameter is set to `true` by default in PE. When `false`, file sync does not overwrite changes in the code directory, but instead logs errors in `/var/log/puppetlabs/puppetserver/puppetserver.log`. To set this parameter to `false`, add via Hiera (`puppet_enterprise::master::file_sync::file_sync_enable_forceful_sync: false`).

The puppet module command and file sync

The `puppet module` command doesn't work with file sync. If you are using file sync, specify modules in the Puppetfile and use Code Manager to handle your syncs.

Permissions

File sync runs as the `pe-puppet` user. To sync files, file sync **must** have permission to read the staging directory and to write to all files and directories in the live code directory. To make sure file sync can read and write what it needs to, ensure that these code directories are both owned by the `pe-puppet` user:

```
chown -R pe-puppet /etc/puppetlabs/code /etc/puppetlabs/code-staging
```

Environment isolation metadata

File sync generates `.pp` metadata files in both your live and staging code directories. These files provide environment isolation for your resource types, ensuring that each environment uses the correct version of the resource type. Do not delete or modify these files. Do not use expressions from these files in regular manifests.

For more details about these files and how they isolate resource types in multiple environments, see [environment isolation](#).

Provisioning with Razor

Razor is a provisioning application that deploys bare-metal systems.

Policy-based provisioning lets you use characteristics of the hardware as well as user-provided data to make provisioning decisions. You can automatically discover bare-metal hardware, dynamically configure operating systems and hypervisors, and hand off nodes to Razor for workload configuration.

Automated provisioning makes Razor ideal for big installation jobs, like setting up a new selection of servers in a server farm. You can also use Razor to regularly wipe and re-provision test machines.

How Razor works

There are five key steps for provisioning nodes with Razor.

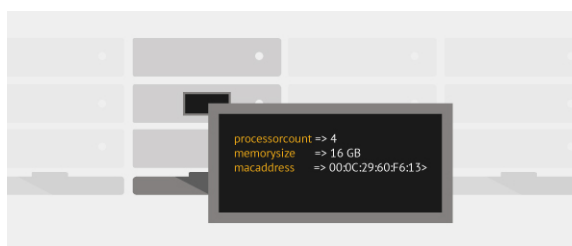


Figure 1: Razor identifies a node

When a new node appears, Razor discovers its characteristics by booting it with the Razor microkernel and using `Factor` to inventory its facts.

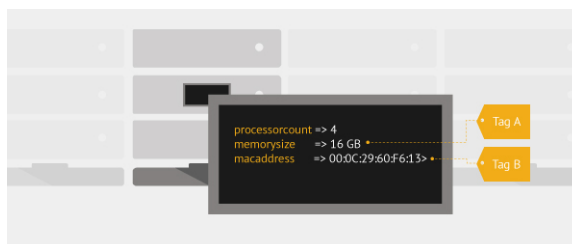


Figure 2: The node is tagged

The node is tagged based on its characteristics. Tags contain a match condition — a Boolean expression that has access to the node's facts and determines whether the tag is to be applied to the node or not.

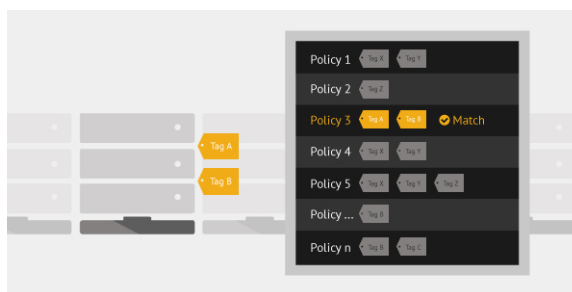


Figure 3: The node tags match a Razor policy

Node tags are compared to tags in the policy table. The first policy with tags that match the node's tags is applied to the node.

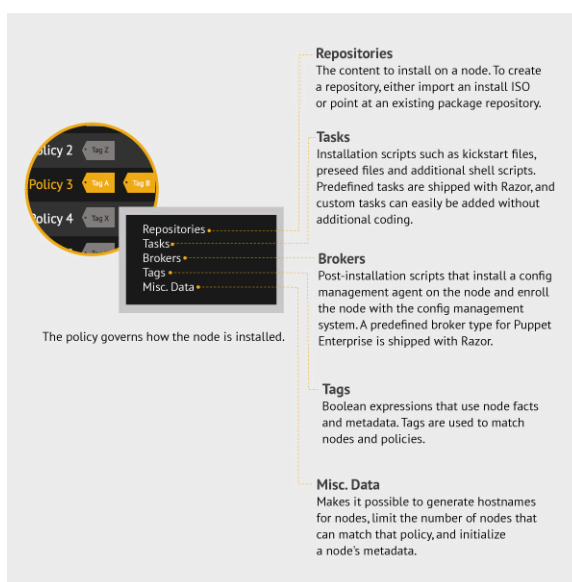


Figure 4: Policies pull together all the provisioning elements

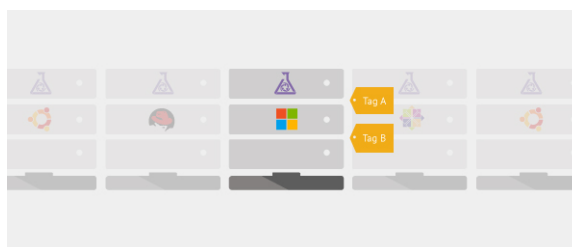


Figure 5: Razor provisions the node

The node is now installed. If you choose, you can hand off management of the node to Puppet Enterprise.

Razor system requirements

The Razor server and client are supported on these operating systems.

Component	Supported OS
Server	<ul style="list-style-type: none"> • RHEL 6 • CentOS 6 and 7
Client	<ul style="list-style-type: none"> • Windows 2008 R2 Server • Windows 2012 R2 Server • Windows 2016 Server • ESXi 5.5 • RHEL 6 • CentOS 6 and 7 • Debian Wheezy

Pre-requisites for machines provisioned with Razor

To successfully install an operating system on a machine using Razor, the machine must meet these specifications.

- Have at least 512MB RAM.
- Be supported by the operating system you're installing.
- Be able to successfully boot into the microkernel. The microkernel is based on [CentOS 7](#), 64-bit only, and supports the [x86-64 Intel architecture](#).
- Be able to successfully boot the [iPXE](#) firmware.

Setting up a Razor environment

Razor relies on a PXE environment to boot the Razor microkernel. You must set up your PXE environment before you can successfully provision with Razor.

There are two ways to enable PXE boot:

- (Recommended) Integrate Razor with the DHCP solution of the system being provisioned.
- Rely on UEFI to directly load the .ipxe file that's required to boot the Razor microkernel.

Because it's difficult to guarantee that all of your hardware is UEFI-enabled, configuring DHCP is the preferred method for setting up a Razor environment.

Set up a Razor environment

Set up a PXE environment using the DHCP and TFTP service of your choice.

Before you begin

You must have Puppet Enterprise installed.

This workflow describes a sample setup using dnsmasq. Dnsmasq is not intended for production environments; however, you can use a similar workflow to set up DHCP and TFTP with more robust solutions.

Important: Set up and test PXE in a completely isolated test environment. Running a second DHCP server on your company's network could bring down the network or replace a server with a fresh installation. See [Protecting existing nodes](#) for strategies on avoiding data loss.

Install and configure dnsmasq DHCP-TFTP service

Install dnsmasq to manage communication between nodes and the Razor server. When a node boots, dnsmasq forwards the booted node to the Razor service.

1. Use YUM to install dnsmasq: `yum install dnsmasq`

2. If it doesn't already exist, create the directory `/var/lib/tftpboot`.
3. Change the permissions for the TFTP boot directory: `chmod 655 /var/lib/tftpboot`.

Temporarily disable SELinux

You must temporarily disable SELinux in order to enable PXE boot. Alternatively, you could craft an enforcement rule for SELinux that enables PXE boot but doesn't completely disable SELinux.

1. In the file `/etc/sysconfig/selinux`, set `SELINUX=disabled`.
2. Restart the computer.

Edit `dnsmasq.conf` to enable DHCP

You must specify a DHCP range to enable communication with your DHCP server.

Tip: For more complex setups, it might be helpful to indicate a range name, for example: `dhcp-range=range1,10.0.1.50,10.0.1.120,24h` `dhcp-range=range2,10.0.1.121,10.0.1.222,48h`.

Edit `/etc/dnsmasq.conf` to specify a DHCP range.

For example, for an IP range from 10.0.1.50 - 10.0.1.120 with a 24-hour lease, your file resembles:

```
# Uncomment this to enable the integrated DHCP server, you need
# to supply the range of addresses available for lease and optionally
# a lease time. If you have more than one network, you must
# repeat this for each network on which you want to supply DHCP
# service.
dhcp-range=10.0.1.50,10.0.1.120,24h
```

Edit the `dnsmasq` configuration file to enable PXE boot

Use `dnsmasq` to enable PXE booting on nodes.

1. At the bottom of the `/etc/dnsmasq.conf` file, add: `conf-dir=/etc/dnsmasq.d`
2. Write and exit the file.
3. Create the file `/etc/dnsmasq.d/razor` and add configuration information.

For example, for `dnsmasq` 2.45:

```
# iPXE sets option 175, mark it for network IPXEBOOT
dhcp-match=IPXEBOOT,175
dhcp-boot=net:IPXEBOOT,bootstrap.ipxe
dhcp-boot=undionly-20140116.kpxe
# TFTP setup
enable-tftp
tftp-root=/var/lib/tftpboot
```

4. Enable `dnsmasq` on boot: `chkconfig dnsmasq on`
5. Start the `dnsmasq` service: `service dnsmasq start`

Installing Razor

After you set up a Razor environment, you're ready to install Razor.

Install Razor

At a minimum, to install Razor, you must install the Razor server. You can also install the client to make interacting with the server easier and to enable authentication security to control permissions.

Tip: Use variables to avoid repeatedly replacing placeholder text. For installation, we recommend declaring a server name and the port to use for Razor with these commands:

```
export RAZOR_HOSTNAME=<server name>
export HTTP_PORT=8150
export HTTPS_PORT=8151
```

The installation tasks on this page use `$<RAZOR_HOSTNAME>`, `$<HTTP_PORT>`, and `$<HTTPS_PORT>` to represent these variables.

Install the Razor server

Installing Razor involves classifying a node with the `pe_razor` module.

When Puppet Enterprise applies this classification, the software downloads automatically and installs a Razor server and a PostgreSQL database. The download can take several minutes.

Because the Razor software is stored online, you need an internet connection to install it.

Note: You cannot install Razor with the `pe_razor` module if your master node has FIPS mode enabled.

Important: Don't install the Razor server on your master.

1. In the console, create a classification node group for the `server`.
2. Click the `server` node group, click **Configuration**, then in the **Add new class** field, enter `pe_razor` and click **Add class**.
3. Commit changes.
4. On the `server`, run `: puppet agent -t`

Related information

[Create classification node groups](#) on page 323

Create classification node groups to assign classification data to nodes.

[Running Puppet on nodes](#) on page 320

Puppet automatically attempts to run on each of your nodes every 30 minutes. To trigger a Puppet run outside of the default 30-minute interval, you can manually run Puppet.

Install the Razor server while you're offline

Install Razor using your own remote directory containing a Puppet Enterprise tarball and the Razor microkernel.

Before you begin

Copy the [Razor microkernel](#) and the [Puppet Enterprise tarball](#) appropriate for your installation to your own FTP site or to the Razor server.

If you save the files locally, when you specify directory paths, use three forward slashes (`file:///`). If you save the files on the Razor server, use two forward slashes (`file://`).

The example parameter values in this task use this directory structure:

```
[root@razor ~]# tree /tmp/razor_files/
/tmp/razor_files/
### 2017.2.3/
#   ### puppet-enterprise-2017.2.3-el-7-x86_64.tar.gz
### microkernel-008.tar
```

1. In the console, create a classification node group for the `server`.

- Click the `server` node group, click **Configuration**, then in the **Add new class** field, enter `pe_razor` and click **Add class**.
- In the `pe_razor` class, specify the URL for your tarball, and then click **Add parameter**.

Parameter	Value
pe_tarball_base_url Note: This parameter can be used only for installation, not upgrades	Full path to a directory that contains <code><PE_VERSION>/<PE_INSTALLER>.tar.gz</code> , for example <code>file:///tmp/razor_files</code> . The complete URL is automatically constructed by appending the PE build and file name to the base directory.

- In the `pe_razor` class, specify the URL for the microkernel, and then click **Add parameter**.

Parameter	Value
microkernel_url	Full path to the microkernel- <code><VERSION>.tar</code> , for example <code>file:///tmp/razor_files/microkernel-008.tar</code> .

- Commit changes.
- On the server, run `puppet agent -t`
- (Optional) Install the Razor client.
 - From a web-enabled machine, fetch these gems:


```
/opt/puppetlabs/puppet/bin/gem fetch colored --version 1.2
/opt/puppetlabs/puppet/bin/gem fetch command_line_reporter --version 3.3.6
/opt/puppetlabs/puppet/bin/gem fetch mime-types --version 1.25.1
/opt/puppetlabs/puppet/bin/gem fetch multi_json --version 1.12.1
/opt/puppetlabs/puppet/bin/gem fetch razor-client --version 1.9.4
/opt/puppetlabs/puppet/bin/gem fetch faraday --version 0.15.4
```
 - Copy the downloaded gems to a directory on the Razor server.
 - Install the gems on the Razor server: `/opt/puppetlabs/puppet/bin/gem install -f --local *.gem`

Related information

[Create classification node groups](#) on page 323

Create classification node groups to assign classification data to nodes.

[Running Puppet on nodes](#) on page 320

Puppet automatically attempts to run on each of your nodes every 30 minutes. To trigger a Puppet run outside of the default 30-minute interval, you can manually run Puppet.

Change the default Razor port

The default ports for Razor are port 8150 for HTTP communication between the server and nodes, and port 8151 for HTTPS, used for accessing the public API. You can optionally change the default ports if they're occupied by another service, out of range, or blocked by a firewall.

- In the console, select the **Classification**, then click the node group that contains the `pe_razor` module.
- On the **Configuration** tab, under `pe_razor` in the **Parameters** box, select whether you want to change `server_http_port` or `server_https_port`.
- In the **Value** field, enter the port number to use, click **Add parameter**, and then commit changes.

Parameters for the `pe_razor` class

Parameters for the `pe_razor` class enable customization of your Razor installation. You can review configuration settings that are currently being used by Razor with the `razor config` command.

The `pe_razor` class has these parameters:

Parameter	Default	Description
<code>api_config_blacklist</code>	<code>['facts.blacklist', 'database_url']</code>	Properties that Razor hides from query results. You can add additional properties to protect sensitive data.
<code>auth_config</code>	<code>/etc/puppetlabs/razor-server/shiro.ini</code>	Path to the authentication configuration file.
<code>auth_enabled</code>	<code>false</code>	<code>true</code> to enable authentication for requests to <code>/api</code> endpoints, or <code>false</code> .
<code>broker_path</code>	<code>/etc/puppetlabs/razor-server/brokers:brokers</code>	Colon-separated list of directories containing broker types.
<code>checkin_interval</code>	<code>15</code>	Interval, in seconds, at which the microkernel checks in with the Razor server.
<code>database_url</code>	<code>jdbc:postgresql:razor?user=razor&sslmode=require&sslcert=/etc/puppetlabs/razor-server/ssl/client.cert.pem&sslkey=/etc/puppetlabs/razor-server/ssl/client.key.pk8</code>	URL for the Razor server.
<code>dbpassword</code>	<code>razor</code>	Password to the Razor database.
<code>facts_blacklist</code>	<code>['domain', 'filesystems', 'fqdn', 'hostname', 'id', '/kernel.*/', 'memoryfree', 'memorysize', 'memorytotal', '/operatingsystem.*/', 'osfamily', 'path', 'ps', 'rubysitedir', 'rubyversion', 'selinux', 'sshdsa_key', '/sshfp_[dr]sa/', 'sshrsa_key', '/swap.*/', 'timezone', '/uptime.*/']</code>	Facts that Razor ignores. Each entry can be a string or a regexp enclosed in <code>/ . . /</code> where any fact that matches the regexp is dropped.
<code>facts_match_on</code>	<code>[]</code>	Array of values used to match nodes from within the microkernel to nodes in the Razor database. By default, this parameter excludes <code>/^macaddress.&#42;/</code> (regex), <code>serialnumber</code> , and <code>uuid</code> , which are already used.
<code>hook_execution_path</code>	<code>/opt/puppetlabs/puppet/bin</code>	Colon-separated list of paths that Razor searches in order when running hooks, prior to using the default execution path.
<code>hook_path</code>	<code>/etc/puppetlabs/razor-server/hooks:hooks</code>	Colon-separated list of directories containing hook types.

Parameter	Default	Description
match_nodes_on	['mac']	Array of values used to match nodes when a node PXE boots. Values can include mac, serial, asset, or uuid.
microkernel_debug_level	quiet	Sets the logging level for the microkernel. Valid values are quiet or debug.
microkernel_extension_zip	/etc/puppetlabs/razor-server/mk-extension.zip	Zip file that specifies custom facts or other code that is unpacked by the microkernel prior to checkin.
microkernel_kernel_args	"	Additional command-line arguments that are supplied to the microkernel during boot.
microkernel_url	https://pm.puppetlabs.com/puppet-enterprise-razor-microkernel-\$	Location of the Razor microkernel used to install Razor offline.
pe_tarball_base_url	https://pm.puppetlabs.com/puppet-enterprise	Location of the Puppet Enterprise tarball used to install Razor offline.
protect_new_nodes	false	true to make new machines ineligible for provisioning, or false.
repo_store_root	/opt/puppetlabs/server/data/razor-server/repo	Directory where repository contents are downloaded and served.
secure_api	true	true to require HTTPS/SSL communication with /api endpoints.
server_http_port	8150	Port that nodes use to communicate with the server over HTTP. Only URLs starting with /svc need to be available on this port.
server_https_port	8151	Port that the client uses to communicate with the server's public API over HTTPS. Only URLs starting with /api need to be available on this port.
task_path	/etc/puppetlabs/razor-server/tasks:tasks	Colon-separated list of directories containing tasks.

Verify the Razor server

Use a test command to verify that the Razor server is correctly installed. The output JSON file `test.out` contains a list of available Razor commands.

Test the Razor configuration:

```
wget https://$<RAZOR_HOSTNAME>:$<HTTPS_PORT>/api -O test.out
```


Install the Razor client

The Razor client is installed as a Ruby gem, `razor-client`. The process for installing the client differs by platform.

Install the Razor client on *nix systems

Make interacting with the Razor server from *nix systems easier by installing the Razor client.

1. Install the client: `gem install razor-client`

Tip: If you're installing the Razor client on your master, you must specify the location of the gem: `/opt/puppetlabs/puppet/bin/gem install razor-client`

2. Point the Razor client to the server: `razor -u https://$<RAZOR_HOSTNAME>:$<HTTPS_PORT>/api`
An error displays if the client isn't installed or can't connect to the server.
3. (Optional) If you receive a warning message about JSON, you can optionally disable it: `gem install json_pure`

Install the Razor client on Windows systems

Make interacting with the Razor server from Windows systems easier by installing the Razor client.

1. From a Puppet command prompt (**Start > Start Command Prompt with Puppet > Run as administrator**), install the client: `gem install razor-client`
2. Set an environment variable for the Razor server URL: `setx RAZOR_API https://$<RAZOR_HOSTNAME>:8151/api`.

Tip: Alternatively, you can set the variable through **User Accounts** in the **Control Panel**.

Enable authentication security

Enable authentication security to control what tasks users can perform. For example, you might limit certain users to read permissions to avoid accidental overwrite of nodes.

Two methods are required to secure your Razor server:

- Authentication security using Shiro – disabled by default
- Protocol security using HTTPS and TLS/SSL – enabled by default if you're using Puppet Enterprise and you install Razor on a managed node

1. In the console, in the `pe_razor` class, specify Shiro authentication parameters.

Parameter	Value
<code>auth_enabled</code>	<code>true</code>
<code>auth_configured</code>	Location of the Shiro file, <code>/etc/puppetlabs/razor-server/shiro.ini</code>

2. Specify users, passwords, and roles in your `shiro.ini` file, located at `/etc/puppetlabs/razor-server/shiro.ini`.

For example, this INI file specifies two users: `razor` is an admin who can perform all functions, and `other` is a user who can perform only read operations, such as viewing collections.

```
[main]
sha256Matcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher
iniRealm.credentialsMatcher = $sha256Matcher

[users]
# define users known to shiro, using the format:
# <username> = <sha256 password hash>, <role>[, <role>...]
razor = 9b4f1d0e11dcc029c3493d945e44ee077b68978466c0aab6dlce453aac5f0384,
admin
```

```

other = d74ff0ee8da3b9806b18c877dbf29bbde50b5bd8e4dad7a3a725000feb82e8f1,
user

[roles]
# define roles and their associated permissions
admin = *
user = query:*

```

3. Restart the Razor service: `service pe-razor-server restart`

Permissions for Razor

These are the available permissions for Razor.

Note: Commands have varying access control patterns. Use `razor <COMMAND_NAME> --help` to view required permissions for each command.

Task	Permission
Query all collections	<code>query:*</code>
Query the node collection	<code>query:nodes</code>
Query a specific node	<code>query:nodes:<NODE_NAME></code>
Run all commands	<code>commands:*</code>
Create policies	<code>commands:create-policy:*</code>
Create policies starting with a specific name	<code>commands:create-policy:<NAME>*</code>

Verify Razor versions

Verify your installation by checking the version of the Razor server and client. This information can also be useful for troubleshooting.

On the client or server, verify versions: `razor --version` or `razor -v`

Using the Razor client

There are three ways to communicate with the Razor server.

- API calls in JSON sent directly to the server
- JSON arguments sent from the Razor client to the server
- Razor client commands

Client commands are the easiest way to interact with the server.

Client commands begin with `razor`, followed by the name of the action, like `razor create-repo` or `razor move-policy`. One or more arguments follow the command.

You can access help for each command by entering the command with the `--help` flag, for example `razor add-policy-tag --help`.

Using positional arguments with Razor client commands

Most Razor client commands allow positional arguments, which means that you don't have to explicitly enter the name of the argument, like `--name`. Instead, you can provide the values for each argument in a specific order.

For example, the `delete-policy` command includes only one argument, `--name`. To delete a policy named *sprocket*, you can enter the command with the argument name and value, or with a positional argument:

- command with argument name and value — `razor delete-policy --name sprocket`
- command with positional argument — `razor delete-policy sprocket`

If a command includes multiple options, you can supply from zero to all available positional arguments. For example, the `add_policy_tag` command has three positional arguments: `name`, `tag`, and `rule`. You can provide no positional arguments, `name` only, `name` and `tag` only, or all three arguments.

Because not all arguments are available as positional arguments, you might need to use a combination of positional arguments and name-value pairs. For example, the `create-hook` command has two positional arguments, `name` and `hook-type`, which you might use along with a `--configuration` value, like:

```
razor create-hook name_of_hook hook_type --configuration someconfig=value
```

You can switch between positional and non-positional arguments, but you must maintain the expected order for positional arguments. For example:

```
razor command positional-arg1 --non-positional value positional-arg2 --non-positional2
```

Positional arguments for client commands

These are the Razor client commands, with available positional arguments listed in accepted order.

Command	Positional arguments
<code>add-policy-tag</code>	<code>name</code> , <code>tag</code> , <code>rule</code>
<code>create-broker</code>	<code>name</code> , <code>broker-type</code>
<code>create-hook</code>	<code>name</code> , <code>hook-type</code>
<code>create-policy</code>	<code>name</code>
<code>create-repo</code>	<code>name</code>
<code>create-tag</code>	<code>name</code> , <code>rule</code>
<code>create-task</code>	<code>name</code>
<code>delete-broker</code>	<code>name</code>
<code>delete-hook</code>	<code>name</code>
<code>delete-node</code>	<code>name</code>
<code>delete-policy</code>	<code>name</code>
<code>delete-repo</code>	<code>name</code>
<code>delete-tag</code>	<code>name</code>
<code>disable-policy</code>	<code>name</code>
<code>enable-policy</code>	<code>name</code>
<code>modify-node-metadata</code>	<code>name</code>
<code>modify-policy-max-count</code>	<code>name</code> , <code>max_count</code>
<code>move-policy</code>	<code>name</code>
<code>reboot-node</code>	<code>name</code>
<code>register-node</code>	(none)
<code>reinstall-node</code>	<code>name</code>
<code>remove-node-metadata</code>	<code>node</code> , <code>key</code>
<code>remove-policy-tag</code>	<code>name</code> , <code>tag</code>
<code>run-hook</code>	<code>name</code>

Command	Positional arguments
set-node-desired-power-state	name, to
set-node-hw-info	node
set-node-ipmi-credentials	name
update-broker-configuration	broker, key, value
update-hook-configuration	hook, key, value
update-node-metadata	node, key, value
update-policy-repo	policy, repo
update-policy-task	policy, task
update-policy-broker	policy, broker
update-policy-node-metadata	policy, key, value
update-repo-task	repo, task
update-tag-rule	name, rule

Razor client commands

Use client commands to interact with Razor and provision bare metal nodes.

Configuration commands

Razor configuration is pulled from a configuration file controlled by Puppet Enterprise. You can change configuration values in the console with class parameters of the `pe_razor` module.

Note: In order for configuration changes to take effect, you must restart the Razor service: `service pe-razor-server restart`.

config

The `config` command displays details about your Razor configuration.

Note: Properties specified in the `api_config_blacklist` aren't returned by the `config` command.

Sample command

To view details about your Razor configuration:

```
razor config
```

Node commands

Nodes are created either through the node boot endpoint, when the node initiates its first web request to the Razor server, or through the `register-node` command.

Both methods of creating a node create a stub in the Razor database. When the node boots into the microkernel, it gathers facts and reports them to the node checkin endpoint, linking new facts to the stub. At that point, the node is fully registered in the Razor system and goes through the binding process.

register-node

The `register-node` command enables you to identify a node before it's discovered. This can be useful to apply metadata before the node boots up, or to indicate to Razor not to provision – or reprovision – a node that's already installed.

Command attributes

Attributes	Required	Positional arguments	Description
hw_info	#		<p>Hardware information about the node.</p> <p>You must include enough information that the node can be identified by hardware information sent by the firmware when the node boots. This usually includes the MAC addresses of all network interfaces.</p> <p>This attribute can include some or all of these entries:</p> <ul style="list-style-type: none"> • netN — The MAC address of each network interface. The order of addresses isn't significant. • serial — DMI serial number of the node. • asset — DMI asset number of the node. • uuid — DMI universally unique identifier of the node.
installed	#		<p>true to indicate that the node won't be provisioned — or reprovisioned — by Razor, or false.</p>

Sample command

To register an installed machine before it's booted, and prevent reprovisioning:

```
razor register-node --hw-info net0=78:31:c1:be:c8:00 \
  --hw-info net1=72:00:01:f2:13:f0 \
  --hw-info net2=72:00:01:f2:13:f1 \
  --hw-info serial=xxxxxxxxxxx \
  --hw-info asset=Asset-1234567890 \
  --hw-info uuid="Not Settable" \
  --installed
```

set-node-hw-info

The `set-node-hw-info` command sets or updates the hardware info for a specified node. This is useful when a node's hardware changes, such as when you replace a network card.

Command attributes

Attributes	Required	Positional arguments	Description
node	#	1	Name of the node to update.

Attributes	Required	Positional arguments	Description
hw_info	#		<p>Hardware information about the node.</p> <p>This attribute can include some or all of these entries:</p> <ul style="list-style-type: none"> • <code>netN</code> — The MAC address of each network interface. The order of addresses isn't significant. • <code>serial</code> — DMI serial number of the node. • <code>asset</code> — DMI asset number of the node. • <code>uuid</code> — DMI universally unique identifier of the node.

Sample command

To update node172 with new hardware information:

```
razor set-node-hw-info --node node172 \
  --hw-info net0=78:31:c1:be:c8:00 \
  --hw-info net1=72:00:01:f2:13:f0 \
  --hw-info net2=72:00:01:f2:13:f1 \
  --hw-info serial=xxxxxxxxxxxx \
  --hw-info asset=Asset-1234567890 \
  --hw-info uuid="Not Settable"
```

delete-node

The `delete-node` command deletes a specified node from the Razor database. The `delete-node` command is similar to `reinstall-node`, except that `delete-node` doesn't retain log information or the node number.

Note: If the deleted node boots again at some point, Razor automatically recreates the node as if it were the first time it contacted the Razor server.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the node to delete.

Sample command

To delete node17:

```
razor delete-node --name node17
```

reinstall-node

The `reinstall-node` command clears a node's installed flag and – by default – removes its association with policies. You can use the `same-policy` attribute to retain the assigned policy.

After restart, the node boots into the microkernel, goes through discovery and tag matching, and can bind to another policy for reinstallation. This command doesn't change the node's metadata or facts.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the node to reinstall.
same-policy			true to retain the same policy for reinstallation, or false. If omitted, the node can bind to a different policy when reinstalled.

Sample commands

To reinstall node17:

```
razor reinstall-node --name node 17
```

To reinstall node17 and retain its assigned policy:

```
razor reinstall-node --name node17 --same-policy
```

IPMI commands

IPMI commands are node commands based on the Intelligent Platform Management Interface.

Note: You must install `ipmitool` on the Razor server before using IPMI commands. To install the tool, run `yum install ipmitool -y`.

set-node-ipmi-credentials

The `set-node-ipmi-credentials` sets or updates the host name, user name, or password for connecting with a BMC/LOM/IPMI LAN or LANplus service.

The command works only with remote IPMI targets, not local targets.

After IPMI credentials have been set up for a node, you can use the `reboot-node` and `set-node-desired-power-state` commands.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the node to update.
ipmi_hostname			IPMI host name or IP address of the BMC of the host.
ipmi_username			IPMA LANplus username, if any, for this BMC.
ipmi_password			IPMI LANplus password, if any, for this BMC.

Sample command

To set IPMI credentials for node17:

```
razor set-node-ipmi-credentials --name node17 \
```

```
--ipmi-hostname bmc17.example.com \
--ipmi-username null \
--ipmi-password sekretskwirrl
```

reboot-node

The `reboot-node` command triggers a hard power cycle on a specified node using IPMI credentials.

Note: You must specify IPMI credentials before using the `reboot-node` command.

If an execution slot isn't available on the target node, the `reboot-node` command is queued and runs as soon as a slot is available. There are no limits on how many commands you can queue up, how frequently a node can be rebooted, or how long a command can stay in the queue. If you restart your Razor server before queued commands are executed, they remain in the queue and run after the server restarts.

The `reboot-node` command is not integrated with IPMI power state monitoring, so you can't see power transitions in the record or when polling the node object.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the node to reboot.

Sample command

To queue a node reboot:

```
razor reboot-node --name node1
```

set-node-desired-power-state

The `set-node-desired-power-state` command specifies whether you want a node to remain powered off or on. By default, Razor checks node power state every few minutes in the background. If it detects a node in a non-desired state, Razor issues an IPMI command directing the node to its desired state.

Note: You must specify IPMI credentials before using the `set-node-desired-power-state` command.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the node to power off or on.
to		2	Desired power state: on, off, or null, meaning that Razor won't enforce a power state.

Sample command

To power on node1234 and keep it on:

```
razor set-node-desired-power-state --name node1234 \
--to on
```


Node metadata commands

Node metadata commands enable you to update, modify, or remove metadata from nodes.

update-node-metadata

The `update-node-metadata` command sets or updates a single metadata key.

Command attributes

Attributes	Required	Positional arguments	Description
node	#	1	Name of the node for which to modify metadata.
key	#	2	Name of the key to modify.
value	#	3	New value for the key.
no_replace			true to cancel the update operation if the specified key is already present, or false.

Sample command

To update the `my_key` key for a node:

```
razor update-node-metadata --node node1 \
  --key my_key --value twelve
```

modify-node-metadata

The `modify-node-metadata` command sets, updates, or clears metadata key-value pairs.

Command attributes

Attributes	Required	Positional arguments	Description
node	#	1	Name of the node for which to modify metadata.
update	*		Key and value pair to update.
remove	*		Key and value pair to remove.
clear	*		true to clear all metadata, or false.
no_replace			true to cancel the update operation if the specified key is already present, or false.
force			true to bypass errors in a batch operation with <code>no_replace</code> , or false. Existing keys aren't modified.

* You must specify one of three attributes: `update`, `remove`, or `clear`.

Sample commands

To add values for key1 and key2 to a node, but not if they are already set, and to remove key3 and key4:

```
razor modify-node-metadata --node node1 --update key1=value1 \
    --update key2='[ "val1", "val2", "val3" ]' --remove key3 --remove key4
    --noreplace
```

To remove all node metadata:

```
razor modify-node-metadata --node node1 --clear
```

remove-node-metadata

The `remove-node-metadata` command removes either a single metadata entry or all metadata entries on a node.

Command attributes

Attributes	Required	Positional arguments	Description
node	#	1	Name of the node for which to modify metadata.
key	*	2	Name of the key to remove.
all	*		true to remove all metadata about the node.

* You must specify one of two attributes: `key` or `all`.

Sample commands

To remove a single key from a node:

```
razor remove-node-metadata --node node1 --key my_key
```

To remove all keys from a node:

```
razor remove-node-metadata --node node1 --all
```

Repository commands

Repository commands enable you to create and delete specified repositories from the Razor database and specify the task that installs the contents of a repository.

create-repo

The `create-repo` command creates a new repository. The repository can contain the content to install a node, or it can point to an existing online repository.

You can create three types of repositories:

- Those that reference content available on another server, for example, on a mirror you maintain (`url`).
- Those where unpacks ISOs for you and serves their contents (`iso_url`).
- Those where creates a stub directory that you can manually fill with content (`no_content`).

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name for the new repository.

Attributes	Required	Positional arguments	Description
<code>url</code>	*		URL of a remote repository.
<code>iso_url</code>	*		<p>URL of an ISO image to download and unpack in a new, local repository.</p> <p>You can use an HTTP or HTTPS URL, or a file URL. If you're using a file URL, manually place the ISO image on the server before you call the command.</p> <p>If you supply the <code>iso_url</code> attribute to create a repository, you can delete it from the server later using the <code>delete-repo</code> command.</p>
<code>no_content</code>	*		<p>Creates a stub directory in the repo store where you can manually extract an image.</p> <p>After this command finishes, you can log into your server and fill the repository directory with the content, for example by loopback-mounting the install media and copying it into the directory. The repository directory is specified by the <code>repo_store_root</code> class parameter of the <code>pe_razor</code> module. By default, the directory is <code>/opt/puppetlabs/server/data/razor-server/repo</code>.</p> <p>This attribute is usually necessary for Windows install media, because the library that Razor uses to unpack ISO images doesn't support Windows ISO images.</p>

Attributes	Required	Positional arguments	Description
task	#		<p>Name of the default task that installs nodes from this repository.</p> <p>We recommend that the task match the operating system specified by the url or iso_url.</p> <p>You can override this parameter at the policy level.</p> <p>Tip: You can use razor tasks to see which tasks are available on your server.</p>

* You must specify one of the attributes url, iso_url, or no_content.

Sample commands

To create a repository from an ISO image, which is downloaded and unpacked by the server in the background:

```
razor create-repo --name fedora21 \
  --iso-url http://example.com/Fedora-21-x86_64-DVD.iso \
  --task fedora
```

To unpack an ISO image from a file on a server without uploading the file from the client:

```
razor create-repo --name fedora21 \
  --iso-url file:///tmp/Fedora-21-x86_64-DVD.iso \
  --task fedora
```

To point to a resource without downloading content to the server:

```
razor create-repo --name fedora21 --url \
  http://mirrors.n-ix.net/fedora/linux/releases/21/Server/x86_64/os/ \
  --task fedora
```

To create a stub directory that you can manually fill with content:

```
razor create-repo --name fedora21 --no-content --task Fedora
```

update-repo-task

The update-repo-task command specifies the task that installs the contents of the repository.

If a node is currently provisioning against the repository when you run this command, provisioning might fail.

Command attributes

Attributes	Required	Positional arguments	Description
repo	#	1	Name of an existing repository to update.
task	#	2	Name of the task to be used by the repository.

Sample command

To update the `my_repo` repository to the task `other_task`:

```
razor update-repo-task --repo my_repo --task other_task
```

delete-repo

The `delete-repo` command deletes a specified repository from the Razor database.

Before deleting a repository, remove any references to it in existing policies. This command fails if the repository is in use by an existing policy.

If you supplied the `iso_url` property when you created the repository, the folder is also deleted from the server. If you didn't supply the `iso_url` property, content remains in the repository directory.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the repository to delete.

Sample command

To delete an obsolete repository:

```
razor delete-repo --name my_obsolete_repo
```

Task commands

Use task commands to create tasks in the Razor database.

Important: tasks differ from `tasks`, which let you run arbitrary scripts and commands using `.` See the orchestrator documentation for information about `tasks`.

create-task

The `create-task` command creates a task in the Razor database. This command is an alternative to manually placing task files in the `task_path`. If you anticipate needing to make changes to tasks, we recommend the disk-backed task approach.

Razor has a set of tasks for installing on supported operating systems. See the [razor-server Github page](#) for more information.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name for the new task.
os	#		Name of the operating system installed by the task.
templates	#		Named templates used for task stages.

Attributes	Required	Positional arguments	Description
boot_seq			<p>List of key-value pairs that describe templates in the order they're installed.</p> <p>Use the optional default hash key to specify the default template to use when no other template applies.</p>

Sample command

To define the RedHat task included with Razor:

```
razor create-task --name redhat-new --os "Red Hat Enterprise Linux" \
  --description "A basic installer for RHEL6" \
  --boot-seq 1=boot_install --boot_seq default=boot_local \
  --templates "boot_install=#!ipxe\necho Razor <%= task.label %> task
boot_call\necho Installation node: <%= node_url %>\necho Installation
repo: <%= repo_url %>\n\nsleep 3\nkernel <%= repo_url("/isolinux/vmlinuz")
%> <%= render_template("kernel_args").strip %> || goto error\ninitrd <%=
repo_url("/isolinux/initrd.img") %> || goto error\nboot\n:error\nprompt --
key s --timeout 60 ERROR, hit 's' for the iPXE shell; reboot in 60 seconds
&& shell || reboot\n" \
  --templates kernel_args="ks=<%= file_url("kickstart") %> network
ksdevice=bootif BOOTIF=01-${netX/mac}" \
  --templates kickstart="#!/bin/bash\n# Kickstart for RHEL/CentOS 6\n#
see: http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/
html/Installation_Guide/sl-kickstart2-options.html\n\ninstall\nurl=
url=<%= repo_url %>\nncmdline\nlang en_US.UTF-8\nkeyboard us\nrootpw
<%= node.root_password %>\nnetwork --hostname <%= node.hostname %>
\nfirewall --enabled --ssh\nauthconfig --enablesshadow --passalgo=sha512
--enablefingerprint\ntimezone --utc America/Los_Angeles\n# Avoid
having 'rhgb quiet' on the boot line\nbootloader --location=mbr --
append="crashkernel=auto"\n# The following is the partition information
you requested\n# Note that any partitions you deleted are not expressed
\n# here so unless you clear all partitions first, this is\n# not
guaranteed to work\nzerombr\nclearpart --all --initlabel\nautopart\n#
reboot automatically\nreboot\n\n# following is MINIMAL https://partner-
bugzilla.redhat.com/show_bug.cgi?id=593309\n%packages --nobase\n@core\n
\n%end\n\n%post --log=/var/log/razor.log\nnecho Kickstart post\ncurl -s
-o /root/razor_postinstall.sh <%= file_url("post_install") %>\n\n# Run
razor_postinstall.sh on next boot via rc.local\nif [ ! -f /etc/rc.d/
rc.local ]; then\n # On systems using systemd /etc/rc.d/rc.local does not
exist at all\n # though systemd is set up to run the file if it exists
\n touch /etc/rc.d/rc.local\n chmod a+x /etc/rc.d/rc.local\nfi\nnecho
bash /root/razor_postinstall.sh >> /etc/rc.d/rc.local\nchmod +x /root/
razor_postinstall.sh\n\ncurl -s <%= stage_done_url("kickstart") %>\n%end
\n#####\n" \
  --templates post_install="#!/bin/bash\n\nexec >> /var/log/razor.log
2>&1\n\nnecho "Starting post_install"\n\n# Wait for network to come up
when using NetworkManager.\nif service NetworkManager status >/dev/
null 2>&1 && type -P nm-online; then\n nm-online -q --timeout=10 ||
nm-online -q -x --timeout=30\n [ "$?" -eq 0 ] || exit 1\nfi\n\n<%=
render_template("set_hostname") %>\n\n<%= render_template("store_ip") %>
\n\n<%= render_template("os_complete") %>\n\n# We are done\ncurl -s <%=
stage_done_url("finished") %>\n"
```

Tag commands

Tag commands enable you to create new tags, set the rules used to apply the tag to nodes, change the rule of a specified tag, or delete a specified tag.

create-tag

The `create-tag` command creates a new tag and sets the rules used to apply the tag to nodes.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name for the new tag.
rule	#	2	Case-sensitive match expression that applies the tag to a node if the expression evaluates as true.

Sample command

To create a tag that's applied to nodes with two processors:

```
razor create-tag --name small --rule '["=", ["fact", "processorcount"],
"2"]'
```

update-tag-rule

The `update-tag-rule` changes the rule of a specified tag.

After updating a tag, Razor reevaluates the tag against all nodes. With the `--force` flag, the tag is reevaluated even if it's used by policies.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of an existing tag to update.
rule	#	2	Case-sensitive match expression that applies the tag to a node if the expression evaluates as true.
force			Reevaluates tags used by an existing policy.

Sample command

To update a tag rule and reevaluate nodes with the tag, even if the tag is used by a policy:

```
razor update-tag-rule --name small --force \
  --rule '["<=", ["fact", "processorcount"], "2"]'
```

delete-tag

The `delete-tag` command deletes a specified tag.

With the `--force` flag, the tag is deleted and removed from any policies that use it.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of an existing tag to delete.
force			Deletes tags used by an existing policy.

Sample commands

To delete an unused tag:

```
razor delete-tag --name my_obsolete_tag
```

To delete a tag whether or not it is used:

```
razor delete-tag --name my_obsolete_tag --force
```

Related information

[Tags](#) on page 663

A tag consists of a unique name and a rule. Tags match a node if evaluating the node against the tag's facts results in `true`. Tag matching is case sensitive.

[Tags API](#) on page 682

These commands enable you to create a tag, delete a tag, or change the rule for a tag.

Policy commands

Policies govern how nodes are provisioned.

Razor maintains an ordered table of policies. When a node boots, Razor traverses the table to find the first eligible policy for that node. A policy might be ineligible for binding to a node if the node doesn't contain all of the tags on the policy, if the policy is disabled, or if the policy has reached its maximum for the number of allowed nodes.

When you list the `policies` collection, the list is in the order that Razor checks policies against nodes.

create-policy

The `create-policy` command creates a new policy that determines how nodes are provisioned. Razor maintains an ordered table of policies, and applies the first policy that matches a node.

Tip: You can create policies in a JSON file to make saving and modifying them easier. To apply a policy with a JSON file, use `razor create-policy --json <FILENAME>.json`.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name for the new policy.
hostname	#		Pattern for host names of nodes bound to this policy. The <code>\${id}</code> references each node's internal ID number.

Attributes	Required	Positional arguments	Description
root_password	#		<p>Root password for the new system.</p> <p>Valid values depend on how the task renders passwords.</p>
enabled			<p>true if the policy is enabled upon creation, or false.</p> <p>If omitted, the policy is enabled.</p>
max_count			<p>Number indicating how many nodes can bind to the policy.</p> <p>If omitted, the policy can bind to an unlimited number of nodes.</p>
before			<p>Name of the policy that this policy is placed before in the policy list.</p> <p>This attribute can't be used with after.</p>
after			<p>Name of the policy that this policy is placed after in the policy list.</p> <p>This attribute can't be used with before.</p>
tags			<p>Names of tags that a node must match to qualify for the policy.</p> <p>A node must have all tags specified for the policy to apply.</p>
repo	#		<p>Name of the repository that contains the operating system installed by this policy.</p>
broker	#		<p>Name of the broker to use after provisioning to hand off the node to a management system.</p> <p>If you don't want to hand off management, use noop.</p>

Attributes	Required	Positional arguments	Description
task			Name of the task to install nodes that bind to this policy. If omitted, the <code>task</code> property of the policy's repo is used.
node_metadata			Metadata to apply to the node when this policy is bound. Existing metadata is not overwritten. To install Windows on non-English systems, specify the <code>win_language</code> using the culture code for the appropriate Microsoft language pack . For example, <code>--node-metadata win_language=es-ES</code>

Sample command

To create a policy that installs CentOS 6.4:

```
razor create-policy --name centos-for-small \
--repo centos-6.4 --task centos --broker noop \
--enabled --hostname "host${id}.example.com" \
--root-password secret --max-count 20 \
--before "other policy" --tag small --node-metadata key=value
```

move-policy

The `move-policy` command lets you change the order in which Razor considers policies for matching against nodes.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the policy to move.
before	*		Name of the policy that this policy is placed before in the policy list.
after	*		Name of the policy that this policy is placed after in the policy list.

* You must specify one of the two ordering attributes, `before` or `after`.

Sample commands

To move a policy before another policy:

```
razor move-policy --name policy --before succeedingpolicy
```

To move a policy after another policy:

```
razor move-policy --name policy --after precedingpolicy
```

enable-policy and disable-policy

The `disable-policy` command prevents the specified policy from binding to any nodes. This can be useful if you want to temporarily inactivate a policy without deleting it. For example, you can use `disable-policy` to prevent nodes from installing a certain OS for a short period.

The `enable-policy` command reactivates a disabled policy.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the policy to disable or enable.

Sample commands

To disable a policy:

```
razor disable-policy --name my_policy
```

To enable a policy:

```
razor enable-policy --name my_policy
```

modify-policy-max-count

The `modify-policy-max-count` command sets or removes the limit on the maximum number of nodes that can bind to a policy.

Note: To reduce the number of nodes a policy can bind to, you must use the `reinstall-node` command to mark nodes as uninstalled.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the policy to modify.
max_count	*	2	Maximum number of nodes the policy can bind to.
no_max_count	*		Removes any limits on the number of nodes that can bind to the policy.

*You must specify one of the two count attributes, `max_count` or `no_max_count`.

Sample commands

To allow a policy to match an unlimited number of nodes:

```
razor --name example --no-max-count
```

To set a policy to match a maximum of 15 nodes:

```
razor --name example --max-count 15
```

add-policy-tag

The `add-policy-tag` command adds a new or existing tag to a policy. Because binding to policies happens only when unbound nodes check in, adding a policy tag doesn't affect nodes that are already bound to a policy.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the policy to modify.
tag	#	2	Name of the tag to add to the policy.
rule		3	Creates a new tag using a case-sensitive match expression that applies the tag to a node if the expression evaluates as true. If tag creation fails, the policy isn't modified.

Sample commands

To add the existing tag `virtual` to the policy `example`:

```
razor add-policy-tag --name example --tag virtual
```

To add a new tag `virtual` to the policy `example`:

```
razor add-policy-tag --name example --tag virtual \
  --rule '["=", ["fact", "virtual", "false"], "true"]'
```

remove-policy-tag

The `remove-policy-tag` command removes a tag from a policy. Because binding to policies happens only when unbound nodes check in, adding a policy tag doesn't affect nodes that are already bound to a policy.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the policy to modify.
tag	#	2	Name of the tag to remove from the policy.

Sample command

To remove the tag `virtual` from the policy example:

```
razor remove-policy-tag --name example --tag virtual
```

update-policy-repo

The `update-policy-repo` command modifies the repository associated with a policy.

Tip: Before using this command, use `razor policies <POLICYNAME> nodes` to verify that no nodes are currently provisioning against the policy. If a node is provisioning against the policy when you run this command, provisioning might fail.

Command attributes

Attributes	Required	Positional arguments	Description
<code>policy</code>	#	1	Name of the policy to modify.
<code>repo</code>	#	2	Name of the new repository associated with the policy.

Sample command

To update a policy's repository to a repository named `fedora21`:

```
razor update-policy-repo --policy my_policy --repo fedora21
```

update-policy-task

The `update-policy-task` command adds or removes a task from a policy.

Tip: Before using this command, use `razor policies <POLICYNAME> nodes` to verify that no nodes are currently provisioning against the policy. If a node is provisioning against the policy when you run this command, provisioning might fail.

Command attributes

Attributes	Required	Positional arguments	Description
<code>policy</code>	#	1	Name of the policy to modify.
<code>task</code>		2	Name of the task to add to the policy.
<code>no_task</code>			true if the policy uses the task in the repository, or false.

Sample commands

To update a policy's task to a task named `other_task`:

```
razor update-policy-task --policy my_policy --task other_task
```

To use the task specified by the policy's repository:

```
razor update-policy-task --policy my_policy --no-task
```

update-policy-broker

The `update-policy-broker` command modifies the broker associated with a policy.

Tip: Before using this command, use `razor policies <POLICYNAME> nodes` to verify that no nodes are currently provisioning against the policy. If a node is provisioning against the policy when you run this command, provisioning might fail.

Command attributes

Attributes	Required	Positional arguments	Description
<code>policy</code>	#	1	Name of the policy to modify.
<code>broker</code>	#	2	Name of the new broker associated with the policy.

Sample command

To update a policy's broker to a broker named `legacy-puppet`:

```
razor update-policy-broker --policy my_policy --broker legacy-puppet
```

update-policy-node-metadata

The `update-policy-node-metadata` command modifies the node metadata associated with a policy.

Tip: Before using this command, use `razor policies <POLICYNAME> nodes` to verify that no nodes are currently provisioning against the policy. If a node is provisioning against the policy when you run this command, provisioning might fail.

Command attributes

Attributes	Required	Positional arguments	Description
<code>policy</code>	#	1	Name of the policy to modify.
<code>key</code>	#	2	Name of the key to modify.
<code>value</code>	#	3	New value for the key.
<code>no_replace</code>			true to cancel the update operation if the specified key is already present, or false.

Sample command

To update a policy's node metadata for the `my_key` value:

```
razor update-policy-node-metadata --policy policy1 --key my_key --value my_value
```

delete-policy

The `delete-policy` command deletes a policy from the Razor database. Nodes bound to the policy aren't re-provisioned.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the policy to delete.

Sample command

To delete an obsolete policy:

```
razor delete-policy --name my_obsolete_policy
```

Broker commands

Broker commands enable you to create a new broker configuration, set or clear a specified key value for a broker, and delete a specified broker.

create-broker

The `create-broker` command creates a new broker configuration used to hand off management of nodes.

If you're using Puppet Enterprise for node management, use the `puppet-pe` broker type.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name for the new broker.
broker-type	#	2	Usually <code>puppet-pe</code> . Other valid values are <code>puppet</code> , to transfer control to an open source Puppet master, and <code>noop</code> , to leave the node unmanaged.
configuration			Configuration details specific to the <code>broker_type</code> . For <code>puppet-pe</code> brokers, properties include: <ul style="list-style-type: none"> <code>server</code> — Host name of the Puppet master. <code>version</code> — The agent version to install. The default value is current. <code>ntpdate_server</code> — URL for an NTP server, such as <code>us.pool.ntp.org</code>, used to synchronize the date and time before installing the Puppet agent.

Sample command

To create a simple `puppet-pe` broker:

```
razor create-broker --name puppet-pe -c server=puppet.example.org \
```

```
-c version=2015.3
```

update-broker-configuration

The `update-broker-configuration` command sets or clears a specified key value for a broker.

Command attributes

Attributes	Required	Positional arguments	Description
broker	#	1	Name of the broker to update.
key	#	2	Name of the key to modify in the broker's configuration file. For puppet-pe brokers, this is usually <code>server</code> or <code>version</code> .
value		3	New value to use for the key. This attribute can't be used with <code>clear</code> .
clear			<code>true</code> to clear the value of the specified key, or <code>false</code> . This attribute can't be used with <code>value</code> .

Sample command

To change the key: `some_key` to `new_value`:

```
razor update-broker-configuration --broker mybroker --key some_key --value new_value
```

delete-broker

The `delete-broker` command deletes a specified broker from the Razor database.

Note: Before deleting a broker, remove any references to it in policies. This command fails if a policy is using the broker.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the broker to delete.

Sample command

To delete an obsolete broker:

```
razor delete-broker --name my_obsolete_broker
```


Related information

[Create a new broker type](#) on page 667

To create a broker called `sample`:

Hook commands

Hooks are custom, executable scripts that are triggered to run when a node hits certain phases in its lifecycle. A hook script receives several properties as input, and can make changes in the node's metadata or the hook's internal configuration.

Hooks can be useful for:

- Notifying an external system about the stage of a node's installation.
- Querying external systems for information that modifies how a node gets installed.
- Calculating complex values for use in a node's installation configuration.

create-hook

The `create-hook` command enables a hook to run when specified events occur.

Command attributes

Attributes	Required	Positional arguments	Description
<code>name</code>	#	1	Name of the new hook.
<code>hook_type</code>	#	2	Type of hook that the new hook is based on. For available hook types on your server, run <code>razor create-hook --help</code> .
<code>configuration</code>			<p>Configuration settings as required by the <code>hook_type</code>.</p> <p>This argument sets the initial configuration values for a hook. Configuration values can change as the hook is executed based on events or the <code>update-hook-configuration</code> command.</p> <p>This attribute can be abbreviated as <code>-c</code>.</p>

Sample command

To create a simple hook:

```
razor create-hook --name myhook --hook-type some_hook --configuration foo=7
```

run-hook

The `run-hook` command executes a hook with parameters you specify. This command is useful to test a hook you're writing, or to re-run a hook that failed.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the hook to run.
event	#		Name of the event to trigger the hook.
node	#		Name of the node you want to run the hook on.
policy			Name of the policy you want to use as input to the hook script when the hook runs. This attribute applies only to events that affect policies, for example <code>node-bound-to-policy</code> and <code>node-unbound-from-policy</code> .
debug			<code>true</code> to include debug information in the event log, or <code>false</code> . If omitted, debug information is not logged.

Sample command

To run a hook called `counter` when a node boots:

```
razor run-hook --name counter --event node-booted --node node1
```

update-hook-configuration

The `update-hook-configuration` command sets or clears a specified key value for a hook. This can be useful to iteratively test a hook you're writing. For example, you can write the hook, test it with `run-hook`, then modify the hook as needed with `update-hook-configuration`.

Command attributes

Attributes	Required	Positional arguments	Description
hook	#	1	Name of the hook to update.
key	#	2	Name of the key to modify in the hook's configuration file.
value	*	3	Value to change the key to.
clear	*		<code>true</code> to clear the value of the specified key, or <code>false</code> .

* You must specify `value` or `clear`.

Sample command

To change the key `some_key` to `new_value`:

```
razor update-hook-configuration --hook hook1 \
  --key my_key --value new_value
```

delete-hook

The `delete-hook` command deletes a specified hook from the Razor database.

Command attributes

Attributes	Required	Positional arguments	Description
name	#	1	Name of the hook to delete.

Sample command

To delete an obsolete hook:

```
razor delete-hook --name my_obsolete_hook
```

Related information

[Available events](#)

Protecting existing nodes

In *brownfield environments* – those in which you already have machines installed that PXE boot against the Razor server – you must take extra precautions to protect existing nodes. Failure to adequately protect existing nodes can result in data loss.

For recommended provisioning workflows in an existing environment, see [Provisioning for advanced users](#).

Protecting new nodes

By default, Razor marks all newly discovered nodes as installed, which prevents modifications to the node. This default is controlled with the `protect_new_nodes` class parameter of the **pe_razor** class.

With `protect_new_nodes` enabled, Razor considers installed nodes eligible for reinstallation only when the installed flag is removed from the node using the `reinstall-node` command.

With `protect_new_nodes` disabled, Razor considers any nodes it detects – including installed nodes – eligible for provisioning. You might choose to disable `protect_new_nodes` if:

- You're sure all nodes in your environment need to be provisioned or reprovisioned.
- You've manually registered existing nodes that you want to protect.

The `protect_new_nodes` option is specified as a class parameter of the **pe_razor** class.

Related information

[Register existing nodes manually](#) on page 651

If you're provisioning in an environment with existing nodes already installed, register the nodes to prevent Razor from re-provisioning them.

[Reinstall the node](#) on page 648

By default, Razor protects existing nodes from reprovisioning by marking all existing nodes as installed. You must specifically instruct the server to reinstall the node in order to trigger provisioning.

[Change the `protect_new_nodes` default](#) on page 652

Because you've already registered existing nodes to protect them from reprovisioning, it's now safe to change the `protect_new_nodes` default to `false`. This removes the `installed` flag from unregistered nodes so that Razor can provision them.

Registering nodes

To identify existing nodes to the Razor server – and prevent reprovisioning – you can manually register nodes using the `register-node` command. The `register-node` command identifies a node as installed, which signals Razor to ignore the node.

To successfully register nodes, you must provide enough `hw-info` details for Razor to identify the nodes when they're detected.

Related information

[Register existing nodes manually](#) on page 651

If you're provisioning in an environment with existing nodes already installed, register the nodes to prevent Razor from re-provisioning them.

Limiting the number of nodes a policy can bind to

You can use the `max_count` attribute for policies to limit the number of slots available for provisioning.

For example, at initial installation, no slots are available, so no machines are provisioned. At this point, you can examine your resource pool or mark specific nodes as registered. If you create a new policy with a value of 1 for `max_count`, there's now one slot available for provisioning. The first qualified node that checks in binds to the policy while all other nodes remain unprovisioned.

Provisioning a *nix node

Provisioning deploys and installs your chosen operating system to target nodes.

What triggers provisioning

There are four requirements for Razor to provision a node.

- The node must boot with iPXE software.
- The node's network must link to the Razor server through TFTP.
- A Razor policy must match the node.
- The node's `installed` flag must be set to `false`.

When these conditions are met, Razor recognizes the node, applies the first matching policy in the policy table, and provisions the node.

With these requirements in mind, you can modify your Razor workflow to suit your goals, your environment, and your familiarity with Razor.

- [Provision for new users](#) on page 644 enables you to learn about Razor and verify tags before provisioning nodes.
- [Provision for advanced users](#) on page 648 enables you to seamlessly provision nodes in an existing environment.

Provision for new users

This workflow enables you to learn about Razor and verify tags before provisioning nodes.

Before you begin

You're ready to provision a node after you configure:

- A DHCP/DNS/TFTP service with SELinux configured to enable PXE boot
-

- The server and client

To follow along with the examples in these workflows, you must have a new node with at least 1GB (2GB recommended) of memory. Don't boot the node before you begin the provisioning process.

In this workflow, you load iPXE software and register nodes with the microkernel so you can view node details. Then you configure objects, finishing with creating a policy. Provisioning is triggered when you reinstall the node in order to remove the `installed` flag.

The examples in this workflow demonstrate provisioning a sample node with 6.7. You can modify the settings and scale up your workflow as needed for your environment.

Load iPXE software

Set your machines to PXE boot so that Razor can interact with the node and provision the operating system. This process uses both the `undionly.kpxe` file from the iPXE open source software stack and a `Razorbootstrap.ipxe` script.

1. Download the iPXE boot image <http://boot.ipxe.org/undionly.kpxe> and copy the image to your TFTP server's `/var/lib/tftpboot` directory:

```
`cp undionly.kpxe /var/lib/tftpboot`
```

2. Download the iPXE bootstrap script from the server and copy the script to your TFTP server's `/var/lib/tftpboot` directory:

```
`wget "https://$<RAZOR_HOSTNAME>:$<HTTPS_PORT>/api/microkernel/
bootstrap?nic_max=1&http_port=$<HTTP_PORT>" -O /var/lib/tftpboot/
bootstrap.ipxe`
```

Note: Don't use `localhost` as the name of the host. The bootstrap script chain-loads the next iPXE script from the server, so it must contain the correct host name.

Register a node with the microkernel

Registering a node lets you learn about the node before Razor provisions it. With registered nodes, you can view facts about the node, add metadata to the node, and see which tags the node matches.

1. Boot the node. This can mean physically pressing the power button, using IPMI to manage the node's power state, or, in the case of a VM, starting the VM.

The node boots into the microkernel and Razor discovers the node. After the initial PXE boot, the node downloads the microkernel.

2. On the Razor server, view the new node: `razor nodes`

The output displays the new node ID and name, for example:

```
id: "http://localhost:8150/api/collections/node/node1"
name: "node1"
spec: "/razor/v1/collections/nodes/member"
```

3. View the node's details: `razor nodes <NODE_NAME>`

The output displays hardware and DHCP information, the path to the log, and a placeholder for tags, for example:

```
hw_info:
  mac: ["08-00-27-8a-5e-5d"]
  serial: "0"
  uuid: "9a717dc3-2392-4853-89b9-27fec1aec7b2"
dhcp_mac: "08-00-27-8a-5e-5d"
log:
  log => http://localhost:8150/api/collections/node/node1/log
tags: []
```

4. When the microkernel is running on the machine, view facts about the node: `razor nodes <NODE_NAME> facts`
 Facter periodically sends facts back to the server about the node, including IP address, details about network cards, and block devices.

Create a repository

Repositories contain – or point to – the operating system to install on a node.

You can create three types of repositories using specific attributes:

- `url` – Points to content available on another server, for example, on a mirror that you maintain.
- `iso-url` – Downloads and unpacks an ISO on the Razor server.
- `no_content` – Creates a stub directory on the Razor server that you can manually fill with content.

To download a CentOS 6.7 ISO and create a repository from it:

```
razor create-repo --name centos-6.7 --task centos
--iso-url http://centos.sonn.com/6.7/isos/x86_64/CentOS-6.7-x86_64-bin-DVD1.iso
```

The ISO is downloaded onto the Razor server, then extracted to the repository. This can take some time to complete. To monitor progress, you can run `razor commands` to view the task status or `ls -al /tmp` to see the downloaded file size.

Related information

[Repositories](#) on page 660

A repository is where you store all of the actual bits used by Razor to install a node. Or, in some cases, the external location of bits that you link to. A repo is identified by a unique name.

[Repository commands](#) on page 626

Repository commands enable you to create and delete specified repositories from the Razor database and specify the task that installs the contents of a repository.

[Repositories API](#) on page 680

These commands enable you to create a new repository or delete a repository from the internal Razor database. You can also ensure that a specified repository uses a specified task.

(Optional) Create tags

Tags let you group nodes based on their characteristics. You can then apply policies based on tags to install appropriate operating systems on tagged nodes. If you don't specify tags for a policy, the policy binds to any node.

1. To create a tag called `small` with a rule that matches machines that have less than 4GB of memory:

```
razor create-tag --name small
--rule '["<", ["num", ["fact", "memorysize_mb"]], 4128]'
```

2. (Optional) Inspect the tag on the server: `razor tags <TAG_NAME>`

For example, `razor tags small` responds with:

```
From https://razor:8151/api/collections/tags/small:
  name: small
  rule: ["<", ["num", ["fact", "memorysize_mb"]], 4128]
  nodes: 1
  policies: 0
```

3. (Optional) Confirm that expected nodes now have the tag: `razor tags <TAG_NAME> nodes`

For example, `razor tags small nodes` displays a table of registered nodes that have less than 4GB of memory.

Tip: To see details about the policies associated with a tag, run `razor tags <TAG_NAME> policies`. To see its rule, run `razor tags <TAG_NAME> rule`.

Related information

[Tags](#) on page 663

A tag consists of a unique name and a rule. Tags match a node if evaluating the node against the tag's facts results in `true`. Tag matching is case sensitive.

[Tag commands](#) on page 631

Tag commands enable you to create new tags, set the rules used to apply the tag to nodes, change the rule of a specified tag, or delete a specified tag.

[Tags API](#) on page 682

These commands enable you to create a tag, delete a tag, or change the rule for a tag.

(Optional) Create a broker

Brokers hand off nodes to configuration management systems like Puppet Enterprise.

To hand off Razor nodes to a Puppet master at `puppet-master.example.com`:

```
razor create-broker --name pe --broker-type puppet-pe
  --configuration server=puppet-master.example.com
```

Related information

[Brokers](#) on page 666

Brokers hand off nodes to configuration management systems like Puppet Enterprise. Brokers consist of two parts: a broker type and information specific to the broker type.

[Broker commands](#) on page 639

Broker commands enable you to create a new broker configuration, set or clear a specified key value for a broker, and delete a specified broker.

[Brokers API](#) on page 686

A broker is responsible for configuring a newly installed node for a specific configuration management system. For Puppet Enterprise, you generally use brokers only with the type `puppet-pe`.

Add the `pe_repo` class to the PE Master node group

To manage a node handed off by the broker, the master must include a class that matches the node's architecture.

Note: Skip this step if the node you're provisioning has the same architecture as your master, or if the master already includes a `pe_repo` class that matches the node's architecture.

1. In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Master** group.
2. On the **Configuration** tab in the **Add new class** field, select `pe_repo::platform::<VERSION>`.

For example, `pe_repo::platform::el_7_x86_64`.

3. Click **Add class** and then commit changes.
4. On the master, run Puppet.

Related information

[Running Puppet on nodes](#) on page 320

Puppet automatically attempts to run on each of your nodes every 30 minutes. To trigger a Puppet run outside of the default 30-minute interval, you can manually run Puppet.

Create a policy

Policies tell Razor what operating system to install on the provisioned node, where to get the OS software, how to configure it, and how to communicate between the node and Puppet Enterprise.

To create a policy that installs CentOS on machines tagged with `small`, then hands them off to Puppet Enterprise for management:

```
razor create-policy --name centos-for-small
  --repo centos-6.7 --broker pe --tag small
  --hostname 'host${id}.example.com' --root-password secret
```

Note: You can view details of a specific policy by running `razor policies <POLICY_NAME>`. You can view a table of all policies by running `razor policies`. The order in which policies are listed in the table is important because Razor applies the first matching policy to a node.

Related information

[Policies](#) on page 665

Policies tell Razor what bits to install, where to get the bits, how to configure them, and how the installed node can communicate with Puppet Enterprise.

[Policy commands](#) on page 632

Policies govern how nodes are provisioned.

[Policies API](#) on page 683

Policies govern how nodes are provisioned depending on how they are tagged.

Reinstall the node

By default, Razor protects existing nodes from reprovisioning by marking all existing nodes as installed. You must specifically instruct the server to reinstall the node in order to trigger provisioning.

You can skip this step if you change the `protect_new_nodes` option to `false`, which allows Razor to provision a node as soon as it PXE boots with a matching policy. Be sure you understand how the `protect_new_nodes` option works before changing it, however. Failure to protect existing nodes can result in data loss.

Reinstall the node: `razor reinstall-node <NODE_NAME>`

When you reinstall the node, Razor clears the `installed` flag and the node restarts and boots into the microkernel. The microkernel reports its facts, and Razor provisions the node by applying the first applicable policy in the policy table.

When provisioning is complete, you can log into the node using the `root_password` as specified by the node's metadata, or by the policy that the node is bound to. You can also see the node and its details in the console, and manage it there as you would any other node.

Provision for advanced users

This workflow enables you to seamlessly provision nodes in an existing environment.

Before you begin

You're ready to provision a node after you configure:

- A DHCP/DNS/TFTP service with SELinux configured to enable PXE boot
-
- The server and client

To follow along with the examples in these workflows, you must have a new node with at least 1GB (2GB recommended) of memory. Don't boot the node before you begin the provisioning process.

In this workflow, you configure Razor objects, register any existing nodes to prevent accidentally overwriting them, and finally, load iPXE so that nodes boot through Razor. Provisioning is triggered when the node PXE boots with a matching policy in place.

The examples in this workflow demonstrate provisioning a sample node with 6.7. You can modify the settings and scale up your workflow as needed for your environment.

Create a repository

Repositories contain – or point to – the operating system to install on a node.

You can create three types of repositories using specific attributes:

- `url` – Points to content available on another server, for example, on a mirror that you maintain.
- `iso-url` – Downloads and unpacks an ISO on the Razor server.
- `no_content` – Creates a stub directory on the Razor server that you can manually fill with content.

To download a CentOS 6.7 ISO and create a repository from it:

```
razor create-repo --name centos-6.7 --task centos
--iso-url http://centos.sonn.com/6.7/isos/x86_64/CentOS-6.7-x86_64-bin-DVD1.iso
```

The ISO is downloaded onto the Razor server, then extracted to the repository. This can take some time to complete. To monitor progress, you can run `razor commands` to view the task status or `ls -al /tmp` to see the downloaded file size.

Related information

[Repositories](#) on page 660

A repository is where you store all of the actual bits used by Razor to install a node. Or, in some cases, the external location of bits that you link to. A repo is identified by a unique name.

[Repository commands](#) on page 626

Repository commands enable you to create and delete specified repositories from the Razor database and specify the task that installs the contents of a repository.

[Repositories API](#) on page 680

These commands enable you to create a new repository or delete a repository from the internal Razor database. You can also ensure that a specified repository uses a specified task.

(Optional) Create tags

Tags let you group nodes based on their characteristics. You can then apply policies based on tags to install appropriate operating systems on tagged nodes. If you don't specify tags for a policy, the policy binds to any node.

1. To create a tag called `small` with a rule that matches machines that have less than 4GB of memory:

```
razor create-tag --name small
--rule '["<", ["num", ["fact", "memorysize_mb"]], 4128]'
```

2. (Optional) Inspect the tag on the server: `razor tags <TAG_NAME>`

For example, `razor tags small` responds with:

```
From https://razor:8151/api/collections/tags/small:
  name: small
  rule: ["<", ["num", ["fact", "memorysize_mb"]], 4128]
  nodes: 1
  policies: 0
```

3. (Optional) Confirm that expected nodes now have the tag: `razor tags <TAG_NAME> nodes`

For example, `razor tags small nodes` displays a table of registered nodes that have less than 4GB of memory.

Tip: To see details about the policies associated with a tag, run `razor tags <TAG_NAME> policies`. To see its rule, run `razor tags <TAG_NAME> rule`.

Related information

[Tags](#) on page 663

A tag consists of a unique name and a rule. Tags match a node if evaluating the node against the tag's facts results in `true`. Tag matching is case sensitive.

[Tag commands](#) on page 631

Tag commands enable you to create new tags, set the rules used to apply the tag to nodes, change the rule of a specified tag, or delete a specified tag.

[Tags API](#) on page 682

These commands enable you to create a tag, delete a tag, or change the rule for a tag.

(Optional) Create a broker

Brokers hand off nodes to configuration management systems like Puppet Enterprise.

To hand off Razor nodes to a Puppet master at `puppet-master.example.com`:

```
razor create-broker --name pe --broker-type puppet-pe
  --configuration server=puppet-master.example.com
```

Related information

[Brokers](#) on page 666

Brokers hand off nodes to configuration management systems like Puppet Enterprise. Brokers consist of two parts: a broker type and information specific to the broker type.

[Broker commands](#) on page 639

Broker commands enable you to create a new broker configuration, set or clear a specified key value for a broker, and delete a specified broker.

[Brokers API](#) on page 686

A broker is responsible for configuring a newly installed node for a specific configuration management system. For Puppet Enterprise, you generally use brokers only with the type `puppet-pe`.

Add the `pe_repo` class to the PE Master node group

To manage a node handed off by the broker, the master must include a class that matches the node's architecture.

Note: Skip this step if the node you're provisioning has the same architecture as your master, or if the master already includes a `pe_repo` class that matches the node's architecture.

1. In the console, click **Classification**, and in the **PE Infrastructure** group, select the **PE Master** group.
2. On the **Configuration** tab in the **Add new class** field, select `pe_repo::platform::<VERSION>`.
For example, `pe_repo::platform::el_7_x86_64`.
3. Click **Add class** and then commit changes.
4. On the master, run Puppet.

Related information

[Running Puppet on nodes](#) on page 320

Puppet automatically attempts to run on each of your nodes every 30 minutes. To trigger a Puppet run outside of the default 30-minute interval, you can manually run Puppet.

Create a policy

Policies tell Razor what operating system to install on the provisioned node, where to get the OS software, how to configure it, and how to communicate between the node and Puppet Enterprise.

To create a policy that installs CentOS on machines tagged with `small`, then hands them off to Puppet Enterprise for management:

```
razor create-policy --name centos-for-small
  --repo centos-6.7 --broker pe --tag small
  --hostname 'host${id}.example.com' --root-password secret
```

Note: You can view details of a specific policy by running `razor policies <POLICY_NAME>`. You can view a table of all policies by running `razor policies`. The order in which policies are listed in the table is important because Razor applies the first matching policy to a node.

Related information

[Policies](#) on page 665

Policies tell Razor what bits to install, where to get the bits, how to configure them, and how the installed node can communicate with Puppet Enterprise.

[Policy commands](#) on page 632

Policies govern how nodes are provisioned.

[Policies API](#) on page 683

Policies govern how nodes are provisioned depending on how they are tagged.

Register existing nodes manually

If you're provisioning in an environment with existing nodes already installed, register the nodes to prevent Razor from re-provisioning them.

You must provide enough `hw-info` details so that nodes can be identified when Razor detects them.

To register an existing node, and indicate with the `installed` attribute that the node isn't eligible for provisioning:

```
razor register-node --hw-info net0=78:31:c1:be:c8:00 \
  --hw-info net1=72:00:01:f2:13:f0 \
  --hw-info net2=72:00:01:f2:13:f1 \
  --hw-info serial=xxxxxxxxxxxx \
  --hw-info asset=Asset-1234567890 \
  --hw-info uuid="Not Settable" \
  --installed
```

Related information

[Protecting existing nodes](#) on page 643

In *brownfield environments* – those in which you already have machines installed that PXE boot against the Razor server – you must take extra precautions to protect existing nodes. Failure to adequately protect existing nodes can result in data loss.

[Node commands](#) on page 620

Nodes are created either through the node boot endpoint, when the node initiates its first web request to the Razor server, or through the `register-node` command.

[Nodes API](#) on page 677

These commands enable you to register a node, set a node's hardware information, remove a single node, or remove a node's associate with any policies and clear its installed flag.

Change the `protect_new_nodes` default

Because you've already registered existing nodes to protect them from reprovisioning, it's now safe to change the `protect_new_nodes` default to `false`. This removes the installed flag from unregistered nodes so that Razor can provision them.

1. In the console, select **Nodes > Classification**, then click the `server` node group.
2. On the **Configuration** tab, select the `protect_new_nodes` parameter, then in the **Value** field, enter `false`.
3. Commit changes, run `pe-razor-server`, and then restart the `pe-razor-server` service.

Related information

[Running Puppet on nodes](#) on page 320

Puppet automatically attempts to run on each of your nodes every 30 minutes. To trigger a Puppet run outside of the default 30-minute interval, you can manually run Puppet.

Load iPXE software

Set your machines to PXE boot so that Razor can interact with the node and provision the operating system. This process uses both the `undionly.kpxe` file from the iPXE open source software stack and a Razor-specific `bootstrap.ipxe` script.

1. Download the iPXE boot image <http://boot.ipxe.org/undionly.kpxe> and copy the image to your TFTP server's `/var/lib/tftpboot` directory:

```
`cp undionly.kpxe /var/lib/tftpboot`
```

2. Download the iPXE bootstrap script from the `server` and copy the script to your TFTP server's `/var/lib/tftpboot` directory:

```
`wget "https://$<RAZOR_HOSTNAME>:$<HTTPS_PORT>/api/microkernel/
bootstrap?nic_max=1&http_port=$<HTTP_PORT>" -O /var/lib/tftpboot/
bootstrap.ipxe`
```

Note: Don't use `localhost` as the name of the `host`. The bootstrap script chain-loads the next iPXE script from the server, so it must contain the correct host name.

3. Boot the node. This can mean physically pressing the power button, using IPMI to manage the node's power state, or, in the case of a VM, starting the VM.

When the node PXE boots with a policy in place, Razor detects the node and provisions it by applying the first applicable policy in the policy table.

When provisioning is complete, you can log into the node using the `root_password` as specified by the node's metadata, or by the policy that the node is bound to. You can also see the node and its details in the console, and manage it there as you would any other node.

Viewing information about nodes

Use these commands to view details about nodes in your environment.

Command	Result
<code>razor nodes</code>	Displays a list of nodes that Razor knows about.
<code>razor nodes <NODE_NAME></code>	Displays details about the specified node.
<code>razor nodes <NODE_NAME> log</code>	Displays a log that includes the timing and status of installation events, as well as downloads of kickstart files and post-install scripts.

Provisioning a Windows node

Provisioning deploys and installs your chosen operating system to target nodes.

What triggers provisioning

There are four requirements for Razor to provision a node.

- The node must boot with iPXE software.
- The node's network must link to the Razor server through TFTP.
- A Razor policy must match the node.
- The node's `installed` flag must be set to `false`.

When these conditions are met, Razor recognizes the node, applies the first matching policy in the policy table, and provisions the node.

With these requirements in mind, you can modify your Razor workflow to suit your goals, your environment, and your familiarity with Razor.

- [Provision for new users](#) on page 644 enables you to learn about Razor and verify tags before provisioning nodes.
- [Provision for advanced users](#) on page 648 enables you to seamlessly provision nodes in an existing environment.

Provision a Windows node

This Windows workflow adapts the provisioning for new users workflow. This process enables you to learn about Razor and verify tags before provisioning nodes.

Before you begin

You're ready to provision a node after you configure:

- A DHCP/DNS/TFTP service with SELinux configured to enable PXE boot
- The server and client

To provision Windows machines, you also need:

- A Windows machine running the same OS that you plan to provision. This machine is used to create a WinPE image.
- (Optional) An activation key for the OS. A trial license is used if you don't have an activation key.

To follow along with the examples in this workflow, you must also have a new node with at least 8GB of memory. Don't boot the node before you begin the provisioning process.

In this workflow, you load iPXE software and register nodes with the microkernel so you can view node details. Then you configure objects, finishing with creating a policy. Provisioning is triggered when you reinstall the node in order to remove the `installed` flag.

The examples in this workflow demonstrate provisioning a sample node with Windows. You can modify the settings and scale up your workflow as needed for your environment.

Configure SMB share

Because neither the WinPE environment nor the Windows installer can use an HTTP source for installation, you must use a server message block (SMB) server to store the Razor repositories.

You can configure SMB share automatically, using a Razor class parameter, or manually.

Automatically configure SMB share

Enable the SMB share class parameter on the Razor server to let Razor set up the SMB share automatically.

1. In the console, select **Nodes > Classification**, then click the `server` node group.
2. On the **Configuration** tab, enable the SMB share parameter.

Parameter	Value
<code>enable_windows_smb</code>	<code>true</code>

Note: If you change `enable_windows_smb` from `true` to `false` later, the share remains enabled but isn't managed by Puppet.

3. Commit changes, run `puppet`, and then restart the `pe-razor-server` service.

Related information

[Running Puppet on nodes](#) on page 320

Puppet automatically attempts to run on each of your nodes every 30 minutes. To trigger a Puppet run outside of the default 30-minute interval, you can manually run Puppet.

Manually configure SMB share

If Samba is already installed on your Razor server, you can manually configure it to work with Razor.

1. Navigate to the Samba directory: `cd /etc/samba`
2. Edit the `smb.conf` file:
 - a) Modify the network settings as necessary for your environment.
 - b) Edit the `global` service definition to [allow unauthenticated access](#).

```
[global]
    security      = user
    map to guest  = bad user
```

- c) Add a service definition that allows anonymous access and points to the `repo_store_root` class parameter of the `pe_razor` module.

```
[razor]
    comment      = Windows Installers
    path         = /opt/puppetlabs/server/data/razor-server/repo
    guest ok     = yes
    writable    = no
    browsable    = yes
```

3. Restart Samba: `service smb restart`

Load iPXE software

Set your machines to PXE boot so that Razor can interact with the node and provision the operating system. This process uses both the `undionly.kpxe` file from the iPXE open source software stack and a `Razorbootstrap.ipxe` script.

1. Download the iPXE boot image <http://boot.ipxe.org/undionly.kpxe> and copy the image to your TFTP server's `/var/lib/tftpboot` directory:

```
`cp undionly.kpxe /var/lib/tftpboot`
```

2. Download the iPXE bootstrap script from the server and copy the script to your TFTP server's `/var/lib/tftpboot` directory:

```
`wget "https://$<RAZOR_HOSTNAME>:$<HTTPS_PORT>/api/microkernel/
bootstrap?nic_max=1&http_port=$<HTTP_PORT>" -O /var/lib/tftpboot/
bootstrap.ipxe`
```

Note: Don't use `localhost` as the name of the host. The bootstrap script chain-loads the next iPXE script from the server, so it must contain the correct host name.

Register a node with the microkernel

Registering a node lets you learn about the node before Razor provisions it. With registered nodes, you can view facts about the node, add metadata to the node, and see which tags the node matches.

1. Boot the node. This can mean physically pressing the power button, using IPMI to manage the node's power state, or, in the case of a VM, starting the VM.

The node boots into the microkernel and Razor discovers the node. After the initial PXE boot, the node downloads the microkernel.

2. On the Razor server, view the new node: `razor nodes`

The output displays the new node ID and name, for example:

```
id: "http://localhost:8150/api/collections/node/node1"
name: "node1"
spec: "/razor/v1/collections/nodes/member"
```

3. View the node's details: `razor nodes <NODE_NAME>`

The output displays hardware and DHCP information, the path to the log, and a placeholder for tags, for example:

```
hw_info:
  mac: ["08-00-27-8a-5e-5d"]
  serial: "0"
  uuid: "9a717dc3-2392-4853-89b9-27fec1aec7b2"
  dhcp_mac: "08-00-27-8a-5e-5d"
  log:
    log => http://localhost:8150/api/collections/node/node1/log
  tags: []
```

4. When the microkernel is running on the machine, view facts about the node: `razor nodes <NODE_NAME> facts`

Facter periodically sends facts back to the server about the node, including IP address, details about network cards, and block devices.

Build a WinPE image

Create a custom Windows Preinstallation Environment (WinPE) WIM image containing Razor scripts.

1. On an existing Windows machine running the same operating system that you plan to install, install the [Windows assessment and deployment kit](#) in the default location.
2. Copy the directory at `/opt/puppetlabs/server/apps/razor-server/share/razor-server/build-winpe` from the Razor server to the existing Windows machine. If the directory is absent from your Razor server, use [this archive](#).
3. Change into the directory you created in the previous step, for example, `c:\build-winpe`
4. (Optional) To include additional Windows drivers in the WIM image – for example, if your machines require proprietary UCS drivers – place `.inf` files in the `extra-drivers` folder inside the `build-winpe` directory.

You can place individual files or directories in the `extra-drivers` folder.

5. Run one of these build scripts, depending on whether you're using unsigned drivers:

- Standard build script

```
powershell -executionpolicy bypass -noninteractive -file build-razor-
winpe.ps1 -razorurl http://razor:8150/svc
```

- Build script for unsigned drivers

```
powershell -executionpolicy bypass -noninteractive -file build-razor-
winpe.ps1 -razorurl http://razor:8150/svc -allowunsigned
```

When the build script completes, a `razor-winpe.wim` image appears in a new `razor-winpe` directory inside the current working directory.

Create a repository and add the WinPE image

Because Razor can't unpack Windows DVD images, you must create a stub repository and manually fill it with content.

1. Copy the Windows ISO image to the Razor server.
2. Create an empty repository.

For example:

```
razor create-repo --name win2012r2 --task windows/2012r2
--no-content true
```

3. After the repository is created, log into your Razor server as root, change into the repository directory, `/opt/puppetlabs/server/data/razor-server/<REPO_NAME>`, then mount the ISO image:

```
$ mount -o loop </PATH/TO/WINDOWS_SERVER.ISO> /mnt
$ cp -pr /mnt/* <REPO_NAME>
$ umount /mnt
$ chown -R pe-razor: <REPO_NAME>
```

Note: The repository directory is specified by the `repo_store_root` class parameter of the `pe_razor` module. Your directory might be different from the default if you customized this parameter.

4. Copy the WinPE image from the existing Windows machine to the repository directory on the Razor server.

Related information

[Repositories](#) on page 660

A repository is where you store all of the actual bits used by Razor to install a node. Or, in some cases, the external location of bits that you link to. A repo is identified by a unique name.

[Repository commands](#) on page 626

Repository commands enable you to create and delete specified repositories from the Razor database and specify the task that installs the contents of a repository.

[Repositories API](#) on page 680

These commands enable you to create a new repository or delete a repository from the internal Razor database. You can also ensure that a specified repository uses a specified task.

(Optional) Create tags

Tags let you group nodes based on their characteristics. You can then apply policies based on tags to install appropriate operating systems on tagged nodes. If you don't specify tags for a policy, the policy binds to any node.

1. To create a tag called `small` with a rule that matches machines that have less than 4GB of memory:

```
razor create-tag --name small
--rule '["<", ["num", ["fact", "memorysize_mb"]], 4128]'
```


2. (Optional) Inspect the tag on the server: `razor tags <TAG_NAME>`

For example, `razor tags small` responds with:

```
From https://razor:8151/api/collections/tags/small:
  name: small
  rule: ["<", ["num", ["fact", "memorysize_mb"]], 4128]
  nodes: 1
  policies: 0
```

3. (Optional) Confirm that expected nodes now have the tag: `razor tags <TAG_NAME> nodes`

For example, `razor tags small nodes` displays a table of registered nodes that have less than 4GB of memory.

Tip: To see details about the policies associated with a tag, run `razor tags <TAG_NAME> policies`. To see its rule, run `razor tags <TAG_NAME> rule`.

Related information

[Tags](#) on page 663

A tag consists of a unique name and a rule. Tags match a node if evaluating the node against the tag's facts results in true. Tag matching is case sensitive.

[Tag commands](#) on page 631

Tag commands enable you to create new tags, set the rules used to apply the tag to nodes, change the rule of a specified tag, or delete a specified tag.

[Tags API](#) on page 682

These commands enable you to create a tag, delete a tag, or change the rule for a tag.

(Optional) Create a broker

Brokers hand off nodes to configuration management systems like Puppet Enterprise.

To hand off Razor nodes to a Puppet master at `puppet-master.example.com`:

```
razor create-broker --name pe --broker-type puppet-pe
--configuration server=puppet-master.example.com
```

Related information

[Brokers](#) on page 666

Brokers hand off nodes to configuration management systems like Puppet Enterprise. Brokers consist of two parts: a broker type and information specific to the broker type.

[Broker commands](#) on page 639

Broker commands enable you to create a new broker configuration, set or clear a specified key value for a broker, and delete a specified broker.

[Brokers API](#) on page 686

A broker is responsible for configuring a newly installed node for a specific configuration management system. For Puppet Enterprise, you generally use brokers only with the type `puppet-pe`.

Create a policy

Policies tell Razor what operating system to install on the provisioned node, where to get the OS software, how to configure it, and how to communicate between the node and Puppet Enterprise.

To create a policy that installs CentOS on machines tagged with `small`, then hands them off to Puppet Enterprise for management:

```
razor create-policy --name centos-for-small
--repo centos-6.7 --broker pe --tag small
--hostname 'host${id}.example.com' --root-password secret
```

Note: You can view details of a specific policy by running `razor policies <POLICY_NAME>`. You can view a table of all policies by running `razor policies`. The order in which policies are listed in the table is important because Razor applies the first matching policy to a node.

Related information

[Policies](#) on page 665

Policies tell Razor what bits to install, where to get the bits, how to configure them, and how the installed node can communicate with Puppet Enterprise.

[Policy commands](#) on page 632

Policies govern how nodes are provisioned.

[Policies API](#) on page 683

Policies govern how nodes are provisioned depending on how they are tagged.

Reinstall the node

By default, Razor protects existing nodes from reprovisioning by marking all existing nodes as installed. You must specifically instruct the server to reinstall the node in order to trigger provisioning.

You can skip this step if you change the `protect_new_nodes` option to `false`, which allows Razor to provision a node as soon as it PXE boots with a matching policy. Be sure you understand how the `protect_new_nodes` option works before changing it, however. Failure to protect existing nodes can result in data loss.

Reinstall the node: `razor reinstall-node <NODE_NAME>`

When you reinstall the node, Razor clears the `installed` flag and the node restarts and boots into the microkernel. The microkernel reports its facts, and Razor provisions the node by applying the first applicable policy in the policy table.

When provisioning is complete, you can log into the node using the `root_password` as specified by the node's metadata, or by the policy that the node is bound to. You can also see the node and its details in the console, and manage it there as you would any other node.

Viewing information about nodes

Use these commands to view details about nodes in your environment.

Command	Result
<code>razor nodes</code>	Displays a list of nodes that Razor knows about.
<code>razor nodes <NODE_NAME></code>	Displays details about the specified node.
<code>razor nodes <NODE_NAME> log</code>	Displays a log that includes the timing and status of installation events, as well as downloads of kickstart files and post-install scripts.

Provisioning with custom facts

You can use a microkernel extension to provision nodes based on hardware info or metadata that isn't available by default in Facter.

When configured, all nodes receive the microkernel extension, which contains instructions for reporting custom facts to the Razor server. The custom facts can then be used to tag nodes, apply policies, and provision eligible nodes.

For example, if you want to provision machines based on hardware chassis or rack location – facts not available by default in Facter – you could create custom facts for these properties. Then, you can use the custom facts to create associated tags and policies that install the appropriate OS. For a detailed example of using a microkernel extension to report rack location, see [Server locality using Razor and LLDP](#).

Creating custom facts for use with Razor is similar to creating custom facts for other Puppet uses. For more information about custom facts, see the [Facter documentation](#).

How the microkernel extension works

On both new and existing nodes, the microkernel retrieves and unpacks the latest extension file before each checkin.

The content of the extension file is placed in a new, non-persistent directory on the microkernel image. Changes to the directory aren't saved, and the directory is overwritten when a new extension file is available.

During unpacking, the executable bit on files is preserved. Permissions for the files in the extension are irrelevant, because the microkernel extension runs as root.

Microkernel extension configuration

The Razor microkernel extension is a ZIP file with the default title `mk-extension.zip`.

Depending on the requirements of the custom facts you're using, the extension can include these directories:

Directory	Contains	Environment variable modified
<code>bin</code>	executables	<code>PATH</code>
<code>lib</code>	shared libraries	<code>LD_LIBRARY_PATH</code>
<code>lib/ruby/facter/fact.rb</code>	Ruby code	<code>RUBYLIB</code>
<code>facts.d</code>	external facts	—

For example, from inside your microkernel extension directory:

```
$ zip -r ../mk-extension.zip
bin/
facts.d/
facts.d/foobar.yaml
```

Note: You can't change environment variables other than `PATH`, `LD_LIBRARY_PATH`, and `RUBYLIB`. For example, static `FACTER_<factname>` environment variables don't work with the microkernel extension.

Create the microkernel extension

Create a microkernel extension ZIP file to customize the facts that are available for provisioning decisions.

1. Create a ZIP file, as described in [Microkernel extension configuration](#), that contains the files required for your custom facts.
2. Place the ZIP file at `/etc/puppetlabs/razor-server` or, if you've changed the default, the location specified by the `microkernel_extension_zip` parameter of the `pe_razor` module.
3. Make sure the ZIP file is readable by the `pe-razor` user. For example, you can use `chmod 444 <FILE_NAME>` to make the file readable by all users.

Tips and limitations of the microkernel extension

Follow these guidelines to ensure that your microkernel extension behaves as expected.

- Use relative paths in your applications or facts, or search standard variables to locate content. Don't use absolute paths, because the content of the directory that the microkernel extension is unpacked to is variable.
- To store persistent state, for example if you want to save the result of a web service lookup to avoid calling the service at each checkin, don't use the location where the zip file is unpacked on the microkernel. Data stored at this location can be lost when the microkernel refreshes. Using `/tmp` is a valid choice, because the data persists in that location as long as the microkernel extension is running.

Working with Razor objects

Provisioning with Razor requires certain objects that define how nodes are provisioned.

Repositories

A repository is where you store all of the actual bits used by Razor to install a node. Or, in some cases, the external location of bits that you link to. A repo is identified by a unique name.

Instructions for the installation, such as what should be installed, where to get it, and how to configure it, are contained in tasks.

To load a repo onto the server, use the command:

```
razor create-repo --name=<repo name> --task <task name> --iso-url <URL>
```

For example:

```
razor create-repo --name centos-6.7 --task centos
--iso-url http://centos.sonn.com/6.7/isos/x86_64/CentOS-6.7-x86_64-
bin-DVD1.iso
```

There are three types of repositories that you might want to use, all created with the `create-repo` command:

- Repos where Razor downloads and unpacks ISOs for you and serves their contents.
- Repos that are external, such as a mirror that you maintain.
- Repos where a stub directory is created and you add the contents manually.

The `task` parameter is mandatory for creating all three of these types of repositories, and indicates the default installer to use with this repo. You can override a `task` parameter at the policy level. If you're not using a task, reference the stock task `noop`.

Unpack an ISO and serve its contents

This repository is created with the `--iso-url` property.

The server downloads and unpacks the ISO image onto its file system:

```
razor create-repo --name centos-6.7 --task centos
--iso-url http://centos.sonn.com/6.7/isos/x86_64/CentOS-6.7-x86_64-
bin-DVD1.iso
```

Point to an existing resource

To make a repository that points to an existing resource without loading anything onto the Razor server, provide a `url` property when you create the repository.

The `url` should be serving the unpacked contents of the install media.

```
razor create-repo --name centos-6.7 --task centos
--url http://mirror.example.org/centos/6.7/
```

Create a stub directory

For some install media, especially Windows install DVDs, Razor is not able to automatically unpack the media; this is a known limitation of the library that Razor uses to unpack ISO images.

In those cases, it is necessary to first use `create-repo` to set up a stub directory on the Razor server, then manually add content to it. The stub directory is created with:

```
razor create-repo --name win2012r2 --task windows/2012r2 \
--no-content true
```

When this command completes successfully, log into your Razor server as root and `cd` into your server's repository directory. The repository directory is specified by the `repo_store_root` class parameter of the `pe_razor` class. By default, the directory is `/opt/puppetlabs/server/data/razor_server/repo`.

```
# mount -o loop /path/to/windows_server_2012_r2.iso /mnt
# cp -pr /mnt/* win2012r2
# umount /mnt
```

Related information

[Create a repository](#) on page 646

Repositories contain – or point to – the operating system to install on a node.

[Repository commands](#) on page 626

Repository commands enable you to create and delete specified repositories from the Razor database and specify the task that installs the contents of a repository.

[Repositories API](#) on page 680

These commands enable you to create a new repository or delete a repository from the internal Razor database. You can also ensure that a specified repository uses a specified task.

Razor tasks

Razor tasks describe a process or collection of actions that are performed when Razor provisions machines. Tasks can be used to designate an operating system or other software to install, where to get it, and the configuration details for the installation.

Important: tasks differ from `tasks`, which let you run arbitrary scripts and commands using `.` See the orchestrator documentation for information about `tasks`.

Tasks consist of a YAML metadata file and any number of ERB templates. Templates are used to generate things like the iPXE script that boots a node into the installer, and automated installation files like kickstart, preseed, or unattended files.

You specify the tasks you want to run in *policies*.

You can use several sources for tasks:

- Supported tasks that are included with Razor
- Copies of supported tasks that you customize as needed
- Tasks that you create from scratch

Supported tasks

Razor includes these supported tasks.

Task	Version	Notes
CentOS	6, 7	
coreOS	1	Enables deployment of clusters.
Debian	wheezy	
microkernel		System task. Boots the Razor microkernel.
noop		System task. Boots a system locally.

Task	Version	Notes
Red Hat	6, 7	
SUSE Linux Enterprise Server	11, 12	
Ubuntu	16.04	
VMware ESXi	5.5, 6	
Windows	2008 R2, 2012 R2, 8 Pro, 2016	

Important: Don't modify supported tasks in place. If you want to customize a supported task, copy the task from the default task directory (`/opt/puppetlabs/server/apps/razor-server/share/razor-server/tasks`) to the custom task directory (`/etc/puppetlabs/razor-server/tasks`) before modifying the task.

Storage directories

Tasks are stored in the file system. The `task_path` class parameter of the `pe_razor` module determines where Razor looks for tasks.

The parameter can include a colon-separated list of paths. Relative paths in that list are taken to be relative to the top-level Razor directory. For example, setting `task_path` to `/opt/puppet/share/razor-server/tasks:/home/me/task:tasks` makes Razor search for tasks in these three directories in order.

By default, there are two directories that store tasks:

- `/opt/puppetlabs/server/apps/razor-server/share/razor-server/tasks` stores default tasks shipped with the product.
- `/etc/puppetlabs/razor-server/tasks` stores custom tasks.

Task metadata

Tasks can include the following metadata in the task's YAML file. This file is called `metadata.yaml` and exists in `tasks/<NAME>.task` where `NAME` is the task name. Therefore, the task name looks like this: `tasks/<NAME>.task/metadata.yaml`.

```
---
description: HUMAN READABLE DESCRIPTION
os: OS NAME
os_version: OS_VERSION_NUMBER
base: TASK_NAME
boot_sequence:
  1: boot_tmpl1
  2: boot_tmpl2
default: boot_local
```

Only `os_version` and `boot_sequence` are required. The `base` key allows you to derive one task from another by reusing some of the base metadata and templates. If the derived task has metadata that's different from the metadata in base, the derived metadata overrides the base task's metadata.

The `boot_sequence` hash indicates which templates to use when a node using this task boots. In the example above, a node first boots using `boot_tmpl1`, then using `boot_tmpl2`. For every subsequent boot, the node uses `boot_local`.

Task templates

Task templates are ERB templates and are searched in all the directories in the `task_path` configuration setting.

Templates are searched in the subdirectories in this order:

1. `name.task`
2. `base.task` if present

3. common

Template helpers

Templates can use the following helpers to generate URLs that point back to the server; all of the URLs respond to a GET request, even the ones that make changes on the server.

- `task`: Includes attributes such as `name`, `os`, `os_version`, `boot_seq`, `label`, `description`, `base`, and `architecture`.
- `node`: Includes attributes such as `name`, `metadata`, and `facts`.
Tip: You can use `node.hw_hash['fact_boot_type'] == "efi"` to evaluate whether a node booted via UEFI.
- `repo`: Includes attributes such as `name`, `iso_url`, `url`.
- `file_url(TEMPLATE, RAW)`: The URL that retrieves `TEMPLATE.erb` (after evaluation) from the current node's task. By default, the file is interpolated (`RAW=false`). If the file doesn't need to be interpolated, specify `RAW=true`.
- `repo_url(PATH)`: The URL to the file at `PATH` in the current repo.
- `repo_file_contents(PATH)`: The contents of the file at `PATH` inside the repo. This is an empty string if the file does not exist.
- `repo_file?(PATH)`: Whether a file exists at the given path. This is `nil` if the file does not exist.
- `log_url(MESSAGE, SEVERITY)`: The URL that logs `MESSAGE` in the current node's log.
- `node_url`: The URL for the current node.
- `store_url(VARS)`: The URL that stores the values in the hash `VARS` in the node. Currently only changing the node's IP address is supported. Use `store_url("ip" => "192.168.0.1")` for that.
- `stage_done_url`: The URL that tells the server that this stage of the boot sequence is finished, and that the next boot sequence should begin upon reboot.
- `broker_install_url`: A URL from which the install script for the node's broker can be retrieved. You can see an example in the script, [os_complete.erb](#), which is used by most tasks.

Each boot (except for the default boot) must culminate in something akin to `curl <%= stage_done_url %>` before the node reboots. Omitting this causes the node to reboot with the same boot template over and over again.

The task must indicate to the Razor server that it has successfully completed by doing a GET request against `stage_done_url("finished")`, for example using `curl` or `wget`. This marks the node installed in the Razor database.

You use these helpers by causing your script to perform an HTTP GET against the generated URL. This might mean that you pass an argument like `ks=<%= file_url("kickstart") %>` when booting a kernel, or that you put `curl <%= log_url("Things work great") %>` in a shell script.

Related information

[Task commands](#) on page 629

Use task commands to create tasks in the Razor database.

[Tasks API](#) on page 681

This commands enables you to create a task in the Razor database.

Tags

A tag consists of a unique name and a rule. Tags match a node if evaluating the node against the tag's facts results in `true`. Tag matching is case sensitive.

For example, to create a tag, *small*, that matches any machine with less than 4GB of memory:

```
razor create-tag --name small
  --rule '["<", ["num", ["fact", "memorysize_mb"]], 4128]'
```

Tag rules

Rule expressions are of the form `op arg1 arg2 ... argn` where `op` is one of the accepted [operators, and `arg1` through `argn` are the arguments for the operator. If the arguments are expressions themselves, they're evaluated before `op`.

Here are some example tag rules:

- To match nodes with more than 10 processors: `[">", ["num", ["fact", "processorcount"]], 10]`
- To match nodes with specified MAC addresses: `["has_macaddress", "de:ea:db:ee:f0:00", "de:ea:db:ee:f0:01"]`

Tag operators

The expression language supports these operators:

Operator	Returns	Aliases
<code>["=", arg1, arg2]</code>	True if the specified arguments are equal.	"eq"
<code>["!=" , arg1, arg2]</code>	True if the specified arguments are not equal.	"neq"
<code>["and", arg1, ..., argn]</code>	True if all arguments are true.	
<code>["or", arg1, ..., argn]</code>	True if any argument is true.	
<code>["not", arg]</code>	True if the argument evaluates to false or nil.	
<code>["fact", arg1 (, arg2)]</code>	The <code>arg1</code> fact for the node. The optional <code>arg2</code> is used if the <code>arg1</code> fact isn't present.	
<code>["metadata", arg1 (, arg2)]</code>	The <code>arg</code> metadata entry for the node. The optional <code>arg2</code> is used if the <code>arg1</code> fact isn't present.	
<code>["tag", arg]</code>	True if the node has the specified tag.	
<code>["has_macaddress", arg1, arg2 ..., argn]</code>	True if any facts that start with "macaddress" matches one of <code>arg1 ... argn</code> .	
<code>["has_macaddress_like", arg1, arg2 ..., argn]</code>	True if the hardware MAC address matches one of <code>arg1 ... argn</code> as regular expressions.	
<code>["in", arg1, arg2, ..., argn]</code>	True if <code>arg1</code> matches one of <code>arg2 ... argn</code> .	
<code>["num", arg1]</code>	<code>arg1</code> as a numeric value, or raises an error.	
<code>[">", arg1, arg2]</code>	True if <code>arg1</code> is greater than <code>arg2</code> .	"gt"
<code>["<", arg1, arg2]</code>	True if <code>arg1</code> is less than <code>arg2</code> .	"lt"
<code>[">=", arg1, arg2]</code>	True if <code>arg1</code> is greater than or equal to <code>arg2</code> .	"gte"
<code>["<=", arg1, arg2]</code>	True if <code>arg1</code> is less than or equal to <code>arg2</code> .	"lte"

Operator	Returns	Aliases
["like", arg1, arg2]	True if arg1 matches the pattern of arg2, interpreted as a regular expression.	
["lower", arg]	The lowercase version of the string arg.	
["upper", arg]	The uppercase version of the string arg.	

Related information

[\(Optional\) Create tags](#) on page 646

Tags let you group nodes based on their characteristics. You can then apply policies based on tags to install appropriate operating systems on tagged nodes. If you don't specify tags for a policy, the policy binds to any node.

[Tag commands](#) on page 631

Tag commands enable you to create new tags, set the rules used to apply the tag to nodes, change the rule of a specified tag, or delete a specified tag.

[Tags API](#) on page 682

These commands enable you to create a tag, delete a tag, or change the rule for a tag.

Policies

Policies tell Razor what bits to install, where to get the bits, how to configure them, and how the installed node can communicate with Puppet Enterprise.

Policies can contain tags, which match characteristics of nodes to specific policies. For example, you might create a policy that installs a certain operating system on nodes greater than 5GB in memory.

Policies without tags bind to any node. You might create policies without tags if:

- You have a single policy that installs the same operating system on all nodes.
- You want to install a specific number of various operating systems on a number of undifferentiated nodes. In this case, you can use the `max-count` attribute to specify how many of each operating system to provision.

For example, to create a policy, *centos-for-small*, that is applied to the first 20 nodes that match the `small` tag:

```
razor create-policy --name centos-for-small
--repo centos-6.7 --broker pe --tag small
--hostname 'host${id}.example.com'
--root-password secret --max-count 20
```

How policies bind to nodes

When a node boots into the Razor microkernel, it sends its facts to the Razor server. The node then binds to the first policy in the policy table that applies to the node. When a node binds to a policy, the node is provisioned with the operating system specified by the policy.

If the node doesn't match any policies, it continues to send facts periodically to the Razor server and remains unprovisioned until it does match a policy.

Policies don't bind to nodes if:

- The policy is disabled.
- The policy has already reached the maximum number of nodes that can bind to it.
- The policy requires tags that don't apply to the node.

Important: If you don't manage policies carefully, you can inadvertently enable Razor to match with and provision machines that you don't want to provision. In the case of existing servers, this can lead to catastrophic data loss.

The policy table

Policies are stored in a policy table. The order of the policy table is important because Razor applies the first policy that matches to a node.

You can influence the order of policies by:

- Using the `create-policy` command with `before` or `after` parameters to indicate where the new policy appears in the policy table.
- Using the `move-policy` command with `before` and `after` parameters to reorder existing policies.

Related information

[Create a policy](#) on page 648

Policies tell Razor what operating system to install on the provisioned node, where to get the OS software, how to configure it, and how to communicate between the node and Puppet Enterprise.

[Policy commands](#) on page 632

Policies govern how nodes are provisioned.

[Policies API](#) on page 683

Policies govern how nodes are provisioned depending on how they are tagged.

[Node metadata commands](#) on page 625

Node metadata commands enable you to update, modify, or remove metadata from nodes.

Brokers

Brokers hand off nodes to configuration management systems like Puppet Enterprise. Brokers consist of two parts: a broker type and information specific to the broker type.

Razor ships with three default broker types:

- `puppet-pe` — Hands off node management to Puppet Enterprise. This broker specifies the address of the Puppet server, the Puppet Enterprise version, and for Windows, the location of the Windows agent installer.
- `puppet` — Hands off management to open source Puppet. This broker specifies the address of the Puppet server, the node certname, and the environment.
- `noop` — Doesn't hand off management. A no-op broker can be useful for getting started quickly or doing a basic installation without configuration management.

You can create brokers using the `create-broker` command. You can also update configuration details for the broker and delete brokers.

Writing new broker types is necessary only when using Razor with other configuration management systems.

Broker storage directories

There are two directories that store brokers:

- `/opt/puppetlabs/server/apps/razor-server/share/razor-server/brokers` stores default brokers shipped with the product.
- `/etc/puppetlabs/razor-server/brokers` stores custom brokers that you create.

Tip: We recommend not modifying the directory or brokers at `/opt . . .`, but you can copy brokers from there to the custom broker directory and modify them as needed.

Creating a PE broker

To create a PE broker that enrolls nodes with the master at `puppet-master.example.com`:

```
razor create-broker --name=my_puppet --broker-type=puppet-pe \
  --configuration server=puppet.example.org \
  --configuration version=2015.3
```

Writing the broker install script

The broker install script is generated from the `install.erb(*nix)` or `install.ps1.erb` (Windows) template of your broker.

The template returns a valid shell script because tasks generally perform the handoff to the broker by running a command like `curl -s <%= broker_install_url %> | /bin/bash`. The GET request to `broker_install_url(*nix)` or `broker_install_url('install.ps1')` (Windows) returns the broker's install script after interpolating the template.

In the install template, you have access to two objects: `node` and `broker`.

The `node` object gives you access to node facts (`node.facts["example"]`), tags (`node.tags`), and metadata (`node.metadata['key']`).

The `broker` object gives you access to the configuration settings. For example, if your `configuration.yaml` specifies that a setting `version` must be provided when creating a broker from this broker type, you can access the value of `version` for the current broker as `broker.version`.

Writing the broker configuration file

The `configuration.yaml` file indicates what parameters can be supplied for any given broker type.

For each parameter, you can supply these attributes:

- `description` — Human-readable description of the parameter.
- `required` — `true` to indicate that the parameter must be supplied. Parameters that aren't required are optional.
- `default` — Value for the parameter if one isn't supplied.

As an example, here's the `configuration.yaml` for the PE broker type:

```
server:
  description: "The puppet master to load configurations and installation
  packages from."
version:
  description: "Override the PE version to install; defaults to
  `current`."
windows_agent_download_url:
  description: "The download URL for a Windows PE agent installer;
  defaults to a URL derived from the `version` config."
```

Create a new broker type

To create a broker called `sample`:

1. From the command line, create a `sample.broker` directory anywhere on the `broker_path`.

The `broker_path` is specified in the `broker_path` class parameter of the `pe_razor` class. By default, the broker directory for custom brokers is `/etc/puppetlabs/razor-server/brokers`.

2. Write the broker install script and place it in the `install.erb(*nix)` or `install.ps1.erb` (Windows PowerShell) template in the `sample.broker` directory.
3. If the broker type requires configuration data, write the broker configuration file and save it in the `sample.broker` directory.

Related information

[\(Optional\) Create a broker](#) on page 647

Brokers hand off nodes to configuration management systems like Puppet Enterprise.

[Broker commands](#) on page 639

Broker commands enable you to create a new broker configuration, set or clear a specified key value for a broker, and delete a specified broker.

[Brokers API](#) on page 686

A broker is responsible for configuring a newly installed node for a specific configuration management system. For Puppet Enterprise, you generally use brokers only with the type `puppet-pe`.

Hooks

Hooks are an optional but useful Razor object. Hooks provide a way to run arbitrary scripts when certain events occur during a node's lifecycle. The behavior and structure of a hook are defined by a *hook type*.

The two primary components for hooks are:

- **Configuration** — A JSON document for storing data on a hook. Configurations have an initial value and can be updated by hook scripts.
- **Event scripts** — Scripts that run when a specified event occurs. Event scripts must be named according to the handled event.

Hook storage directories

There are two directories that store hooks:

- `/opt/puppetlabs/server/apps/razor-server/share/razor-server/hooks` stores default hooks shipped with the product.
- `/etc/puppetlabs/razor-server/hooks` stores custom hooks.

Note: Don't modify the directory or hooks at `/opt . . .`, but you can copy hooks from there to the custom hook directory and modify them as needed.

File layout for a hook type

Similar to brokers and tasks, hook types are defined through a `.hook` directory and optional event scripts within that directory.

```
hooks/
  some.hook/
    configuration.yaml
    node-bind-policy
    node-unbind-policy
    ...
```

Available events

These are the events that you can create hooks for.

- `node-booted` — Triggered every time a node boots via iPXE.
- `node-registered` — Triggered after a node has been registered. Limited hardware information is available after registration.
- `node-deleted` — Triggered after a node has been deleted.
- `node-bound-to-policy` — Triggered after a node has been bound to a policy. The script input contains a `policy` property with the details of the policy that has been bound to the node.
- `node-unbound-from-policy` — Triggered after a node has been marked as uninstalled by the `reinstall-node` command and thus has been returned to the set of nodes available for installation.
- `node-facts-changed` — Triggered whenever a node changes its facts.
- `node-install-finished` — Triggered when a policy finishes its last step.

Creating hooks

The `create-hook` command is used to create a hook object from a hook type. For example:

```
razor create-hook --name myhook --hook-type some_hook
  --configuration example1=7 --configuration example2=rhubarb
```

The hook object created by this command sets its initial configuration to the JSON document:

```
{
  "example1": 7,
  "example2": "rhubarb"
}
```

Each time an event script for a hook runs, it has an opportunity to modify the hook's configuration. These changes to the configuration are preserved by the Razor server. The server also makes sure that hooks don't modify their configurations concurrently in order to avoid data corruption.

The `delete-hook` command is used to remove a hook.

Hook configuration

Hook scripts can use the hook object's configuration.

The hook type specifies the configuration data that it accepts in `configuration.yaml`. That file must define a hash:

```
example1:
  description: "Explain what example1 is for"
  default: 0
example2:
  description "Explain what example2 is for"
  default: "Barbara"
...
```

For each event that the hook type handles, it must contain a script with the event's name. That script must be executable by the Razor server. All hook scripts for a certain event are run in an indeterminate order when that event occurs.

Event scripts

The general protocol is that hook event scripts receive a JSON object on their `stdin`, and might return a result by printing a JSON object to their `stdout`.

The properties of the input object vary by event, but they always contain a `hook` property:

```
{
  "hook": {
    "name": hook name,
    "configuration": ... operations to perform ...
  }
}
```

The `configuration` object is initialized from the hash described in the hook's `configuration.yaml` and the properties set by the current values of the hook object's configuration. With the `create-hook` command above, the input JSON would be:

```
{
  "hook": {
    "name": "myhook",
    "configuration": {
      "update": {
        "example1": 7,
        "example2": "rhubarb"
      }
    }
  }
}
```

The script might return data by producing a JSON object on its `stdout` to indicate changes that need to be made to the hook's configuration. The updated configuration is used on subsequent invocations of any event for that hook. The output must indicate which properties to update, and which ones to remove:

```
{
  "hook": {
    "configuration": {
      "update": {
        "example1": 8
      },
      "remove": [ "frob" ]
    }
  }
}
```

The Razor server ensures that invocations of hook scripts are serialized. For any hook, events are processed one-by-one to allow for transactional safety around the changes any event script might make.

Node events

Most events are directly related to a node. The JSON input to the event script has a `node` property that contains the representation of the node in the same format the API produces for node details.

The JSON output of the event script can modify the node metadata:

```
{
  "node": {
    "metadata": {
      "update": {
        "example1": 8
      },
      "remove": [ "frob" ]
    }
  }
}
```

Error handling

The hook script must exit with exit code 0 if it succeeds; any other exit code is considered a failure of the script.

Whether the failure of a script has any other effects depends on the event. A failed execution can still make updates to the hook and node objects by printing to `stdout` in the same way as a successful execution.

To report error details, the script produces a JSON object with an `error` property on its `stdout` in addition to exiting with a non-zero exit code. If the script exits with exit code 0, the `error` property is still recorded, but the event's severity isn't an error. The `error` property itself contains an object whose `message` property is a human-readable message; additional properties can be set. For example:

```
{
  "error": {
    "message": "connection refused by frobnicate.example.com",
    "port": 2345,
    ...
  }
}
```

Sample input

The input to the hook script is in JSON, containing a structure like this:

```
{
  "hook": {
```

```

    "name": "counter",
    "configuration": { "value": 0 }
  },
  "node": {
    "name": "node10",
    "hw_info": {
      "mac": [ "52-54-00-30-8e-45" ],
      ...
    },
    "dhcp_mac": "52-54-00-30-8e-45",
    "tags": [ "compute", "anything", "any", "new" ],
    "facts": {
      "memorysize_mb": "995.05",
      "facterversion": "2.0.1",
      "architecture": "x86_64",
      ...
    },
    "state": {
      "installed": false
      "physicalprocessorcount": "1",
    },
    "hostname": "client-1.watzmann.net",
    "root_password": "secret",
    "netmask_eth0": "255.255.255.0",
    "ipaddress_lo": "127.0.0.1",
    "last_checkin": "2014-05-21T03:45:47+02:00"
  },
  "policy": {
    "name": "client-1",
    "repo": "centos-6.7",
    "task": "ubuntu",
    "broker": "noop",
    "enabled": true,
    "hostname_pattern": "client-1.watzmann.net",
    "root_password": "secret",
    "tags": [ "client-1" ],
    "nodes": { "count": 0 }
  }
}

```

Sample hook

Here is an example of a basic hook called `counter` that counts the number of times Razor registers a node.

This example creates a corresponding directory for the hook type, `counter.hook`, inside the `hooks` directory.

You can store the current count as a configuration entry with the key `count`. Thus the `configuration.yaml` file might look like this:

```

count:
  description: "The current value of the counter"
  default: 0

```

To make sure a script runs whenever a node is bound to a policy, create a file called `node-bound-to-policy` and place it in the `counter.hook` folder. Then write this script, which reads in the current configuration value, increments it, then returns some JSON to update the configuration on the hook object:

```

#!/bin/bash

json=$(< /dev/stdin)

name=$(jq '.hook.name' <<< $json)

```

```

value=$(( $(jq '.hook.config.count' <<< $json) + 1 ))

cat <<EOF
{
  "hook": {
    "configuration": {
      "update": {
        "count": $value
      }
    }
  },
  "node": {
    "metadata": {
      $name: $value
    }
  }
}
EOF

```

Note that this script uses `jq`, a bash JSON manipulation framework. This must be on the `$PATH` in order for execution to succeed.

Next, create the hook object, which stores the configuration:

```
razor create-hook --name counter --hook-type counter
```

Because the configuration is absent from this creation call, the default value of 0 in `configuration.yaml` is used. Alternatively, this could be set using `--configuration count=0` or `--c count=0`.

The hook is now ready to use. You can query the existing hooks in a system via `razor hooks`. To query the current value of the hook's configuration, `razor hooks counter` shows `count` initially set to 0. When a node gets bound to a policy, the `node-bound-to-policy` script is triggered, yielding a new configuration value of 1.

Assign dynamic hostnames using hooks

You can use a hook to create more advanced dynamic hostnames than the simple incremented pattern — `$_{id}.example.com` — from the `hostname` property on a policy.

Before you begin

Ruby must be installed in `$PATH` for the hook script to succeed. By default, Ruby is installed as required with Puppet Enterprise. If Ruby isn't installed, add it to one of the paths specified in the `hook_execution_path` class parameter of the `pe_razor` class.

This type of hook calculates the correct hostname and returns that hostname as metadata on the node. To do so, it uses a basic counter system that stores the number of nodes bound to a given policy.

This hook is intended to be extended for cases where an external system needs to be contacted to determine the correct hostname. In such a scenario, the new value is still returned as metadata for the node.

1. Create an instance of a default hook:

```

razor create-hook --name some_policy_hook --hook-type hostname \
  --configuration policy=some_policy \
  --configuration hostname-pattern='$_{policy}$_{count}.example.com'

```

2. (Optional) If multiple policies require their own counter, create multiple instances of this hook with different `policy` or `hostname-pattern` hook configurations.

Running the `create-hook` command kicks off this sequence of events:

1. The counter for the policy starts at 1.
2. When a node boots, the `node-bound-to-policy` event is triggered.
3. The policy's name from the event is then passed to the hook as input.

4. The hook matches the node's policy name to the hook's policy name.
5. If the policy matches, the hook calculates a rendered `hostname-pattern`:
 - It replaces `${count}` with the current value of the `counter` hook configuration.
 - It left-pads the `${count}` with padding zeroes. For example, if the hook configuration's padding equals 3, a count of 7 is rendered as 007.
 - It replaces `${policy}` with the name of its policy.
6. The hook returns the rendered `hostname-pattern` as the node metadata of `hostname` and returns the incremented value for the counter that was used, so that the next execution of the hook uses the next value.

Viewing the hook's activity log

To view the status of the hook's executions, see `razor hooks $name log`:

```
timestamp: 2015-04-01T00:00:00-07:00
policy: policy_name
cause: node-bound-to-policy
exit_status: 0
severity: info
actions: updating hook configuration: {"update"=>{"counter"=>2}} and
updating node metadata:
{"update"=>{"hostname"=>"policy_name1.example.com"}}
```

Related information

[Hook commands](#) on page 641

Hooks are custom, executable scripts that are triggered to run when a node hits certain phases in its lifecycle. A hook script receives several properties as input, and can make changes in the node's metadata or the hook's internal configuration.

[Hooks API](#) on page 687

Hooks are custom, executable scripts that are triggered to run when a node hits certain phases in its lifecycle. A hook script receives several properties as input, and can make changes in the node's metadata or the hook's internal configuration.

Keeping Razor scalable

Speed provisioning and reduce load on the Razor server by following these scalability best practices.

- When [creating repos](#), use `url` rather than `iso-url`. If you have a mirror hosting installation files, they can be distributed straight to the node rather than having the Razor server distribute the files.
- When [matching policies](#), limit the number of tags you use.
 - Create tags using the `has_macaddress` operator to identify a list of nodes to be assigned the tag, for example, `["has_macaddress", "de:ea:db:ee:f0:00", "de:ea:db:ee:f0:01"]`.
 - Set the `policy` metadata key on nodes using the `modify-node-metadata` command, then create matching tags, for example, `["=", ["metadata", "policy"], "policy1"]`.
- When [running hooks](#), avoid long- and frequent-running scripts, which all run on the same thread synchronously.

Using the Razor API

The Razor API is REST-based.

By default, API calls are sent over HTTPS with TLS/SSL. The default URL for the API endpoint is `https://localhost:8151/api`. This URL varies if you're running the Razor client on a different machine than the server, or if you changed the default port due to a port conflict.

The API uses JSON exclusively, and all requests must set the HTTP `Content-Type` header to `application/json` and must `Accept: application/json` in the response.

The Razor API is stable, and clients can expect operations that work against this version of the API to work against future versions of the API. However, we might add information or functionality to the API, so clients must ignore anything in responses they receive from the server that they don't understand.

Structure and keys

Everything underneath `/api` is part of the public API and is stable.

Note: The `/svc` namespace is an internal namespace used for communication with the iPXE client, the microkernel, and other internal components of Razor. This namespace is not enumerated under `/api` and has no stability guarantee. We recommend making `/svc` URLs available only to that part of the network that contains nodes that need to be provisioned.

The top-level `/api` endpoint serves as the start for navigating through the Razor command and query facilities. For example, given the default API URL, call `GET https://razor:8151/api` to view the API.

The response is a JSON object with the following keys:

- `commands` — The commands available on this server.
- `collections` — Read-only queries available on this server.
- `version` — The version of Razor that is running on the server. The version should be used only for diagnostic purposes and for bug reporting, and never to decide whether the server supports a certain operation or not.

Each of those keys contains a JSON array, with a sequence of JSON objects that have the following keys:

- `name` — A human-readable label for an object, usually unique only among objects of the same type on the same server.
- `id` — The URL of an entity. A `GET` request against a URL with an `id` attribute produces a representation of the object.
- `rel` — A "spec URL" that indicates the type of data contained. Use this key to discover the endpoint that you want to follow, rather than using the name.

For example, `{"name": "add-policy-tag", "rel": "https://api.puppetlabs.com/razor/v1/commands/add-policy-tag", "id": "https://localhost:8151/api/commands/add-policy-tag"}`.

Commands

The list of commands that the Razor server supports is returned as part of a request to `GET /api` in the `commands` array. Clients can identify commands using the `rel` attribute of each entry in the array, and make their `POST` requests to the URL given in the `id` attribute.

The `id` URL for each command supports the following HTTP methods:

- `GET` — Retrieve information about the command, such as a help text and machine-readable information about the parameters this command takes.
- `POST` — Execute the command. Command parameters are supplied in a JSON document in the body of the `POST` request.

Commands are generally asynchronous and return a status code of 202 `Accepted` on success. The response from a command generally has this form:

```
{
  "result": "Policy win2012r2 disabled",
  "command": "http://razor:8088/api/collections/commands/74"
}
```

Here, `result` is a human-readable explanation of what the command did, and `command` points into the collection of all the commands that were ever run against this server. Performing a `GET` against the `command` URL provides additional information about the execution of this command, such as the status of the command, the parameters sent to the server, and details about errors.

Tip: Most client commands allow positional arguments, which can save keystrokes.

Related information

[Using positional arguments with Razor client commands](#) on page 618

Most Razor client commands allow positional arguments, which means that you don't have to explicitly enter the name of the argument, like `--name`. Instead, you can provide the values for each argument in a specific order.

Collections

In addition to the supported commands above, a `GET /api` request returns a list of supported collections in the `collections` array.

Each entry contains at minimum the following keys:

- `name` — A human-readable name for the collection.
- `id` — The endpoint through which the collection can be retrieved (via `GET`).
- `rel` — The type of the collection.

A `GET` request to the `id` of a collection returns a JSON object. The `spec` property of that object indicates the type of collection. The `total` indicates how many items there are in the collection in total (not just how many were returned by the query). The `items` value is the actual list of items in the collection, a JSON array of objects. Each object has these properties:

- `id` — A URL that uniquely identifies the object. A `GET` request to this URL provides further detail about the object.
- `spec` — A URL that identifies the type of the object.
- `name` — A human-readable name for the object.

Object details

Performing a `GET` request against the `id` of an item in a collection returns further detail about that object. Different types of objects provide different properties.

For example, here is a sample tag listing:

```
{
  "spec": "http://api.puppetlabs.com/razor/v1/collections/tags/member",
  "id": "https://razor:8151/api/collections/tags/anything",
  "name": "anything",
  "rule": [ "=", 1, 1 ],
  "nodes": {
    "id": "http://razor:8151/api/collections/tags/anything/nodes",
    "count": 2,
    "name": "nodes"
  },
  "policies": {
    "id": "http://razor:8151/api/collections/tags/anything/policies",
    "count": 0,
    "name": "policies"
  }
}
```

References to other resources are represented as a single JSON object (in the case of a one-to-one relationship) or an array of JSON objects (for a one-to-many or many-to-many relationship). Each JSON object contains these fields:

- `id` — A URL that uniquely identifies the associated object or collection of objects.
- `spec` — The type of the associated object.
- `name` — A human-readable name for the object.
- `count` — The number of objects in the associated collection.

Querying the node collection

You can query nodes based on `hostname` or fields stored in `hw_info`.

- `hostname` — A regular expression to match against hostnames. The results include partial matches, so `hostname=example` returns all nodes whose hostnames include `example`.
- fields stored in `hw_info` — `mac`, `serial`, `asset`, and `uuid`.

For example, the following queries the UUID to return the associated node:

```
/api/collections/nodes?uuid=9ad1e079-b9e3-347c-8b13-9b42cbf53a14'
```

```
{
  "items": [
    {
      "id": "https://razor.example.com:8151/api/collections/nodes/node14",
      "name": "node14",
      "spec": "http://api.puppetlabs.com/razor/v1/collections/nodes/member"
    }
  ],
  "spec": "http://api.puppetlabs.com/razor/v1/collections/nodes"
}
```

Paging collections

The nodes and events collections are paginated.

GET requests for them can include the following parameters to limit the number of items returned:

- `limit` — Return this many items.
- `start` — Return items starting at `start`.

Razor API reference

Use the Razor API to interact with Razor and provision bare metal nodes

Related information

[API index](#) on page 33

APIs allow you to interact with Puppet and Puppet Enterprise (PE) applications from your own code or application integration hooks.

The default bootstrap iPXE file

A GET request to `/api/microkernel/bootstrap` returns the iPXE script that you should put on your TFTP server (usually called `bootstrap.ipxe`). The script gathers information about the node (the `hw_info`) that it sends to the server and that the server uses to identify the node and determine how exactly the node should boot.

The URL accepts the parameter `nic_max`, which you should set to the maximum number of network interfaces that respond to DHCP on any given node. It defaults to 4.

The URL also accepts an `http_port` parameter, which tells Razor which port its internal HTTP communications should use, and the `/svc` URLs must be available through that port. The default install uses 8150 for this.

Configuration API

Razor configuration is pulled from a configuration file controlled by Puppet Enterprise. You can change configuration values in the console with class parameters of the `pe_razor` class.

Note: In order for configuration changes to take effect, you must restart the Razor service by running `service pe-razor-server restart` on the Razor server.

View configuration (`config`)

The `config` endpoint displays details about your Razor configuration.

Note: Properties specified in the `api_config_blacklist` aren't returned by the `config` endpoint.

The endpoint handles a GET request to the collection URL specified in `/api`.

Nodes API

These commands enable you to register a node, set a node's hardware information, remove a single node, or remove a node's associate with any policies and clear its installed flag.

Register a node (`register-node`)

Register a node with Razor before it is discovered, and potentially provisioned.

In environments in which some nodes have been provisioned outside of the purview of Razor, this command offers a way to tell Razor that a node is valuable and not eligible for automatic reprovisioning. Such nodes are reprovisioned only if they are marked available with the `reinstall-node` command.

The `register-node` command allows you to perform the same registration that would happen when a new node checks in, but ahead of time. The `register-node` command uses the `installed` value to indicate that a node has already been installed, which signals to Razor to ignore the node and act as if it had successfully installed that node.

In order for this command to be effective, `hw_info` must contain enough information that the node can successfully be identified based on the hardware information sent from iPXE when the node boots; this usually includes the MAC addresses of all network interfaces of the node.

`register-nodes` accepts the following parameters:

- `hw_info` — Required, provides the hardware information for the node. This is used to match the node on first boot with the record in the database. The `hw_info` can contain all or a subset of the following entries:
 - `netN` — The MAC addresses of each network interface, for example `net0` or `net2`: The order of the MAC addresses is not significant.
 - `serial` — The DMI serial number of the node.
 - `asset` — The DMI asset tag of the node.
 - `uuid` — DMI UUID of the node.
- `installed` — A Boolean flag indicating whether this node is considered installed and therefore not eligible for reprovisioning by Razor

API example

Registering a machine before booting it with `installed` set to `true` protects it from accidental reinstallation by Razor:

```
{
  "hw_info": {
    "net0": "78:31:c1:be:c8:00",
    "net1": "72:00:01:f2:13:f0",
    "net2": "72:00:01:f2:13:f1"
  },
  "installed": true
}
```

Set node hardware info (`set-node-hw-info`)

When a node's hardware changes, such as a network card being replaced, the Razor server needs to be informed so it can correctly match the new hardware to the existing node definition.

The `set-node-hw-info` command lets you replace the existing hardware data with new data, prior to booting the modified node on the network. For example, update `node172` with new hardware information as follows:

```
{
  "node": "node172",
  "hw_info": {
    "net0": "78:31:c1:be:c8:00",
  }
}
```

```

        "net1": "72:00:01:f2:13:f0",
        "net2": "72:00:01:f2:13:f1"
    }
}

```

The format of the `hw_info` is the same as for the `register-node` command.

Delete node (`delete-node`)

To remove a single node, provide its name:

```

{
  "name": "node17"
}

```

Note: If the deleted node boots again at some point, Razor automatically recreates it.

Reinstall node (`reinstall-node`)

To remove a node's association with any policy and clear its `installed` flag, provide its name:

```

{
  "name": "node17"
}

```

After the node reboots, it boots back into the microkernel, goes through discovery and tag matching, and can bind to another policy for reinstallation. This command does not change the node's metadata or facts.

IPMI API

IPMI commands are node commands based on the Intelligent Platform Management Interface.

Note: You must install the `ipmitool` on the Razor server before using IPMI commands. To install the tool, run `yum install ipmitool -y`.

Set node IPMI credentials (`set-node-ipmi-credentials`)

Razor can store IPMI credentials on a per-node basis. These credentials include a hostname (or IP address), username, and password to use when contacting the BMC/LOM/IPMI LAN or LANplus service to check or update power state and other node data.

After IPMI credentials have been set up for a node, you can use the `reboot-node` and `set-node-desired-power-state` commands.

These three data items must be set or reset together, in a single operation. When you omit a parameter, Razor sets it to NULL (representing no value, or the NULL username/password as defined by IPMI).

The structure of a request is:

```

{
  "name": "node17",
  "ipmi_hostname": "bmc17.example.com",
  "ipmi_username": null,
  "ipmi_password": "sekretskwirrl"
}

```

This command works only with remote IPMI targets, not locally; therefore, you *must* provide an IPMI hostname.

Reboot node (`reboot-node`)

If you've associated IPMI credentials with a node, Razor can use IPMI to trigger a hard power cycle.

Just provide the name of the node:

```
{
  "name": "node1",
}
```

The IPMI communication spec includes some generous internal rate limits to prevent it from overwhelming the network or host server. If an execution slot isn't available on the target node, your `reboot-node` command goes into a background queue, and runs as soon as a slot is available.

This background queue is cumulative and persistent: there are no limits on how many commands you can queue up, how frequently a node can be rebooted, or how long a command can stay in the queue. If you restart your Razor server before the queued commands are executed, they remain in the queue and run after the server restarts.

The `reboot node` command is not integrated with IPMI power state monitoring, so you can't see power transitions in the record or when polling the node object.

Set a node's desired power state (`set-node-desired-power-state`)

By default, Razor checks your nodes' power states every few minutes in the background. If it detects a node in a non-desired state, Razor issues an IPMI command directing the node to its desired state.

To set the desired state for a node:

```
{
  "name": "node1234",
  "to":    "on" | "off" | null
}
```

The `name` parameter identifies the node to change the setting on. The `to` parameter contains the desired power state to set. Valid values are `on`, `off`, or `null` (the JSON NULL/nil value), which reflect "power on", "power off", and "do not enforce power state" respectively.

Node metadata API

These commands enable you to add, update, or remove metadata keys and remove metadata entries.

Modify node metadata (`modify-node-metadata`)

Node metadata is a collection of key/value pairs, much like a node's facts. The difference is that the facts represent what the node tells Razor about itself, while its metadata represents what you tell Razor about the node.

The `modify-node-metadata` command lets you add, update, or remove individual metadata keys, or clear a node's metadata:

```
{
  "node": "node1",
  "update": {
    "key1": "value1",
    "key2": "value2",
    ...
  }
  "remove": [ "key3", "key4", ... ],
  "no_replace": true
}
```

Add or update these keys

Remove these keys

Do not replace keys on update. Only add new keys

or

```
{
  "node": "node1",
  "clear": true
}
```

Clear all metadata

```
}
```

You can submit multiple updates or removals in a single command. However, `clear` only works on its own.

Tip: In batch updates with `no_replace`, use `force` to bypass errors. Existing keys aren't modified.

Update node metadata (`update-node-metadata`)

The `update-node-metadata` command offers a simplified way to update a single metadata key.

The body for the command must be:

```
{
  "node"      : "model",
  "key"       : "my_key",
  "value"     : "my_val",
  "no_replace": true
}
```

The `no_replace` parameter is optional. If it is `true`, the metadata entry is not modified if it already exists.

Remove node metadata (`remove-node-metadata`)

The `remove-node-metadata` command offers a simplified way to remove either a single metadata entry or all metadata entries on a node:

```
{
  "node" : "node1",
  "key"  : "my_key",
}
```

or

```
{
  "node" : "node1",
  "all"  : true,      # Removes all keys
}
```

Repositories API

These commands enable you to create a new repository or delete a repository from the internal Razor database. You can also ensure that a specified repository uses a specified task.

Create new repository (`create-repo`)

The `create-repo` command creates a new repository. The repository can contain the content to install a node, or it can point to an existing online repository.

You can create three types of repositories:

- Those that reference content available on another server, for example, on a mirror you maintain (`url`).
- Those where `unpacks` ISOs for you and serves their contents (`iso_url`).
- Those where `creates` a stub directory that you can manually fill with content (`no_content`).

The `task` parameter is mandatory in all three variants of this command, and indicates the default task that should be used when installing machines using this repository. The `task` parameter can be overridden at the policy level. If you're not using a task, reference the stock task `noop`.

To have Razor unpack an ISO for you and serve its content:

```
{
  "name": "fedora19",
}
```



```

    "iso_url": "file:///tmp/Fedora-19-x86_64-DVD.iso"
    "task": "puppet"
  }

```

Tip: Supplying the `iso_url` property when you create a repository ensures that you can delete it from the server with the `delete-repo` command.

To create a repository that points to an existing resource without loading anything onto the Razor server:

```

{
  "name": "fedora19",
  "url": "http://mirrors.n-ix.net/fedora/linux/releases/19/Fedora/x86_64/os/"
  "task": "noop"
}

```

To create an empty directory that you can manually fill later:

```

{
  "name": "win2012r2",
  "no_content": true
  "task": "noop"
}

```

After creating a `no_content` repository, you can log into your Razor server and fill the repository directory with the content, for example by loopback-mounting the install media and copying it into the directory. The repository directory is specified by the `repo_store_root` class parameter of the `pe_razor` class. By default, the directory is in `/opt/puppetlabs/server/data/razor_server/repo`.

Using `no_content` is usually required for Windows install media, because the library that Razor uses to unpack ISO images can't handle Windows ISO images.

Delete a repository (`delete-repo`)

The `delete-repo` command deletes a repository from the internal Razor database.

The command accepts a single repository name:

```

{
  "name": "fedora16"
}

```

This command deletes the repository from the internal Razor database. If you supplied the `iso_url` property when you created the repository, the folder is also deleted from the server. If you didn't supply the `iso_url` property, content remains in the repository directory.

Update a repository's specified task (`update-repo-task`)

Ensures that a specified repository uses the task this command specifies, setting the task if necessary. If a node is currently provisioning against the repo when you run this command, provisioning might fail.

```

{
  "repo": "my_repo",
  "task": "other_task"
}

```

Tasks API

This commands enables you to create a task in the Razor database.

Important: `tasks` differ from `tasks`, which let you run arbitrary scripts and commands using `.` See the orchestrator documentation for information about `tasks`.

Create task (`create-task`)

The `create-task` command creates a task in the Razor database. This command is an alternative to manually placing task files in the `task_path`. If you anticipate needing to make changes to tasks, we recommend the disk-backed task approach.

The body of the POST request for this command has the following form:

```
{
  "name": "redhat6",
  "os": "Red Hat Enterprise Linux",
  "boot_seq": {
    "1": "boot_install",
    "default": "boot_local"
  },
  "templates": {
    "boot_install": "... ERB template for an ipxe boot file ...",
    "installer": "... another ERB template ..."
  }
}
```

The possible properties in the request are:

- `name` — The name of the task; must be unique.
- `os` — The name of the OS; mandatory.
- `description` — Human-readable description.
- `boot_seq` — A hash mapping the boot counter or 'default' to an ERB template.
- `templates` — A hash mapping template names to the actual ERB template text.

Tags API

These commands enable you to create a tag, delete a tag, or change the rule for a tag.

Create tag (`create-tag`)

To create a tag, use the following in the body of your POST request:

```
{
  "name": "small",
  "rule": ["=", ["fact", "processorcount"], "2"]
}
```

The name of the tag must be unique; the rule is a match expression.

Delete tag (`delete-tag`)

To delete a tag, use the following in the body of your POST request:

```
{
  "name": "small",
  "force": true
}
```

You can't delete a tag while it's being used by a policy, unless you set the optional `force` parameter to `true`. In that case, Razor removes the tag from all policies using it and then deletes it.

Update tag (`update-tag-rule`)

To change the rule for a tag, use the following in the body of your POST request:

```
{
  "name": "small",
```

```

    "rule": [ "<=", [ "fact", "processorcount" ], "2" ],
    "force": true
}

```

This changes the rule of the given tag to the new rule. Razor then reevaluates the tag against all nodes and updates each node's tag attribute to reflect whether the tag now matches or not.

If the tag is used by any policies, the update is performed only if you set the optional `force` parameter to `true`. Otherwise, the command returns status code 400.

Policies API

Policies govern how nodes are provisioned depending on how they are tagged.

Razor maintains an ordered table of policies. When a node boots, Razor traverses this table to find the first eligible policy for that node. A policy might be ineligible for binding to a node if the node does not contain all of the tags on the policy, if the policy is disabled, or if the policy has reached its maximum for the number of allowed nodes.

When you list the `policies` collection, the list is in the order in which Razor checks policies against nodes.

Create policy (`create-policy`)

```

{
  "name": "a policy",
  "repo": "some_repo",
  "task": "redhat6",
  "broker": "puppet",
  "hostname": "host${id}.example.com",
  "root_password": "secret",
  "max_count": 20,
  "before|"after": "other policy",
  "node_metadata": { "key1": "value1", "key2": "value2" },
  "tags": [ "existing_tag", "another_tag" ]
}

```

Note: Because the policy contains many fields, you might want to put it in a JSON file. If you do, then your `create-policy` command would include the file name, like this: `razor create-policy --json <name of policy file>.json`.

Tags, repos, tasks, and brokers are referenced by name. The `tags` are optional. A policy with no tags can be applied to a node. The `task` is also optional. If it is omitted, the `repo`'s task is used. The `repo` and `broker` entries are required.

The `hostname` parameter defines a simple pattern for the hostnames of nodes bound to your policy. The `${id}` references each node's DB id.

The `max_count` parameter sets an upper limit on how many nodes can be bound to your policy at a time. You can specify a positive integer, or make it unlimited by setting it to `nil`.

Razor considers each policy sequentially, based on its order in a table. By default, new policies go at the end of the table. To override the default order, include a `before` or `after` argument referencing an existing policy by name.

The `node_metadata` parameter lets your policy apply metadata to a node when it binds. This does not overwrite existing metadata; it only adds keys that are missing. To install Windows on non-English systems, specify the `win_language` using the culture code for the appropriate [Microsoft language pack](#)). For example, `--node-metadata win_language=es-ES`.

Move policy (`move-policy`)

This command lets you change the order in which Razor considers your policies for matching against nodes.

To move an existing policy into a different place in the order, use the `move-policy` command with a body like:

```

{

```

```

    "name": "a policy",
    "before"|"after": "other policy"
  }

```

This changes the policy table so that "a policy" appears before or after "other policy".

Enable/disable policy (**enable-policy/disable-policy**)

To keep a policy from being matched against any nodes, disable it with the `disable-policy` command.

To enable a disabled policy, use the `enable-policy` command. Both commands use a body like:

```

{
  "name": "a policy"
}

```

Modify the `max_count` for a policy (**modify-policy-max-count**)

The command `modify-policy-max-count` lets you set the maximum number of nodes that can be bound to a specific policy.

Form the body of the request like this:

```

{
  "name": "a policy"
  "max_count": new-count
}

```

`new-count` can be an integer, which must be greater than the number of nodes that are currently bound to the policy. Alternatively, the `no_max_count` argument makes the policy unbounded:

```

{
  "name": "a policy"
  "no_max_count": true
}

```

Add tags to policy (**add-policy-tag**)

To add tags to a policy, supply the name of a policy and of the tag:

```

{
  "name": "a-policy-name",
  "tag" : "a-tag-name",
}

```

To create the tag in addition to adding it to the policy, supply the `rule` argument:

```

{
  "name": "a-policy-name",
  "tag" : "a-new-tag-name",
  "rule": "new-match-expression"
}

```

Remove tags from policy (**remove-policy-tag**)

To remove tags from a policy, supply the name of a policy and the name of the tag.

```

{
  "name": "a-policy-name",
  "tag" : "a-tag-name",
}

```

```
}
```

A policy with no tags can still be applied to any node.

Update a policy's specified repository (`update-policy-repo`)

Ensures that a policy uses the repository this command specifies. If necessary, `update-policy-repo` sets the repository, for example if a policy has already been created and you want to add a repository to it.

The following shows how to update a policy's repository to a repository called "fedora21":

```
{
  "node": "node1",
  "policy": "my_policy",
  "repo": "fedora21"
}
```

Update a policy's specified task (`update-policy-task`)

Ensures that a policy uses the task this command specifies. If necessary, `update-policy-task` sets the task, for example if a policy has already been created and you want to add a task to it.

If a node is currently provisioning against the policy when you run this command, provisioning can fail.

The following shows how to update a policy's task to a task called "other_task".

```
{
  "node": "node1",
  "policy": "my_policy",
  "task": "other_task"
}
```

Update a policy's specified broker (`update-policy-broker`)

Ensures that a policy uses the broker this command specifies. If necessary, `update-policy-broker` sets the broker, for example if a policy has already been created and you want to add a broker to it.

If a node is currently provisioning against the policy when you run this command, provisioning can fail.

The following shows how to update a policy's broker to a broker called "legacy-puppet":

```
{
  "node": "node1",
  "policy": "my_policy",
  "broker": "legacy-puppet"
}
```

Update a policy's node metadata (`update-policy-node-metadata`)

Ensures that a policy uses the node metadata this command specifies. If necessary, `update-policy-node-metadata` sets the node metadata, for example if a policy has already been created and you want to add node metadata to it.

The following shows how to update a policy's node metadata for the "my_key" value:

```
{
  "node": "node1",
  "policy": "my_policy",
  "key": "my_key",
  "value": "my_value"
}
```

Delete policy (delete-policy)

To delete a policy, supply the name of a single policy:

```
{
  "name": "my-policy"
}
```

Note that this does not affect the installed status of a node, and therefore can't, by itself, make a node bind to another policy upon reboot.

Brokers API

A broker is responsible for configuring a newly installed node for a specific configuration management system. For Puppet Enterprise, you generally use brokers only with the type `puppet-pe`.

Create broker (create-broker)

To create a broker, post the following to the `create-broker` URL:

```
{
  "name": "puppet",
  "configuration": {
    "server": "puppet.example.org"
  },
  "broker-type": "puppet-pe"
}
```

The `broker-type` must correspond to a broker that is present in a valid broker directory. Broker directories are specified in the `broker_path` class parameter of the `pe_razor` class. By default, the broker path for custom brokers is `/etc/puppetlabs/razor-server/brokers:brokers`.

The permissible settings for the `configuration` hash depend on the broker type and are declared in the broker type's `configuration.yaml`. For the `puppet-pe` broker type, these are:

- `server` — The hostname of the master.
- `version` — The agent version to install; this defaults to `current` and must be used only in exceptional circumstances.
- `ntpdate_server` — URL for an NTP server, such as `us.pool.ntp.org`, used to synchronize the date and time before installing the agent.

Update broker configuration (update-broker-configuration)

To set or clear a specified key value for a broker, use `update-broker-configuration`.

This argument changes the key `"some_key"` to `"new_value"`:

```
{
  "broker": "mybroker",
  "key": "some_key",
  "value": "new_value"
}
```

This argument clears the value for `"some_key"`:

```
{
  "broker": "mybroker",
  "key": "some_key",
  "clear": true
}
```

The `update-broker-configuration` command can be useful to update the Puppet Enterprise version of a `puppet-pe` broker. For example, if you created a `puppet-pe` broker for version 2015.2.0, you can update it with:

```
{
  "broker": "puppet-pe",
  "key": "version",
  "value": "2015.3.0"
}
```

Delete broker (`delete-broker`)

The `delete-broker` command requires only the name of the broker:

```
{
  "name": "small",
}
```

It is not possible to delete a broker that is used by a policy.

Hooks API

Hooks are custom, executable scripts that are triggered to run when a node hits certain phases in its lifecycle. A hook script receives several properties as input, and can make changes in the node's metadata or the hook's internal configuration.

Hooks can be useful for:

- Notifying an external system about the stage of a node's installation.
- Querying external systems for information that modifies how a node gets installed.
- Calculating complex values for use in a node's installation configuration.

Create hook (`create-hook`)

To create a new hook, use the `create-hook` command with a body like this:

```
{
  "name": "myhook",
  "hook_type": "some_hook",
  "configuration": {"foo": 7, "bar": "rhubarb"}
}
```

The `hook_type` must correspond to a hook with the specified name in a valid hook directory. Hook directories are specified in the `hook_path` class parameter of the `pe_razor` class. By default, the hook directory for custom hooks is `/etc/puppetlabs/razor-server/hooks:hooks`.

The optional `configuration` parameter lets you provide a starting configuration corresponding to that `hook_type`.

Update hook configuration (`update-hook-configuration`)

To set or clear a specified key value for a hook, use `update-hook-configuration`.

This argument changes the key "some_key" to "new_value":

```
{
  "hook": "myhook",
  "key": "some_key",
  "value": "new_value"
}
```

This argument clears the value for "some_key":

```
{
  "hook": "myhook",
  "key": "some_key",
  "clear": true
}
```

Delete hook (delete-hook)

To delete a single hook, provide its name:

```
{
  "name": "my-hook"
}
```

Upgrading Razor

If you used Razor in a previous Puppet Enterprise environment, upgrade Razor to keep your Puppet Enterprise and Razor versions synced.

After a Puppet Enterprise upgrade, the `pe_razor` class continues to operate normally, but we recommend upgrading Razor as soon as possible to avoid unintended effects.

Note: If nodes are actively provisioning during upgrade, provisioning might fail. You can resume provisioning after `pe-razor-server` restarts.

Upgrade Razor from Puppet Enterprise 2015.2.x or later

Upgrading from 2015.2.x or later is a mostly automated process that replaces the software repository, installs software packages, and migrates the Razor database. Upgrading to 2016.2 or later requires manual migration of any custom configuration from your `config.yaml` file to parameters in the `pe_razor` class.

Before you begin

If you're upgrading to Puppet Enterprise 2016.2 or later and you've modified your `config.yaml` file – for example, by changing `protect_new_nodes` or customizing tasks, brokers, or hooks – make a note of the modified settings.

For instructions on upgrading Razor from Puppet Enterprise 3.8, see earlier versions of the Razor documentation.

1. Upgrade the master.
2. Upgrade the agent on the Razor server node.

The `pe-razor-server` service automatically restarts.

3. (Optional) If you're upgrading to 2016.2 or later, transfer any customized configurations to parameters in the `pe_razor` class.

Note: To prevent accidentally overwriting machines during upgrade, the default for `protect_new_nodes` was changed to `true` in Puppet Enterprise 2016.2 and later. If your environment and workflows rely on provisioning all new nodes, you must manually change `protect_new_nodes` to `false` after upgrade, then run `puppet` and restart the `pe-razor-server` service.

Tip: You can run `razor --version` to verify that the upgrade was successful.

Related information

[Running Puppet on nodes](#) on page 320

Puppet automatically attempts to run on each of your nodes every 30 minutes. To trigger a Puppet run outside of the default 30-minute interval, you can manually run Puppet.

Uninstalling Razor

If you're permanently done provisioning nodes, you can uninstall Razor.

Uninstall the Razor server

To uninstall the Razor server, erase Razor from the server node and update your environment so that nodes continue to boot as expected.

1. On the node with the Razor server, run `yum erase pe-razor`
2. Drop the PostgreSQL database that the server used.
3. Change DHCP/TFTP so that the machines that have been installed continue to boot outside the scope of Razor.

Uninstall the Razor client

Uninstall the Razor client if you no longer wish to use it to interact with a Razor server.

Run `gem uninstall razor-client`.

SSL and certificates

Network communications and security in Puppet Enterprise are based on HTTPS, which secures traffic using X.509 certificates. PE includes its own CA tools, which you can use to regenerate certs as needed.

Regenerate certificates

Regenerating certificates and security credentials—both private and public keys—created by the built-in PE certificate authority can help ensure the security of your installation in certain cases.

The process for regenerating certificates varies depending on your goal.

If your goal is to...	Do this...
Upgrade to the intermediate certificate architecture introduced in Puppet 6.0.	Complete these tasks in order: <ol style="list-style-type: none"> 1. Delete and recreate the certificate authority on page 690 2. Regenerate compiler certificates on page 690, if applicable 3. Regenerate *nix agent certificates on page 691 or Regenerate Windows agent certificates
Fix a compromised or damaged certificate authority.	
Fix a compromised compiler certificate or troubleshoot SSL errors on compilers.	
Fix a compromised agent certificate or troubleshoot SSL errors on agent nodes.	Regenerate *nix agent certificates on page 691 or Regenerate Windows agent certificates
Specify a new DNS alt name or other trusted data.	Regenerate master certificates on page 692

Delete and recreate the certificate authority

Recreate the certificate authority only if you're upgrading to the new certificate architecture introduced in Puppet 6.0, or if your certificate authority was compromised or damaged beyond repair.

Before you begin

The `puppet infrastructure run` command leverages built-in plans to automate certain management tasks. To use this command, you must be able to connect using SSH from your master to any nodes that the command modifies. You can establish an SSH connection using key forwarding, a local key file, or by specifying keys in `.ssh/config` on your master. For more information, see [OpenSSH configuration options](#).

To view all available parameters, use the `--help` flag. The logs for all `puppet infrastructure run` Bolt plans are located at `/var/log/puppetlabs/installer/bolt_info.log`.



CAUTION: Replacing your certificate authority invalidates all existing certificates in your environment. Complete this task only if and when you're prepared to regenerate certificates for both your infrastructure nodes (including external PE-PostgreSQL in extra-large installations) and your entire agent fleet.

On your master logged in as root, run:

```
puppet infrastructure run rebuild_certificate_authority
```

The SSL and cert directories on your CA server are backed up with "_bak" appended to the end, CA files are removed and certificates are rebuilt, and a Puppet run completes.

Note: To support recovery, backups of your certificates are saved and the location of the backup directory is output to the PE console. If the command fails after deleting the certificates, you can restore your certificates with the contents of this backup directory.

Regenerate compiler certificates

Regenerate compiler certificates to fix a compromised certificate or troubleshoot SSL errors on compilers, or if you recreated your certificate authority.

Before you begin

The `puppet infrastructure run` command leverages built-in plans to automate certain management tasks. To use this command, you must be able to connect using SSH from your master to any nodes that the command modifies. You can establish an SSH connection using key forwarding, a local key file, or by specifying keys in `.ssh/config` on your master. For more information, see [OpenSSH configuration options](#).

To view all available parameters, use the `--help` flag. The logs for all `puppet infrastructure run` Bolt plans are located at `/var/log/puppetlabs/installer/bolt_info.log`.

On your master logged in as root, run:

```
puppet infrastructure run regenerate_compiler_certificate
  target=<COMPILER_HOSTNAME>
```

You can also specify these optional parameters:

- `dns_alt_names` – Comma-separated list of alternate DNS names to be added to the certificates generated for your agents.

Important: To use the `dns_alt_names` parameter, you must configure Puppet Server with `allow-subject-alt-names` in the `certificate-authority` section of `ca.conf`. To ensure naming consistency, if your `puppet.conf` file includes a `dns_alt_names` entry, you must include the `dns_alt_names` parameter and pass in all alt names included in the entry when regenerating your compiler certificates.

- `tmpdir` — Path to a directory to use for uploading and executing temporary files.

The compiler's SSL directory is backed up to `/etc/puppetlabs/puppet/ssl_bak`, its certificate is regenerated and signed, a Puppet run completes, and the compiler resumes its role in your deployment.

Note: To support recovery, backups of your certificates are saved and the location of the backup directory is output to the PE console. If the command fails after deleting the certificates, you can restore your certificates with the contents of this backup directory.

Regenerate *nix agent certificates

Regenerate *nix agent certificates to fix a compromised certificate or troubleshoot SSL errors on agents, or if you recreated your certificate authority.

Before you begin

The `puppet infrastructure run` command leverages built-in plans to automate certain management tasks. To use this command, you must be able to connect using SSH from your master to any nodes that the command modifies. You can establish an SSH connection using key forwarding, a local key file, or by specifying keys in `.ssh/config` on your master. For more information, see [OpenSSH configuration options](#).

To view all available parameters, use the `--help` flag. The logs for all `puppet infrastructure run` Bolt plans are located at `/var/log/puppetlabs/installer/bolt_info.log`.

On your master logged in as root, run:

```
puppet infrastructure run regenerate_agent_certificate
agent=<AGENT_HOSTNAME>
```

You can also specify these optional parameters:

- `dns_alt_names` – Comma-separated list of alternate DNS names to be added to the certificates generated for your agents.

Important: To use the `dns_alt_names` parameter, you must configure Puppet Server with `allow-subject-alt-names` in the `certificate-authority` section of `ca.conf`. To ensure naming consistency, if your `puppet.conf` file includes a `dns_alt_names` entry, you must include the `dns_alt_names` parameter and pass in all alt names included in the entry when regenerating your agent certificates.

- `tmpdir` — Path to a directory to use for uploading and executing temporary files.

To regenerate certificates for multiple *nix agents, use a comma-separated list of the agents' hostnames, such as:

```
puppet infrastructure run regenerate_agent_certificate
agent=agent1.example.net,agent2.example.net,agent3.example.net
```

The *nix agent's SSL directory is backed up to `/etc/puppetlabs/puppet/ssl_bak`, its certificate is regenerated and signed, a Puppet run completes, and the agent resumes its role in your deployment.

Note: To support recovery, backups of your certificates are saved and the location of the backup directory is output to the PE console. Note that the location shown in the console is relative to your agent's files. If the command fails after deleting the certificates, you can restore your certificates with the contents of this backup directory.

Regenerate Windows agent certificates

Regenerate Windows agent certificates to fix a compromised certificate or troubleshoot SSL errors on agents, or if you recreated your certificate authority.

Unless otherwise indicated, perform these steps on the Windows agent node that you're regenerating certificates for.

1. If you did not recreate your certificate authority, you must log into your master and clear the cert for the agent node:

```
puppetserver ca clean --certname <CERTNAME>
```

2. On the agent, back up the `/etc/puppetlabs/puppet/ssl/` directory.

If something goes wrong, you might need to restore these directories so your deployment remains functional.

3. Stop the Puppet agent and PXP agent services.

```
puppet resource service puppet ensure=stopped
puppet resource service pxp-agent ensure=stopped
```

4. Delete the agent SSL directory by using the Administrator confdir to delete the `$confdir\ssl` directory.
5. Remove the agent's cached catalog. Use the Administrator confdir to delete the `$client_datadir\catalog\<CERTNAME>.json` file.
6. Re-start the Puppet agent service: `puppet resource service puppet ensure=running`
After the agent starts, it automatically generates keys and request a new certificate from the Puppet CA.
7. If you aren't using autosigning, sign each agent node's certificate request using the console's request manager, or from your master:

```
puppetserver ca list
puppetserver ca sign --certname <NAME>
```

Note: For more information about autosigning, see [Autosigning certificate requests](#).

8. From the console or command line, run Puppet on the node.

The Windows agent performs a full catalog run, restarts the PXP agent service, and resumes its role in your PE deployment.

Regenerate master certificates

Regenerate master certificates to specify a new DNS alt name or other trusted data. This process regenerates the certificates for all primary infrastructure nodes, including external PE-PostgreSQL in extra-large installations.

Before you begin

The `puppet infrastructure run` command leverages built-in plans to automate certain management tasks. To use this command, you must be able to connect using SSH from your master to any nodes that the command modifies. You can establish an SSH connection using key forwarding, a local key file, or by specifying keys in `.ssh/config` on your master. For more information, see [OpenSSH configuration options](#).

To view all available parameters, use the `--help` flag. The logs for all `puppet infrastructure run` Bolt plans are located at `/var/log/puppetlabs/installer/bolt_info.log`.

On your master logged in as root, run:

```
puppet infrastructure run regenerate_master_certificate
```

You can specify these optional parameters:

- `dns_alt_names` – Comma-separated list of alternate DNS names to be added to the certificates generated for your master.

Important: To use the `dns_alt_names` parameter, you must configure Puppet Server with `allow-subject-alt-names` in the `certificate-authority` section of `ca.conf`. To ensure naming consistency, if your `puppet.conf` file includes a `dns_alt_names` entry, you must include the `dns_alt_names` parameter and pass in all alt names included in the entry when regenerating your master certificate.

- `tmpdir` — Path to a directory to use for uploading and executing temporary files.

The certificates are regenerated for your primary infrastructure nodes.

Note: To support recovery, backups of your certificates are saved and the location of the backup directory is output to the PE console. If the command fails after deleting the certificates, you can restore your certificates with the contents of this backup directory.

Regenerate replica certificates

Regenerate master replica certificates for your high availability installation to specify a new DNS alt name or other trusted data.

Before you begin

The `puppet infrastructure run` command leverages built-in plans to automate certain management tasks. To use this command, you must be able to connect using SSH from your master to any nodes that the command modifies. You can establish an SSH connection using key forwarding, a local key file, or by specifying keys in `.ssh/config` on your master. For more information, see [OpenSSH configuration options](#).

To view all available parameters, use the `--help` flag. The logs for all `puppet infrastructure run` Bolt plans are located at `/var/log/puppetlabs/installer/bolt_info.log`.

On your master logged in as root, run:

```
puppet infrastructure run regenerate_replica_certificate
  target=<REPLICA_HOSTNAME>
```

You can specify these optional parameters:

- `dns_alt_names` – Comma-separated list of alternate DNS names to be added to the certificates generated for your master.

Important: To use the `dns_alt_names` parameter, you must configure Puppet Server with `allow-subject-alt-names` in the `certificate-authority` section of `ca.conf`. If your `puppet.conf` file includes a `dns_alt_names` entry, you must include the `dns_alt_names` parameter and pass in all alt names included in the entry when regenerating your replica certificate.

- `tmpdir` — Path to a directory to use for uploading and executing temporary files.

The replica's SSL directory is backed up to `/etc/puppetlabs/puppet/ssl_bak`, its certificate is regenerated and signed, a Puppet run completes, and the replica resumes its role in your deployment.

Note: To support recovery, backups of your certificates are saved on your replica and the location of the backup directory is output to the PE console. If the command fails after deleting the certificates, you can restore your certificates with the contents of this backup directory.

Use an independent intermediate certificate authority

The built-in Puppet certificate authority automatically generates a root and intermediate certificate, but you can set up an independent intermediate certificate authority during installation if you need additional intermediate certificates, or if you prefer to use a public authority CA.

Before you begin

You must have a CA chain, a CRL chain, and a private key. Certificate chains must be ordered from least to most authoritative, with the cert for the CA that you intend to use positioned first in the chain files.

1. Copy your CA chain, CRL chain, and private key to the node where you're installing the master.

Tip: Allow access to your private key only from the PE installation process, which runs as root.

2. Follow the instructions to install PE in text mode, adding the `signing_ca` parameter to `pe.conf`.

You must include all three key/value pairs for the `signing_ca` parameter: `bundle`, `crl_chain`, and `private_key`.

```
{
  "pe_install::signing_ca": {
    "bundle": "/root/ca/int_ca_bundle"
    "crl_chain": "/root/ca/int_crl_chain"
    "private_key": "/root/ca/int_key"
  }
}
```

3. Validate that the CA is working using the standard openssl tools installed with PE.

```
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
  ...
```

Related information

[Install using text install](#) on page 136

When you run the installer in text mode, you provide a configuration file (`pe.conf`) to the installer. The `pe.conf` file contains values for the parameters needed for installation.

Use a custom SSL certificate for the console

The console uses a certificate signed by PE's built-in certificate authority (CA). Because this CA is specific to PE, web browsers don't know it or trust it, and you have to add a security exception in order to access the console. You might find that this is not an acceptable scenario and want to use a custom CA to create the console's certificate.

Before you begin

- You should have a X.509 cert, signed by the custom party CA, in PEM format, with matching private and public keys.
 - If your custom cert is issued by an intermediate CA, the CA bundle needs to contain a complete chain, including the applicable root CA.
 - The keys and certs used in this procedure must be in PEM format.
1. Retrieve the custom certificate's public and private keys, and, for ease of use, name them as follows:
 - `public-console.cert.pem`
 - `public-console.private_key.pem`
 2. Add the files from step 1 to `/opt/puppetlabs/server/data/console-services/certs/`.

3. Use the console to edit the parameters of the `puppet_enterprise::profile::console` class.
 - a) Click **Classification**, and in the **PE Infrastructure** group, select the **PE Console** group.
 - b) On the **Configuration** tab, in the `puppet_enterprise::profile::console` class, add the following parameters:

Parameter	Value
<code>browser_ssl_cert</code>	<code>/opt/puppetlabs/server/data/console-services/certs/public-console.cert.pem</code>
<code>browser_ssl_private_key</code>	<code>/opt/puppetlabs/server/data/console-services/certs/public-console.private_key.pem</code>

- c) Commit changes.

4. Run Puppet.
5. When the run is complete, restart the console and the nginx service for the changes to take effect.

```
puppet resource service pe-console-services ensure=stopped
puppet resource service pe-console-services ensure=running
puppet resource service pe-nginx ensure=stopped
puppet resource service pe-nginx ensure=running
```

You should now be able to navigate to your console and see the custom certificate in your browser.

Change the hostname of a master

To change the hostnames assigned to your PE infrastructure nodes requires updating the corresponding PE certificate names. You might have to update your master hostname, for example, when migrating to a new PE installation, or after an organizational change such as a change in company name.

Before you begin

Download and install the [puppetlabs-support_tasks](#) module.

Make sure you are using the latest version of the `support_tasks` module.

To update the hostname of your master:

1. On the master, set the new hostname by running the following command:

```
hostnamectl set-hostname newhostname.example.com
```

2. Make sure the `hostname -f` command returns the new fully qualified hostname, and that it resolves to the same IP address as the old hostname, by adding an entry for the new hostname in `/etc/hosts`:

```
<IP address> <newhostname.example.com> <oldhostname.example.com> ...
```

3. Run a task for changing the host name against the old certificate name on the master, using one of the following methods:

- Using Bolt:

```
bolt task run support_tasks::st0263_rename_pe_master -n $(puppet config
  print certname) --modulepath="/etc/puppetlabs/code/environments/
  production/modules"
```

Note: When running the task, Bolt must be using the default SSH transport, rather than the PCP protocol, to avoid errors when services are restarted.

- Using the command line:

```
puppet task run support_tasks::st0263_rename_pe_master -n $(puppet
  config print certname)
```

The task restarts all Puppet services, which causes a connection error. You can ignore the error while the task continues to run in the background. To check if the task is complete, tail `/var/log/messages`. When the output from the `puppet agent -t` command is displayed in the system log, the task is complete. For example:

```
# tail /var/log/messages
Aug 15 09:08:28 oldhostname systemd: Reloading pe-orchestration-services
Service.
Aug 15 09:08:29 oldhostname systemd: Reloaded pe-orchestration-services
Service.
Aug 15 09:08:29 oldhostname puppet-agent[4780]: (/
Stage[main]/Puppet_enterprise::Profile::Orchestrator/
Puppet_enterprise::Trapperkeeper::Pe_service[orchestration-services]/
Service[pe-orchestration-services]) Triggered 'refresh' from 1 event
Aug 15 09:08:34 oldhostname puppet-agent[4780]: Applied catalog in 19.16
seconds
```

Generate a custom Diffie-Hellman parameter file

The "Logjam Attack" (CVE-2015-4000) exposed several weaknesses in the Diffie-Hellman (DH) key exchange. To help mitigate the "Logjam Attack," PE ships with a pre-generated 2048 bit Diffie-Hellman param file. In the case that you don't want to use the default DH param file, you can generate your own.

Note: In the following procedure, `<PROXY-CUSTOM-dhparam>.pem` can be replaced with any file name, except `dhparam_puppetproxy.pem`, as this is the default file name used by PE.

1. On the console node, (for a mono install, this is the same node as the Puppet master), run the following command:

```
/opt/puppetlabs/puppet/bin/openssl dhparam -out /etc/puppetlabs/nginx/
<PROXY-CUSTOM-dhparam>.pem 2048
```

Note: After running this command, PE can take several minutes to complete this step.

2. On the master, open your `pe.conf` file (located at `/etc/puppetlabs/enterprise/conf.d/pe.conf`) and add the following parameter and value:

```
"puppet_enterprise::profile::console::proxy::dhparam_file": "/etc/
puppetlabs/nginx/<PROXY-CUSTOM-dhparam>.pem"
```

3. On the console node, run Puppet: `puppet agent -t`.

Enable TLSv1

TLSv1 and TLSv1.1 are disabled by default in PE.

You must enable TLSv1 to install agents on these platforms:

- AIX
- CentOS 5
- RHEL 5
- SLES 11
- Solaris 10
- Windows Server 2008r2

To comply with security regulations, PE 2019.1 and later uses only version 1.2 of the Transport Layer Security (TLS) protocol.

1. In the console, click **Classification** > **PE Infrastructure**.
2. On the **Configuration** tab, in the **Data** area, add the following class, parameter, and value:

Class	Parameter	Value
puppet_enterprise::master	ssl_puppetserver	["TLSv1", "TLSv1.1", "TLSv1.2"]

3. Click **Add data**, and commit changes.
4. Run Puppet on the master.

Maintenance

Backing up and restoring Puppet Enterprise

Keep regular backups of your PE infrastructure. Backups allow you to more easily migrate to a new master, troubleshoot, and quickly recover in the case of system failures.

Note: These instructions are for standard (previously called monolithic) installations only. Split installations, where the master, console, and PuppetDB are installed on separate nodes, are no longer supported in PE. We recommend migrating from an existing split installation to a standard installation—with or without compilers—and a standalone PE-PostgreSQL node. See the docs for how to [Migrate from a split to a standard installation](#) on page 184.

Important: Always store your backups in a safe location that you can access if your master fails. Backup files are not encrypted, so secure these as you would any sensitive information.

You can use the `puppet-backup` command to back up and restore your master. You can't use this command to back up compilers. By default, the backup command creates a backup of:

- Your PE configuration, including license, classification, and RBAC settings. However, configuration backup does not include Puppet gems or Puppet Server gems.
- PE CA certificates and the full SSL directory.
- The Puppet code deployed to your code directory at backup time.
- PuppetDB data, including facts, catalogs and historical reports.

Each time you create a new backup, PE creates a single, timestamped backup file, named in the format `pe_backup-
<TIMESTAMP>.tgz`. This file includes everything you're backing up. By default, PE writes backup files to `/var/puppetlabs/backups`, but you can change this location when you run the backup command. When you restore, specify the backup file you want to restore from.

If you are restoring to a previously existing master, uninstall and reinstall PE before restoring your infrastructure. If you are restoring or migrating your infrastructure to a master with a different hostname than the previous master, you'll redirect your agents to the new master during the restore process. In both cases, the freshly installed PE must be the same PE version that was in use when you backed up the files.

By default, backup and restore functions include your Puppet configuration, certificates, code, and PuppetDB. However, you can limit the scope of backup and restore with command line options. This allows you to back up to or restore from multiple files. This is useful if you want to back up some parts of your infrastructure more often than others.

For example, if you have frequent code changes, you might back up the code more often than you back up the rest of your infrastructure. When you limit backup scope, the backup file contains only the specified parts of your infrastructure. Be sure to give your backup file a name that identifies the scope so that you always know what a given file contains.

During restore, you must restore all scopes: code, configuration, certificates, and PuppetDB. However, you can restore each scope from different files, either by restoring from backup files with limited scope or by limiting the scope of the restore. For example, by specifying scope when you run the restore command, you could restore code, configuration, and certificates from one backup file and PuppetDB from a different one.

Back up your infrastructure

PE backup creates a copy of your Puppet infrastructure, including configuration, certificates, code, and PuppetDB.

Before you begin

You can back up only your master. The backup and restore commands are not supported for compilers.

1. On your master, from the command line logged in as root, run `puppet-backup create`

To change the default command behavior, pass option flags and values to the command. See the backup and restore [reference](#) for a complete list of options.

For example, to limit a backup to certain parts of your PE infrastructure, pass the `--scope` option, specifying the scopes in a comma-separated list. To change the name of your backup file, pass the `--name` option with a string specifying the filename. For example, to back up PuppetDB only and name your file with the scope and date, run:

```
puppet-backup create --scope=puppetdb --name=puppetdb_backup_03032018.tgz
```

2. Make a backup of the secret key used to encrypt and decrypt sensitive data stored in the inventory service.

Secure the key as you would any sensitive information. The secret key is stored at: `/etc/puppetlabs/orchestration-services/conf.d/secrets/keys.json`



CAUTION: The backup command does not include the secret key. You must back up this information separately.

By default, if you don't specify the `--dir` or `--name` options, PE creates files to `/var/puppetlabs/backups` and names them with a timestamp, such as `pe_backup-<TIMESTAMP>.tgz`

After backing up, you can move your backup files to another location. Always store your backups in a safe location that is not on the master.

Related information

[Configuration parameters and the `pe.conf` file](#) on page 138

A `pe.conf` file is a HOCON formatted file that declares parameters and values needed to install, upgrade, and configure PE.

Restore your infrastructure

Use the restore commands to migrate your PE master to a new host or to recover from system failure.

Before you begin

Remember that you must restore files to a fresh installation of the same version of PE used in your backup file.

1. If you are restoring to an existing master, purge any existing PE installation from it.
 - a) On the master, uninstall PE by running `sudo /opt/puppetlabs/bin/puppet-enterprise-uninstaller -p -d`
 - b) Ensure that the directories `/opt/puppetlabs/` and `/etc/puppetlabs/` are no longer present on the system.

For details about uninstalling PE, see the [uninstalling](#) documentation.

2. Install PE on the master you are restoring to. This must be the same PE version that was used for the backup files.
 - a) If you don't have the PE installer script on the machine you want to restore to, download the installer tarball to the machine from the [download](#) site, and unpack it by running `tar -xf <TARBALL_FILENAME>`
 - b) Navigate to the directory containing the install script. The installer script is located in the PE directory created when you unpacked the tarball.
 - c) Install PE by running `sudo ./puppet-enterprise-installer`

For details about the PE installation process, see the [installing](#) documentation.

3. On the master logged in as root, restore your PE infrastructure by running `puppet-backup restore <BACKUP-FILENAME>`

To change the default command behavior, pass option flags and values to the command. See the [command reference](#) for a complete list of options.

For example, to restore your scope, certificates, and code from one backup file, but your PuppetDB data from another, pass the `--scope` option:

```
puppet-backup restore pe_backup-2018-03-03_20.07.15.UTC.tgz --
scope=config,certs,code
```

```
puppet-backup restore pe_backup-2018-04-04_20.07.15.UTC.tgz --
scope=puppetdb
```

4. Restore your backup of the secret key used to encrypt and decrypt sensitive data stored in the inventory service. The secret key is stored at: `/etc/puppetlabs/orchestration-services/conf.d/secrets/keys.json`



CAUTION: The restore command does not include the secret key. You must restore this information separately.

5. Ensure the secret key ownership is `pe-orchestration-services:pe-orchestration-services`.
6. Restart the `pe-orchestration-services` service.

7. If you restored PE onto a master with a different hostname than the original installation, and you have not configured the `dns_alt_names` setting in the `pe.conf` file, redirect your agents to the new master. An easy way to do this is by running a task with the Bolt task runner.

- Download and [install](#) Bolt, if you don't already have it installed.
- Update the `puppet.conf` file to point all agents at the new master by running:

```
bolt task run puppet_conf action=set section=agent setting=server
value=<RESTORE_HOSTNAME> --nodes <COMMA-SEPARATED LIST OF NODES>
```

- Run the agent on all nodes by running:

```
bolt command run "puppet agent -t" --nodes <COMMA-SEPARATED LIST OF
NODES>
```

8. If your infrastructure had Code Manager enabled when your backup file was created, deploy your code by running:

```
puppet access login
puppet code deploy --all --wait
```

9. Test the agents' connection to the master by running `puppet agent --test`

Related information

[Configuration parameters and the `pe.conf` file](#) on page 138

A `pe.conf` file is a HOCON formatted file that declares parameters and values needed to install, upgrade, and configure PE.

[Installing Puppet Enterprise](#) on page 135

You can install PE using *express install*, which relies on defaults, or *text install*, where you provide a `pe.conf` file with installation parameters. Either of these methods is appropriate for installing infrastructure components on your master.

[Uninstalling](#) on page 177

Puppet Enterprise includes a script for uninstalling. You can uninstall component infrastructure nodes or from agent nodes.

Backup and restore reference

Use these options to change the backup and restore scope and other options for the `puppet-backup` command.

Note: Backup commands must be run as root.

`puppet-backup create`

Run the `puppet-backup create` command to create backup files of your PE infrastructure.

Usage:

```
puppet-backup create [--dir=<DIRECTORY_PATH>] [--name=<BACKUP_NAME>.tgz] [--
scope=<SCOPE_LIST>] [--force]
```

Option	Description	Values	Default
<code>--dir=BACKUP_DIR</code>	Specifies the directory to write the backup file to.	A valid filepath that the <code>pe-postgres</code> user has write permission for.	<code>/var/puppetlabs/backups/</code>
<code>--name=BACKUP_NAME.tgz</code>	Specifies the name for the backup file.	A string designating a file name.	<code>pe_backup-<TIMESTAMP>.tgz</code>

Option	Description	Values	Default
<code>--pe-environment=ENVIRONMENT</code>	Specifies the environment to back up. To ensure configuration is recovered correctly, this must be the environment where your master is located.	A valid environment name.	production
<code>--scope=SCOPE</code>	Scope of backup to create.	Either all or any combination of the other available scopes, as a comma-separated list: <ul style="list-style-type: none"> • <code>config</code>: PE configuration including license, classification, and RBAC settings. Does not include Puppet gems or Puppet Server gems. • <code>certs</code>: PE CA certificates and full SSL directory • <code>code</code>: Puppet code deployed to your codedir at backup time. • <code>puppetdb</code>: PuppetDB data, including facts, catalogs, and historical reports • <code>all</code>: All listed scopes. 	all
<code>--force</code>	Bypass validation checks and ignore warnings.	None.	If you don't specify <code>--force</code> , PE verifies that the destination directory exists and has enough space for the backup process.

For example, to create a backup of PuppetDB only and give it a name that shows the scope of the file:

```
puppet-backup create --scope=puppetdb --name=puppetdb_backup_03032018.tgz
```

puppet-backup restore

Run the `puppet-backup restore` command to restore your PE infrastructure from backup files.

Usage:

```
puppet-backup restore <PATH/TO/BACKUP_FILE.tgz> [--scope=<SCOPE_LIST>] [--force]
```

Option	Description	Values	Default
<code>--pe-environment=ENVIRONMENT</code>	Specifies the environment to restore.	A valid environment name for which you have an existing backup.	production

Option	Description	Values	Default
<code>--scope=SCOPE</code>	Scope of backup to restore. All scopes must eventually be restored, but you can restore different scopes from different backup files with successive restore commands.	Either <code>all</code> or any combination of the other available scopes, as a comma-separated list: <ul style="list-style-type: none"> <code>config</code>: PE configuration including license, classification, and RBAC settings. Does not include Puppet gems or Puppet Server gems. <code>certs</code>: PE CA certificates and full SSL directory <code>code</code>: Puppet code deployed to your <code>codedir</code> at backup time. <code>puppetdb</code>: PuppetDB data, including facts, catalogs, and historical reports <code>all</code>: All listed scopes. 	<code>all</code>
<code>--force</code>	Bypass validation checks and ignore warnings.	None.	If you don't specify <code>--force</code> , PE verifies that the destination directory exists and has enough space for the restore process. Returns warnings for insufficient space or invalid locations.

For example, to restore PuppetDB from one backup file and restore configuration, certificates, and code from another backup file:

```
puppet-backup restore /mybackups/pe_backup_03032018.tgz --scope=puppetdb
```

```
puppet-backup restore /mybackups/pe_backup_04042018.tgz --
scope=config,certs,code
```

Directories and data backed up

Scope	Directories and databases backed up
<code>certs</code>	<ul style="list-style-type: none"> <code>/etc/puppetlabs/puppet/ssl/</code>
<code>code</code>	<ul style="list-style-type: none"> <code>/etc/puppetlabs/code/</code> <code>/etc/puppetlabs/code-staging/</code> <code>/opt/puppetlabs/server/data/puppetserver/filesync/storage/</code>

Scope	Directories and databases backed up
config	<ul style="list-style-type: none"> • Orchestrator database • RBAC database • Classifier database • /etc/puppetlabs/ , except: <ul style="list-style-type: none"> • /etc/puppetlabs/code/ • /etc/puppetlabs/code-staging/ • /etc/puppetlabs/puppet/ssl • /opt/puppetlabs/ , except: <ul style="list-style-type: none"> • /opt/puppetlabs/puppet • /opt/puppetlabs/server/pe_build • /opt/puppetlabs/server/data/packages • /opt/puppetlabs/server/apps • /opt/puppetlabs/server/data/postgresql • /opt/puppetlabs/server/data/enterprise/modules • /opt/puppetlabs/server/data/puppetserver/vendored-jruby-gems • /opt/puppetlabs/bin • /opt/puppetlabs/client-tools • /opt/puppetlabs/server/share • /opt/puppetlabs/server/data/puppetserver/filesync/storage • /opt/puppetlabs/server/data/puppetserver/filesync/client
puppetdb	<ul style="list-style-type: none"> • PuppetDB • /opt/puppetlabs/server/data/puppetdb

Database maintenance

You can optimize the Puppet Enterprise (PE) databases to improve performance.

Databases in Puppet Enterprise

PE uses PostgreSQL as the backend for its databases. Use the native tools in PostgreSQL to perform database exports and imports.

The PE PostgreSQL database includes the following databases:

Database	Description
pe-activity	Activity data from the Classifier, including user, nodes, and time of activity.
pe-classifier	Classification data, all node group information.
pe-puppetdb	PuppetDB data, including exported resources, catalogs, facts, and reports.

Database	Description
pe-rbac	Role-based access control data, including users, permissions, and AD/LDAP info.
pe-orchestrator	Orchestrator data, including user, node, and result details about job runs.

Optimize a database

If your databases are slow, begin taking up too much disk space, or need general performance enhancements, use the PostgreSQL vacuum command to optimize any of the PE databases.

To optimize a database, run: `su - pe-postgres -s /bin/bash -c "/opt/puppetlabs/server/bin/vacuumdb -z --verbose <DATABASE_NAME> "`

Related information

[List all database names](#) on page 704

Use these instructions to list all database names.

List all database names

Use these instructions to list all database names.

To generate a list of database names:

1. Assume the `pe-postgres` user:

```
sudo su - pe-postgres -s /bin/bash
```

2. Open the PostgreSQL command-line:

```
/opt/puppetlabs/server/bin/psql
```

3. List the databases:

```
\l
```

4. Exit the PostgreSQL command line:

```
\q
```

5. Log out of the `pe-postgres` user:

```
logout
```

Troubleshooting

Use this guide to troubleshoot issues with your Puppet Enterprise installation.

Troubleshooting installation

If installation fails, check for these issues.

Note: If you encounter errors during installation, you can troubleshoot and run the installer as many times as needed.

DNS is misconfigured

DNS must be configured correctly for successful installation.

1. Verify that agents can reach the master hostname you chose during installation.
2. Verify that the master can reach *itself* at the master hostname you chose during installation.
3. If the master and console components are on different servers, verify that they can communicate with each other.

Security settings are misconfigured

Firewall and security settings must be configured correctly for successful installation.

1. On your master, verify that inbound traffic is allowed on ports 8140, 61613, and 443.
2. If your master has multiple network interfaces, verify that the master allows traffic via the IP address that its valid DNS names resolve to, not just via an internal interface.

Troubleshooting high availability

If high availability commands fail, check for these issues.

Latency over WAN

If the master and replica communicate over a slow, high latency, or lossy connection, the `provision` and `enable` commands can fail.

If this happens, try re-running the command.

Replica is connected to a compiler instead of a master

The `provision` command triggers an error if you try to provision a replica node that's connected to a compiler. The error is similar to the following:

```
Failure during provision command during the puppet agent run on replica 2:
Failed to generate additional resources using 'eval_generate':
  Error 500 on SERVER: Server Error: Not authorized
  to call search on /file_metadata/pe_modules with
  {:rest=>"pe_modules", :links=>"manage", :recurse=>true, :source_permissions=>"ignore",
  Source: /Stage[main]/Puppet_enterprise::Profile::Primary_master_replica/
  File[/opt/puppetlabs/server/share/installer/modules]File: /opt/
  puppetlabs/puppet/modules/puppet_enterprise/manifests/profile/
  primary_master_replica.ppLine: 64
```

On the replica you want to provision, edit `/etc/puppetlabs/puppet.conf` so that the `server` and `server_list` settings use a master, rather than a compiler.

Both `server` and `server_list` are set in the agent configuration file

When the agent configuration file contains settings for both `server` and `server_list`, a warning appears. This warning can occur after enabling a replica in a high availability configuration. You can ignore the warning, or hide it by removing the `server` setting from the agent configuration, leaving only `server_list`.

Node groups are empty

When provisioning and enabling a replica, the orchestrator is used to run Puppet on different groups of nodes. If a group of nodes is empty, the tool reports that there's nothing for it to do and the job is marked as `failed` in the output of `puppet job show`. This is expected, and doesn't indicate a problem.

Troubleshooting puppet infrastructure run commands

If puppet infrastructure run commands fail, review the logs at `/var/log/puppetlabs/installer/bolt_info.log` and check for these issues.

Running commands when logged in as a non-root user

All puppet infrastructure run commands require you to act as the root user on all nodes that the command touches. If you are trying to run a puppet infrastructure run command as a non-root user, you must be able to SSH to the impacted nodes (as the same non-root user) in order for the command to succeed.

When you run a puppet infrastructure run command, Bolt uses your system's existing OpenSSH `ssh_config` configuration file to connect to your nodes. If this file is missing or configured incorrectly, Bolt tries to connect as root. To make sure the correct user connects to the nodes, you have the following options.

1. Set up your OpenSSH `ssh_config` configuration file to point to a user with sudo privileges:

```
Host *.example.net
  UserKnownHostsFile=~/.ssh/known_hosts
  User <USER WITH SUDO PRIVILEGES>
```

2. When running a puppet infrastructure run command, pass in the `--user <USER WITH SUDO PRIVILEGES>` flag.

If your sudo configuration requires a password to run commands, pass in the `--sudo-password <PASSWORD>` flag when running a puppet infrastructure run command.

Note: To avoid logging your password to `.bash_history`, set `HISTCONTROL=ignorespace` in your `.bashrc` file and add a space to the beginning of the command.

If your operating system distribution includes the `requiretty` option in its `/etc/sudoers` file, you must either remove this option from the file or pass the `--tty` flag when running a puppet infrastructure run command.

Passing hashes from the command line

When passing a hash on the command line as part of a puppet infrastructure run command, the hash must be quoted, much like a JSON object. For example:

```
'{"parameter_one": "value_one", "parameter_two": "value_two"}'
```

Troubleshooting connections between components

If agent nodes can't retrieve configurations, check for communication, certificate, DNS, and NTP issues.

Agents can't reach the Puppet master

Agent nodes must be able to communicate with the Puppet master in order to retrieve configurations.

If agents can't reach the Puppet master, running `telnet <puppet master's hostname> 8140` returns the error "Name or service not known."

1. Verify that the Puppet master server is reachable at a DNS name your agents recognize.
2. Verify that the `pe-puppetserver` service is running.

Agents don't have signed certificates

Agent certificates must be signed by the Puppet master.

If the node's Puppet agent logs have a warning about unverified peer certificates in the current SSL session, the agent has submitted a certificate signing request that hasn't yet been signed.

1. On the master, view a list of pending certificate requests: `puppet cert list`
2. Sign a specified node's certificate: `puppetserver ca sign <NODE NAME>`

Agents aren't using the master's valid DNS name

Agents trust the master only if they contact it at one of the valid hostnames specified when the master was installed.

On the node, if the results of `puppet agent --configprint server` don't return one of the valid DNS names you chose during installation of the master, the node and master can't establish communication.

1. To edit the master's hostname on nodes, in `/etc/puppetlabs/puppet/puppet.conf`, change the `server` setting to a valid DNS name.
2. To reset the master's valid DNS names, run:

```
/etc/init.d/pe-nginx stop
puppet cert clean <MASTER_CERTNAME>
puppet cert generate <MASTER_CERTNAME> --dns_alt_names=<COMMA-
SEPARATED_LIST_OF_DNS_NAMES>
/etc/init.d/pe-nginx start
```

Time is out of sync

The date and time must be in sync on the Puppet master and agent nodes.

If time is out of sync on nodes, running `date` returns incorrect or inconsistent dates.

Get time in sync by setting up NTP. Keep in mind that NTP can behave unreliably on virtual machines.

Node certificates have invalid dates

The date and time must be in sync when certificates are created.

If certificates were signed out of sync, running `openssl x509 -text -noout -in $(puppet config print --section master ssl_dir)/certs/<NODE NAME>.pem` returns invalid dates, such as certificates dated in the future.

1. On the master, delete certificates with invalid dates: `puppet cert clean <NODE NAME>`
2. On nodes with invalid certificates, delete the SSL directory: `rm -r $(puppet config print --section master ssl_dir)`
3. On agent nodes, generate a new certificate request: `puppet agent --test`
4. On the master, sign the request: `puppetserver ca sign <NODE NAME>`

A node is re-using a certname

If a node re-uses an old node's certname and the master retains the previous node's certificate, the new node is unable to request a new certificate.

1. On the master, clear the node's certificate: `puppetserver ca clean <NODE NAME>`
2. On agent node, generate a new certificate request: `puppet agent --test`
3. On the master, sign the request: `puppetserver ca sign <NODE NAME>`

Agents can't reach the filebucket server

If the master is installed with a certname that doesn't match its hostname, agents can't back up files to the filebucket on the Puppet master.

If agents log errors like "could not back up," nodes are likely attempting to back up files to the wrong hostname.

On the master, edit `/etc/puppetlabs/code/environments/production/manifests/site.pp` so that filebucket server attribute points to the correct hostname:

```
# Define filebucket 'main':
filebucket { 'main':
  server => '<PUPPET_MASTER_DNS_NAME>',
  path   => false,
}
```

Changing the filebucket server attribute on the master fixes the error on all agent nodes.

Orchestrator can't connect to PE Bolt server

Debug a faulty connection between the orchestrator and PE Bolt server by setting the `bolt_server_loglevel` in the `puppet_enterprise::profile::bolt_server` class and running Puppet, or by manually updating `loglevel` in `/etc/puppetlabs/bolt-server/conf.d/bolt-server.conf`. The server logs are located at `/var/log/puppetlabs/bolt-server/bolt-server.log`.

Troubleshooting the databases

If you have issues with the databases that support the console, make sure that the PostgreSQL database is not too large or using too much memory, that you don't have port conflicts, and that `puppet apply` is configured correctly.

Note: If you're using your own instance of PostgreSQL for the console and PuppetDB, you must use version 9.1 or higher.

Related information

[Install using text install](#) on page 136

When you run the installer in text mode, you provide a configuration file (`pe.conf`) to the installer. The `pe.conf` file contains values for the parameters needed for installation.

PostgreSQL is taking up too much space

The PostgreSQL `autovacuum=on` setting prevents the database from growing too large and unwieldy. Routine vacuuming is turned on by default.

Verify that `autovacuum` is set to on.

PostgreSQL buffer memory causes installation to fail

When installing PE on machines with large amounts of RAM, the PostgreSQL database might use more shared buffer memory than is available.

If this issue is present, `/var/log/pe-postgresql/pgstartup.log` shows the error:

```
FATAL: could not create shared memory segment: No space left on device
DETAIL: Failed system call was shmget(key=5432001, size=34427584512,03600).
```

1. On the master, set the `shmmax` kernel setting to approximately 50% of the total RAM.
2. Set the `shmall` kernel setting to the quotient of the new `shmmax` setting divided by the page size. You can confirm page size by running `getconf PAGE_SIZE`.

3. Set the new kernel settings:

```
sysctl -w kernel.shmmax=<your shmmax calculation>
sysctl -w kernel.shmall=<your shmall calculation>
```

The default port for PuppetDB conflicts with another service

By default, PuppetDB communicates over port 8081. In some cases, this might conflict with existing services, for example McAfee ePolicy Orchestrator.

Install in text mode with a parameter in `pe.conf` that specifies a different port using `puppet_enterprise::puppetdb_port`.

puppet resource generates Ruby errors after connecting puppet apply to PuppetDB

If Puppet apply is configured incorrectly, for example by modifying `puppet.conf` to add the parameters `storeconfigs_backend = puppetdb` and `storeconfigs = true` in both the main and master sections, `puppet resource` ceases to function and displays a Ruby run error.

Modify `/etc/puppetlabs/puppet/routes.yaml` to correctly [connect Puppet apply](#) without affecting other functions.

Troubleshooting Windows

Troubleshoot Windows issues with failed installations and upgrades, and failed or incorrectly applied manifests. Use the error message reference to solve your issues. Enable debugging.

Installation fails

Check for these issues if Windows installation with Puppet fails.

The installation package isn't accessible

The source of an MSI or EXE package must be a file on either a local filesystem, a network mapped drive, or a UNC path.

RI-based installation sources aren't supported, but you can achieve a similar result by defining a file whose source is the Puppet master and then defining a package whose source is the local file.

Installation wasn't attempted with admin privileges

Puppet fails to install when trying to perform an unattended installation from the command line. A "norestart" message is returned, and installation logs indicate that installation is forbidden by system policy.

You must install as an administrator.

Upgrade fails

The Puppet MSI package overwrites existing entries in the `puppet.conf` file. If you upgrade or reinstall using a different master hostname, Puppet applies the new value in `$confdir\puppet.conf`.

When you upgrade Windows, you must use the same master hostname that you specified when you installed.

For information on what settings are preserved during an upgrade, see installing [Install Windows agents](#) on page 154.

Errors when applying a manifest or doing a Puppet agent run

Check for the following issues if manifests cannot be applied or are not applied correctly on Windows nodes.

Path or file separators are incorrect

For Windows, path separators must use a semi-colon (;), while file separators must use forward or backslashes as appropriate to the attribute.

In most resource attributes, the Puppet language accepts either forward or backslashes as the file separator. However, some attributes absolutely require forward slashes, and some attributes absolutely require backslashes.

When backslashes are double-quoted(""), they must be escaped. When single-quoted('), they can be escaped. For example, these are valid file resources:

```
file { 'c:\path\to\file.txt': }
file { 'c:\\path\\to\\file.txt': }
file { "c:\\path\\to\\file.txt": }
```

But this is an invalid path, because \p, \t, and \f are interpreted as escape sequences:

```
file { "c:\path\to\file.txt": }
```

For more information, see the [language reference about backslashes on Windows](#).

Cases are inconsistent

Several resources are case-insensitive on Windows, like files, users, groups. However, these resources can be case sensitive in Puppet.

When establishing dependencies among resources, make sure to specify the case consistently. Otherwise, Puppet can't resolve dependencies correctly. For example, applying this manifest fails, because Puppet doesn't recognize that ALEX and alex are the same user:

```
file { 'c:\foo\bar':
  ensure => directory,
  owner  => 'ALEX'
}
user { 'alex':
  ensure => present
}
...
err: /Stage[main]/File[c:\foo\bar]: Could not evaluate: Could not find user
ALEX
```

Shell built-ins are not executed

Puppet doesn't support a shell provider on Windows, so executing shell built-ins directly fails.

Wrap the built-in in `cmd.exe`:

```
exec { 'cmd.exe /c echo foo':
  path => 'c:\windows\system32;c:\windows'
}
```

Tip: In the 32-bit versions of Puppet, you might encounter file system redirection, where `system32` is switched to `sysWoW64` automatically.

PowerShell scripts are not executed

By default, PowerShell enforces a restricted execution policy which prevents the execution of scripts.

Specify the appropriate execution policy in the PowerShell command, for example:

```
exec { 'test':
  command => 'powershell.exe -executionpolicy remotesigned -file C:\test.ps1',
  path     => $::path
}
```

Or use the Puppet supported PowerShell.

Services are referenced with display names instead of short names

Windows services support a short name and a display name, but Puppet uses only short names.

1. Verify that your Puppet manifests use short names, for example `wuauserv`, not `Automatic Updates`.

Error messages

Use this reference to troubleshoot error messages when using Windows with Puppet.

- Error: Could not connect via HTTPS to `https://forge.puppet.com` / Unable to verify the SSL certificate / The certificate may not be signed by a valid CA / The CA bundle included with OpenSSL may not be valid or up to date

This error occurs when you run the `puppet module` subcommand on newly provisioned Windows nodes. The Forge uses an SSL certificate signed by the GeoTrust Global CA certificate. Newly provisioned Windows nodes might not have that CA in their root CA store yet.

Download the "GeoTrust Global CA" certificate from GeoTrust's list of root certificates and manually install it by running `certutil -addstore Root GeoTrust_Global_CA.pem`.

- Service 'Puppet Agent' (`puppet`) failed to start. Verify that you have sufficient privileges to start system services.

This error occurs when installing Puppet on a UAC system from a non-elevated account. Although the installer displays the UAC prompt to install Puppet, it does not elevate privileges when trying to start the service.

Run from an elevated `cmd.exe` process when installing the MSI.

- Cannot run on Microsoft Windows without the `sys-admin`, `win32-process`, `win32-dir`, `win32-service` and `win32-taskscheduler` gems.

This error occurs if you attempt to run Windows without required gems.

Install specified gems: `gem install <GEM_NAME>`

- "

```
err: /Stage[main]//Scheduled_task[task_system]: Could not evaluate: The operation completed successfully.
```

"

This error occurs when using a the task scheduler gem earlier than version 0.2.1.

Update the task scheduling gem: `gem update win32-taskscheduler`

- err: /Stage[main]//Exec[C:/tmp/foo.exe]/returns: change from notrun to 0 failed: CreateProcess() failed: Access is denied.

This error occurs when requesting an executable from a remote Puppet master that cannot be executed.

Set the user/group executable bits appropriately on the master:

```
file { "C:/tmp/foo.exe":
  source => "puppet:///modules/foo/foo.exe",
}
```

```
exec { 'C:/tmp/foo.exe':
  logoutput => true
}
```

```
}
```

- `err: getaddrinfo: The storage control blocks were destroyed.`

This error occurs when the agent can't resolve a DNS name into an IP address or if the reverse DNS entry for the agent is wrong.

Verify that you can run `nslookup <dns>`. If this fails, there is a problem with the DNS settings on the Windows agent, for example, the primary dns suffix is not set. For more information, see [Microsoft's documentation on Naming Hosts and Domains](#).

- `err: Could not request certificate: The certificate retrieved from the master does not match the agent's private key.`

This error occurs if the agent's SSL directory is deleted after it retrieves a certificate from the master, or when running the agent in two different security contexts.

Elevate privileges using **Run as Administrator** when you select **Start Command Prompt with Puppet**.

- `err: Could not send report: SSL_connect returned=1 errno=0 state=SSLv3 read server certificate B: certificate verify failed. This is often because the time is out of sync on the server or client.`

This error occurs when Windows agents' time isn't synched. Windows agents that are part of an Active Directory domain automatically have their time synchronized with AD.

For agents that are not part of an AD domain, enable and add the Windows time service manually:

```
w32tm /register
net start w32time
w32tm /config /manualpeerlist:<ntpserver> /syncfromflags:manual /update
w32tm /resync
```

- `Error: Could not parse for environment production: Syntax error at '='; expected '}'`

This error occurs if `puppet apply -e` is used from the command line and the supplied command is surrounded with single quotes (`'`), which causes `cmd.exe` to interpret any `=>` in the command as a redirect.

Surround the command with double quotes (`"`) instead.

Logging and debugging

The Windows Event Log can be helpful when troubleshooting issues with Windows.

Enable Puppet to emit `--debug` and `--trace` messages to the Windows Event Log by stopping the Puppet service and restarting it:

```
c:\>sc stop puppet && sc start puppet --debug --trace
```

Note that this setting takes effect only until the next time the service is restarted, or until the system is rebooted.

For more information on logging, see [Configuring Puppet agent on Windows](#).