Red Hat Satellite 6.10

Puppet Guide

A guide to building your own Puppet module and importing it into Satellite 6

Red Hat Satellite 6.10 Puppet Guide

A guide to building your own Puppet module and importing it into Satellite 6

Red Hat Satellite Documentation Team

satellite-doc-list@redhat.com

Legal Notice

Abstract

Puppet is a system configuration tool used in Red Hat Satellite 6. This book runs through the creation of a basic Puppet Module and how to use this module in your Red Hat Satellite 6 infrastructure.

Table of Contents

CHAPTER 1. OVERVIEW

Puppet is a tool for applying and managing system configurations. Puppet collects system information, or facts, and uses this information to create a customized system configuration using a set of modules. These modules contain parameters, conditional arguments, actions, and templates. Puppet is used as either a local system command-line tool or in a client-server relationship where the server acts as the Puppet master and applies configuration to multiple client systems using a Puppet agent. This provides a way to automatically configure newly provisioned systems, either individually or simultaneously to create a specific infrastructure.

1.1. DEFINING THE PUPPET WORKFLOW

Puppet uses the following workflow to apply configuration to a system.

1. Collect facts about each system. These facts can include hardware, operating systems, package versions, and other information. The Puppet agent on each system collects this information and sends it to the Puppet master.

2. The Puppet master generates a custom configuration for each system and sends it to the Puppet agent. This custom configuration is called a catalog.

3. The Puppet agent applies the configuration to the system.

4. The Puppet agent sends a report back to the Puppet master that indicates the changes applied and if any changes were unsuccessful.

5. Third-party applications can collect these reports using Puppet's API.

1.2. USING PUPPET ON SATELLITE 6

Red Hat Satellite 6 uses Puppet in several ways:

- Red Hat Satellite 6 imports Puppet modules used to define the system configuration. This includes control over module versions and their environments.

- Red Hat Satellite 6 imports sets of parameters, also known as Puppet Smart Class parameters, from Puppet modules. Users can accept the default values from Puppet classes or provide their own at a global or system-specific level.

- Red Hat Satellite 6 triggers the execution of Puppet between the master and the respective agents on each system. Puppet runs can occur either:

  - Automatically, such as after the provisioning process completes or as a daemon that checks and manages the machine's configuration over its life cycle.

  - Manually, such as when the administrator needs to trigger an immediate Puppet run.

- Satellite 6 collects reports from Puppet after the configuration workflow completes. This helps with auditing and archiving system configuration over long term periods.

These functions provide an easy way for users to control system configuration aspects of the application life cycle using Puppet.

If required, to find out the version of Puppet used in your Satellite environment, see the Package Manifest.

1.3. PUPPET PERFORMANCE AND SCALABILITY ON SATELLITE 6

The performance of Puppet in Satellite 6 is influenced more by the Satellite and Capsule storage capacity, CPU performance, and available memory, than by application limitations. Therefore, testing of your hardware and configuration is the only way to be sure the performance is acceptable for your needs.

For information about recommended storage and system requirements, see Preparing your Environment for Installation in *Installing Satellite Server from a Connected Network* .

CHAPTER 2. BUILDING PUPPET MODULES FROM SCRATCH

This chapter explores how to build and test your own Puppet modules. This includes a basic tutorial on creating a Puppet module that deploys a simple web server configuration.

2.1. EXAMINING THE ANATOMY OF A PUPPET MODULE

Before creating our module, we need to understand the components that create a Puppet module.

Puppet Manifests

Do not confuse with Manifest (Subscription Manifest). Puppet Manifests are files that contain code to define a set of resources and their attributes. A resource is any configurable part of a system. Examples of resources include packages, services, files, users and groups, SELinux configuration, SSH key authentication, cron jobs, and more. A manifest defines each required resource using a set of key-value pairs for their attributes. For example:

```
package { 'httpd':
  ensure => installed,
}
```

This declaration checks if the **httpd** package is installed. If not, the manifest executes **yum** and installs it.

The Puppet Manifest used to start compiling is called 'main manifest' or 'site manifest'.

Manifests are located in the **manifest** directory of a module.

Puppet modules also use a **test** directory for test manifests. These manifests are used to test certain classes contained in your official manifests.

Static Files

Modules can contain static files that Puppet can copy to certain locations on your system. These locations, and other attributes such as permissions, are defined through **file** resource declarations in manifests.
Static files are located in the **files** directory of a module.

Templates

Sometimes configuration files require custom content. In this situation, users would create a template instead of a static file. Like static files, templates are defined in manifests and copied to locations on a system. The difference is that templates allow Ruby expressions to define customized content and variable input. For example, if you wanted to configure **httpd** with a customizable port then the template for the configuration file would include:

```
Listen <%= @httpd_port %>
```

The **httpd_port** variable in this case is defined in the manifest that references this template.

Templates are located in the **templates** directory of a module.

Plug-ins

Plug-ins allow for aspects that extend beyond the core functionality of Puppet. You can use plug-ins

to define custom facts, custom resources, or new functions. For example, a database administrator might need a resource type for PostgreSQL databases. This could help the database administrator populate PostgreSQL with a set of new databases after installing PostgreSQL. As a result, the database administrator need only create a Puppet Manifest that ensures PostgreSQL installs and the databases are created afterwards.

Plug-ins are located in the **lib** directory of a module. This includes a set of subdirectories depending on the plug-in type. For example:

- **/lib/facter** — Location for custom facts.

- **/lib/puppet/type** — Location for custom resource type definitions, which outline the key-value pairs for attributes.

- **/lib/puppet/provider** — Location for custom resource providers, which are used in conjunction with resource type definitions to control resources.

- **/lib/puppet/parser/functions** — Location for custom functions.

2.2. SETTING UP A PUPPET DEVELOPMENT SYSTEM

A Puppet development system is useful for creating and testing your own modules.

Procedure

Use this procedure to deploy a new Puppet development system.

1. Deploy a new Red Hat Enterprise Linux 7 system and register the system to the Red Hat CDN or your Satellite Server.

2. Enable the Red Hat Satellite Tools 6.10 repository for Puppet 5:

   ```
   # subscription-manager repos \
   --enable=rhel-7-server-satellite-tools-6.10-rpms
   ```

3. Install the Puppet agent:

   ```
   # yum install puppet
   ```

4. Install the Puppet Development Kit:

   ```
   # yum install pdk
   ```

2.3. GENERATING A NEW MODULE BOILERPLATE

The first step in creating a new module is to change to the Puppet module directory and create a basic module structure. Either create this structure manually or use the Puppet Development Kit to create a boilerplate for your module:

```
# cd /etc/puppetlabs/code/modules
# pdk new module user_name-module_name
```

An interactive wizard appears and guides you through populating the module's **metadata.json** file with metadata. For more information, see Create a module in *Puppet Development Kit* documentation.

When the module generation process completes, the new modules contains some basic files, including a **manifests** directory. This directory already contains a manifest file called **init.pp**, which is the module's main manifest file. View the file to see the empty class declaration for the module:

```
class mymodule {

}
```

The module also contains an **examples** directory containing a manifest also named **init.pp**. This test manifest contains a reference to the **mymodule** class within **manifests/init.pp**:

```
include::mymodule
```

Puppet will use this test manifest to test our module.

We are now ready to add our system configuration to our module.

2.4. INSTALLING A HTTP SERVER

Our Puppet module will install the packages necessary to run an HTTP server. This requires a resource definition that defines configurations for the **httpd** package.

In the module's **manifests** directory, create a new manifest file called **httpd.pp**:

```
# touch mymodule/manifests/httpd.pp
```

This manifest will contain all HTTP configuration for our module. For organizational purposes, we will keep this manifest separate from the **init.pp** manifest.

Add the following content to the new **httpd.pp** manifest:

```
class mymodule::httpd {
  package { 'httpd':
    ensure => installed,
  }
}
```

This code defines a subclass of **mymodule** called **httpd**, then defines a package resource declaration for the **httpd** package. The **ensure ⇒ installed** attribute tells Puppet to check if the package is installed. If it is not installed, Puppet executes **yum** to install it.

We also need to include this subclass in our main manifest file. Edit the **init.pp** manifest:

```
class mymodule {
  include mymodule::httpd
}
```

It is now time to test the module. Run the following command:

```
# puppet apply mymodule/examples/init.pp --noop
...
Notice: /Stage[main]/Mymodule::Httpd/Package[httpd]/ensure: current_value absent, should be
```

```
present (noop)
...
```

This output notice message is the result of the **ensure ⇒ installed** attribute. The **current_value absent** means that Puppet has detected the **httpd** package is not installed. Without the **--noop** option, Puppet would install the **httpd** package.

The **puppet apply** command applies the configuration in the manifest to your system. We use the test init.pp manifest, which refers to the main init.pp manifest. The **--noop** performs a dry run of the configuration, which shows only the output but does not actually apply the configuration.

2.5. RUNNING THE HTTP SERVER

After installing the **httpd** package, we start the service using another resource declaration: **service**.

Edit the **httpd.pp** manifest and add the highlighted lines:

```
class mymodule::httpd {
  package { 'httpd':
    ensure => installed,
  }
  service { 'httpd':
    ensure => running,
    enable => true,
    require => Package["httpd"],
  }
}
```

This achieves a couple of things:

- The **ensure ⇒ running** attribute checks if the service is running. If not, Puppet starts it.

- The **enable ⇒ true** attribute sets the service to run when the system boots.

- The **require ⇒ Package["httpd"]** attribute defines an ordering relationship between one resource declaration and another. In this case, it ensures the **httpd** service starts after the **httpd** package installs. This creates a dependency between the service and its respective package.

Run the **puppet apply** command again to test the changes to our module:

```
# puppet apply mymodule/tests/init.pp --noop
...
Notice: /Stage[main]/Mymodule::Httpd/Service[httpd]/ensure: current_value stopped, should be running (noop)
...
```

This output notice message is the result of our new resource definition for the **httpd** service.

2.6. CONFIGURING THE HTTP SERVER

The HTTP Server is now installed and enabled. The next step is to provide some configuration. The HTTP server already provides some default configuration in **/etc/httpd/conf/httpd.conf**, which provides a web server on port 80. We will add some additional configuration to provide an additional web server on a user-specified port.

We use a template file to store our configuration content because the user-defined port requires variable input. In our module, create a directory called **templates** and add a file called **myserver.conf.erb** in the new directory. Add the following contents to the file:

```
Listen <%= @httpd_port %>
NameVirtualHost *:<%= @httpd_port %>
<VirtualHost *:<%= @httpd_port %>>
  DocumentRoot /var/www/myserver/
  ServerName <%= @fqdn %>
  <Directory "/var/www/myserver/">
    Options All Indexes FollowSymLinks
    Order allow,deny
    Allow from all
  </Directory>
</VirtualHost>
```

This template follows the standard syntax for Apache web server configuration. The only difference is the inclusion of Ruby escape characters to inject variables from our module. For example, **httpd_port**, which we use to specify the web server port.

Notice also the inclusion of **fqdn**, which is a variable that stores the fully qualified domain name of the system. This is known as a system fact. System facts are collected from each system prior to generating each respective system's Puppet catalog. Puppet uses the **facter** command to gather these system facts and you can also run **facter** to view a list of these facts.

Edit the **httpd.pp** manifest and add the highlighted lines:

```
class mymodule::httpd {
  package { 'httpd':
    ensure => installed,
  }
  service { 'httpd':
    ensure => running,
    enable => true,
    require => Package["httpd"],
  }
  file {'/etc/httpd/conf.d/myserver.conf':
  notify => Service["httpd"],
    ensure => file,
    require => Package["httpd"],
    content => template("mymodule/myserver.conf.erb"),
  }
  file { "/var/www/myserver":
    ensure => "directory",
  }
}
```

This achieves the following:

- We add a file resource declaration for the server configuration file **/etc/httpd/conf.d/myserver.conf**.

- We add a relationship between the configuration file and the **httpd** service using the **notify ⇒ Service["httpd"]** attribute. This checks our configuration file for any changes. If the file has changed, Puppet restarts the service.

- We check the **httpd** package is installed before adding this file.

- The **content** for this **/etc/httpd/conf.d/myserver.conf** file is the **myserver.conf.erb** template we created earlier.

- We add a second file resource declaration. This one creates a directory **/var/www/myserver/** for our web server.

We also need to include the **httpd_port** parameter in our main manifest file. Edit the init.pp manifest and add the following text shown in bold:

```
class mymodule (
  $httpd_port = 8120
) {
  include mymodule::httpd
}
```

This sets the **httpd_port** parameter to a default value of 8120. You can override this value with the Satellite Server.

Run the **puppet apply** command again to test the changes to our module:

```
# puppet apply mymodule/tests/init.pp --noop
...
Notice: /Stage[main]/Mymodule::Httpd/File[/var/www/myserver]/ensure: current_value absent, should be directory (noop)
...
Notice: /Stage[main]/Mymodule::Httpd/File[/etc/httpd/conf.d/myserver.conf]/ensure: current_value absent, should be file (noop)
...
```

These output notice messages show the creation of the configuration file and our web server directory.

2.7. CONFIGURING THE FIREWALL

The web server requires an open port so people can access the pages hosted on our web server. The open problem is that different versions of Red Hat Enterprise Linux uses different methods for controlling the firewall. For Red Hat Enterprise Linux 6 and below, we use **iptables**. For Red Hat Enterprise Linux 7, we use **firewalld**.

This decision is something Puppet handles using conditional logic and system facts. For this step, we add a statement to check the operating system and run the appropriate firewall commands.

Add the following code inside your **mymodule::httpd** class:

```
if versioncmp($::operatingsystemmajrelease, '6') <= 0 {
  exec { 'iptables':
    command => "iptables -I INPUT 1 -p tcp -m multiport --ports ${httpd_port} -m comment --comment 'Custom HTTP Web Host' -j ACCEPT &amp;&amp; iptables-save > /etc/sysconfig/iptables",
    path => "/sbin",
    refreshonly => true,
    subscribe => Package['httpd'],
  }
  service { 'iptables':
    ensure => running,
```

```
      enable => true,
      hasrestart => true,
      subscribe => Exec['iptables'],
    }
  }
  elsif $operatingsystemmajrelease == 7 {
    exec { 'firewall-cmd':
      command => "firewall-cmd --zone=public --add-port=${httpd_port}/tcp --permanent",
      path => "/usr/bin/",
      refreshonly => true,
      subscribe => Package['httpd'],
    }
    service { 'firewalld':
      ensure => running,
      enable => true,
      hasrestart => true,
      subscribe => Exec['firewall-cmd'],
    }
  }
```

This code performs the following:

- Use the **operatingsystemmajrelease** fact to determine whether the operating system is Red Hat Enterprise Linux 6 or 7.

- If using Red Hat Enterprise Linux 6, declare an executable (**exec**) resource that runs **iptables** and **iptables-save** to add a permanent firewall rule. The **httpd_port** variable is used inline to define the port to open. After the **exec** resource completes, we trigger a refresh of the **iptables** service. To achieve this, we define a service resource that includes the **subscribe** attribute. This attribute checks if there are any changes to another resource and, if so, performs a refresh. In this case, it checks the **iptables** executable resource.

- If using Red Hat Enterprise Linux 7, declare a similar executable resource that runs **firewall-cmd** to add a permanent firewall rule. The **httpd_port** variable is also used inline to define the port to open. After the **exec** resource completes, we trigger a refresh of the **firewalld** service but with a **subscribe** attribute pointing to the **firewall-cmd** executable resource.

- The code for both firewall executable resources contains **refreshonly ⇒ true** and **subscribe ⇒ Package['httpd']** attributes. This ensures the firewall commands only run after the **httpd** installs. Without these attributes, subsequent runs will add multiple instances of the same firewall rule.

Run the **puppet apply** command again to test the changes to our module. The following example is a test of Red Hat Enterprise Linux 6:

```
# puppet apply mymodule/tests/init.pp --noop
...
Notice: /Stage[main]/Mymodule::Httpd/Exec[iptables]/returns: current_value notrun, should be 0
(noop)
...
Notice: /Stage[main]/Mymodule::Httpd/Service[iptables]: Would have triggered 'refresh' from 1 events
...
```

These output notice messages show the execution of the firewall rule creation and the subsequent service refresh as a result of the **subscribe** attribute.
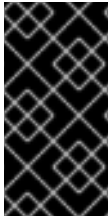
IMPORTANT

This configuration serves only as an example of using conditional statements. If you aim to manage multiple firewall rules for your system in the future, it is recommended to create a custom resource for firewalls. It is inadvisable to use executable resources to constantly chain many Bash commands.

2.8. CONFIGURING SELINUX

SELinux restricts non-standard access to the HTTP server by default. If we define a custom port, we need to add configuration that allows SELinux to grant access.

Puppet contains resource types to manage some SELinux functions, such as Booleans and modules. However, we need to execute the **semanage** command to manage port settings. This tool is a part of the **policycoreutils-python** package, which is not installed on Red Hat Enterprise Linux systems by default.

Add the following code inside your **mymodule::httpd** class:

```
exec { 'semanage-port':
  command => "semanage port -a -t http_port_t -p tcp ${httpd_port}",
  path => "/usr/sbin",
  require => Package['policycoreutils-python'],
  before => Service['httpd'],
  subscribe => Package['httpd'],
  refreshonly => true,
}
package { 'policycoreutils-python':
  ensure => installed,
}
```

This code performs the following:

- The **require ⇒ Package['policycoreutils-python']** attribute makes sure the **policycoreutils-python** is installed prior to executing the command.

- Puppet executes **semanage**, using **httpd_port** as a variable, to add the custom port to the list of TCP ports Apache is allowed to listen on.

- The **before ⇒ Service ['httpd']** makes sure to execute this command before the **httpd** service starts. If **httpd** starts before the SELinux command, SELinux denies access to the port and the service fails to start.

- The code for the SELinux executable resource contains **refreshonly ⇒ true** and **subscribe ⇒ Package['httpd']** attributes. This ensures the SELinux commands only run after the **httpd** installs. Without these attributes, subsequent runs result in failure. This is because SELinux detects the port is already enabled and reports an error.

Run the **puppet apply** command again to test the changes to our module.

```
# puppet apply mymodule/tests/init.pp --noop
...
Notice: /Stage[main]/Mymodule::Httpd/Package[policycoreutils-python]/ensure: current_value absent, should be present (noop)
...
```

```
Notice: /Stage[main]/Mymodule::Httpd/Exec[semanage-port]/returns: current_value notrun, should be
0 (noop)
...
Notice: /Stage[main]/Mymodule::Httpd/Service[httpd]/ensure: current_value stopped, should be
running (noop)
...
```

Puppet installs **policycoreutils-python** first, then configures port access before starting the **httpd** service.

2.9. COPYING A HTML FILE TO THE WEB HOST

The HTTP server configuration is now complete. This provides a platform for installing a web-based application, which Puppet can also configure. For this example, however, we will only copy over a simple index web page to our web server.

Create a file named **index.html** in the **files** directory. Add the following content to this file:

```
<html>
  <head>
    <title>Congratulations</title>
  <head>
  <body>
    <h1>Congratulations</h1>
    <p>Your puppet module has correctly applied your configuration.</p>
  </body>
</html>
```

Create a manifest named **app.pp** in the **manifests** directory. Add the following content to this file:

```
class mymodule::app {
  file { "/var/www/myserver/index.html":
    ensure => file,
    mode   => '755',
    owner  => root,
    group  => root,
    source => "puppet:///modules/mymodule/index.html",
    require => Class["mymodule::httpd"],
  }
}
```

This new class contains a single resource declaration. This declaration copies a file from the module's file directory from the Puppet Server to the system and sets its permissions. Additionally, the **require** attribute ensures the **mymodule::httpd** class completes the configuration successfully before we apply **mymodule::app**.

Finally, include this new manifest in our main **init.pp** manifest:

```
class mymodule (
  $httpd_port = 8120
) {
  include mymodule::httpd
  include mymodule::app
}
```

Run the **puppet apply** command again to test the changes to our module. The output should resemble the following:

```
# puppet apply mymodule/tests/init.pp --noop
....
Notice: /Stage[main]/Mymodule::App/File[/var/www/myserver/index.html]/ensure: current_value absent, should be file (noop)
...
```

This output notice message shows that the **index.html** file would be copied to the web server.

2.10. FINALIZING THE MODULE

Our module is ready for use. To export the module into an archive for Red Hat Satellite 6 to use, enter the following command:

```
# puppet module build mymodule
```

This creates an archive file at **mymodule/pkg/mymodule-0.1.0.tar.gz**, which contains the contents of our **mymodule** directory. We upload this module to our Red Hat Satellite 6 Server to provision our own HTTP server.

If any changes are required, edit the files within the modules directories and rebuild the module using the **puppet module build** command. The changes will only be reflected in Satellite if the module version is increased. To increase the version number, edit the **/etc/puppetlabs/code/modules/mymodule/metadata.json** file and then rebuild the module. Upload and publish the new version in Satellite Server.

CHAPTER 3. ADDING PUPPET MODULES TO RED HAT SATELLITE

You can import Puppet classes from modules on your Puppet server and classify Hosts and Puppet Environments, to which the classes belong. You can also customize the parameters of assigned classes. This allows you to control what parameters are passed to the puppet-agent on the Host through the external node classifier (ENC).

3.1. PUPPET ENVIRONMENTS

A Puppet environment is defined as an isolated set of Puppet agent nodes that can be associated with a specific set of Puppet modules. In Satellite context we can think of a Puppet environment as a set of Hosts running the Puppet agent that are associated with a specific set of Puppet modules. For example, nodes that are associated with environment *Production* only have access to the modules that are in environment *Production*.

Puppet environments are used to separate Puppet modules from different types of Host. A typical use is to enable changes to a module to be tested in one environment before being pushed to another. A Puppet module can contain facts and functions as well as one or more Puppet classes that you can assign to Hosts. Modules are part of an environment, Puppet classes are only part of an environment because they are part of a module.

In Red Hat Satellite, a Puppet environment will be created automatically for a Content View if the CV has a Puppet module. The automatically created Puppet environment name includes the organization label, the life-cycle environment, the Content View name, and the Content View ID. For example, **KT_Example_Org_Library_RHEL6Server_3**.

Creating Puppet Environments within Red Hat Satellite

1. Navigate to **Configure → Environments** and select **New Puppet Environment**.

2. Give the new environment a name and click **Submit** to save the changes.

Note that if the environment does not exist on the Puppet master and you subsequently run an import, Satellite will prompt for the environment to be deleted.

> **NOTE**
>
> Puppet fails to retrieve the Puppet CA certificate while registering a host with a host group associated with a Puppet environment created inside a **Production** environment. To create a suitable Puppet environment to be associated with a host group, follow these steps:
>
> 1. Manually create a directory and change the owner:
>
>    ```
>    # mkdir /etc/puppetlabs/code/environments/example_environment
>    # chown apache /etc/puppetlabs/code/environments/example_environment
>    ```
>
> 2. Navigate to **Configure → Environments** and click **Import environment from**. The button name will include the FQDN of the internal or external Capsule.
>
> 3. Choose the created directory and click **Update**.

Importing Puppet Environments to Red Hat Satellite

Satellite can detect all the environments and Puppet modules contained on a Puppet master, and import them automatically. To do this, go to **Configure** > **Environments** and click the **Import from** button. The button name will include the FQDN of the internal or external Capsule. Satellite will scan the Puppet master via the Capsule, and display a list of the detected changes. Select the changes you want to apply and select **Update** to apply the changes.

Note that the Capsule will only detect environments that contain one or more Puppet classes, so ensure that at least one Puppet module containing a class has been deployed to the Puppet master.

Assigning a Puppet Environment to a Host

Navigate to **Hosts** > **All Hosts**, select a host by name from the **Hosts** list and then select **Edit**. On the **Host** tab you can select a **Puppet Environment** for the host. Selecting an environment will filter the classes on the **Puppet Classes** tab to help distinguish the classes in the selected environment.

You can mass-assign an environment to a group of hosts. Select the check boxes of the required hosts in the **Hosts** list, and then select **Change Environment** from the **Select Action** drop-down menu at the top of the page.

Assigning Puppet Environments to Host Groups

When creating a **Host Group**, the environment field is pre-populated by the environment that was automatically created by the associated Content View, if any.

By default, a user creating a new host and selecting the **Host Group** will automatically have the environment pre-selected. The user can change the environment of the new host.

3.2. PARAMETERS

Red Hat Satellite parameters define key-value pairs to use when provisioning hosts. These are similar to Puppet's concept of a default scope parameter. You can define parameters when setting up a host with Puppet.

Types of Parameters

Red Hat Satellite has two types of parameters:

Simple Parameters

> String parameters that define a relationship between a key and value pair. They cannot be overridden by user configuration, but they are overridden according to Satellite's parameter hierarchy. The following parameters are simple parameters in Red Hat Satellite: Global, organization-level, location-level, domain-level, subnet level, operating system level, host group, and host parameters.

Smart Class Parameters

> Complex parameters that define a value for a key but allow conditional arguments, validation, and overrides for specific object types. Smart class parameters enable a Puppet class to get external data. They are used in Puppet classes, and are called parameterized classes in Puppet terminology. The hierarchy for these parameters can be configured in the web UI.

The following parameter hierarchy applies for simple parameters:

Global Parameters

> Default parameters that apply to every host in Satellite. Configured in **Configure > Global parameters**. To set global parameters using Hammer CLI, enter the following command:

```
# hammer global-parameter set --name parameter_name --value parameter_value
```

Organization-level parameters

Parameters that affect all hosts in a given organization. Organization-level parameters override Global parameters. Configured in **Administer > Organizations > Edit > Parameters**To set organization-level parameters using Hammer CLI, enter the following command:

```
# hammer organization set-parameter --organization "Your Organization" \
--name parameter_name --value parameter_value
```

Location-level parameters

Parameters that affect all hosts in a given location. Location-level parameters override Organization-level and Global parameters. Configured in **Administer > Locations > Edit > Parameters**To set location-level parameters using Hammer CLI, enter the following command:

```
# hammer location set-parameter --location "Your_Location" \
--name parameter_name --value parameter_value
```

Domain Parameters

Parameters that affect all hosts in a given domain. Domain parameters override Location-level and higher parameters. Configured in **Infrastructure > Domains > [choose_a_domain] > Parameters**To set domain parameters using Hammer CLI, enter the following command:

```
# hammer domain set-parameter --domain domain_name \
--name parameter_name --value parameter_value
```

Subnet Parameters

Parameters that affect all hosts that have a primary interface in a given Subnet. Subnet parameters override Host Group system level and higher parameters. Configured in **Infrastructure > Subnets > [choose_a_subnet] > Parameters**. To set subnet parameters using Hammer CLI, enter the following command:

```
# hammer subnet set-parameter --subnet subnet_name \
--name parameter_name --value parameter_value
```

Operating System Level Parameters

Parameters that affect all hosts with a given operating system. Operating system level parameters override Domain and higher parameters. Configured in **Hosts > Operating systems > [choose_an_operating_system] > Parameters**. To set operating system parameters using Hammer CLI, enter the following command:

```
# hammer os set-parameter --operatingsystem os_name \
--name parameter_name --value parameter_value
```

Host Group Parameters

Parameters that affect all hosts in a given Host Group. Host Group parameters override Operating system level and higher parameters. Configured in **Configure > Host Groups > [choose_a_host_group] > Parameters**. To set host group parameters using Hammer CLI, enter the following command:

```
# hammer hostgroup set-parameter --hostgroup hostgroup_name \
--name parameter_name --value parameter_value
```

Host Parameters

Parameters that affect a specific host. All previously inherited parameters are visible on the Parameters subtab and can be overridden. Configured in **Hosts > All hosts > Edit > Parameters**To set host parameters using Hammer CLI, enter the following command:

```
# hammer host set-parameter --host host_name \
--name parameter_name --value parameter_value
```

Using Parameters with Puppet Classes

Red Hat Satellite has two ways to supply values to a Puppet master for a host to use with a Puppet class:

Smart Variables

A tool to provide global parameters to the Puppet master, in key–value form, for classes that do not have Smart class parameters. They enable overriding parameter values in a Puppet manifest. They are intended for use when a class does not have Smart class parameters or in special cases when a global parameter is desired. They can have multiple possible values, all depending on hierarchical context or various conditions a user can apply. They existed before Puppet had parameterized classes and today are kept either for backward compatibility or for the use of global parameters where you want validations, to use only with specific Puppet classes, and for types other than string (because otherwise you could just use the simple parameters).

Parameterized Classes

Puppet classes containing Smart class parameters. The classes are imported from the Puppet master and the name of the parameter, for example **$::name** (preferred) or **$name**, is defined by the person who wrote the class and cannot be changed. They enable you to decide the value of the variable for a specific class rather than globally.

Configured parameters are included in each host's corresponding YAML file and sent to the Puppet master. The YAML file can be viewed in the web UI on the page for a specific host. You should not manually change the **/etc/foreman/settings.yaml** configuration file because they are overwritten the next time you run the **satellite-installer** command.

> **IMPORTANT**
>
> Parameterized class support is enabled by default in Satellite. To confirm it is enabled, navigate to **Administer > Settings**, select the **Puppet** tab, and ensure the **Parameterized Classes in ENC** is set to **True**.

3.3. CONFIGURING SMART CLASS PARAMETERS

The following procedure configures parameters within a class. Classes that contain parameters are referred to as parameterized classes.

Smart class parameters are common for all organizations. Any user with the **edit_external_parameters** permission can edit these parameters. If you want to restrict permissions to edit smart class parameters, see the KCS solution Restrict permissions to edit puppet classes and their smart class parameters which are common between multiple organizations.

Configuring Smart Class Parameters

1. Click **Configure > Puppet Classes**

2. Select a class from the list that has parameters as indicated in the **Parameters** column.

3. Click the **Smart Class Parameter** tab. This displays a new screen. The left section contains a list of possible parameters the class supports. The right section contains the configuration options for the parameter selected.

4. Select a parameter from the left-hand list.

5. Edit the **Description** text box to add any plain text notes.

6. Select **Override** to allow Satellite control over this variable. If the check box is not selected, Satellite does not pass the new variable to Puppet.

7. Select the **Parameter type** of data to pass. This is most commonly a string, but other data types are supported.

8. Enter a **Default Value** for the parameter to be sent to the Puppet master if no host match occurs.

9. Optional: Select **Omit** to **not** send a value to the Puppet master unless an override match occurs.

10. Optional: Select **Hidden value** if the field contains data you do not want to be displayed while you are working.

11. Use the **Optional Input Validator** section to restrict the allowed values for the parameter. Choose a **Validator type** (either a **list** of comma-separated values or a regular expression, **regexp**) and input the allowed values or regular expression code in the **Validator rule** field.

12. The **Prioritize Attribute Order** section will appear if the **Override** option is selected. This provides options for overriding values for specific hosts based upon conditional arguments. The attribute type and its value is known as a matcher.

13. In the **Order** field, set the order of precedence in which the host attributes or Facts are to be evaluated against the matchers by arranging the entries in the list. You can add an attribute that is present in Facter and cannot be confused with a host attribute to the default list.
    If you want to create a logical **AND** condition between matchers and use multiple attributes as a matcher key, arrange them on one line as a comma-separated list in the following format:

    ```
    location,environment
    ```

    NOTE

    Until BZ#1772381 is resolved, note that the **Order** tooltip displays incorrect example configuration for using multiple attributes as a matcher key.

14. Click **Add Matcher** to add a conditional argument. The attributes to match against correspond to the entries in the **Order** list. If no matcher is configured then only the default value can be used for the override feature.

15. From the **Attribute type** list, select the attribute.

16. In the field next to the attributes, enter the attribute string.

17. In the **Value** field, enter the value that you want.
Dynamic data is possible by using parameters and Puppet Facts in the **Value** field in Embedded Ruby (ERB) template syntax. For example, to use a Puppet Fact as part of the value:

> <%= @host.facts['network_eth0'] %>

To list available Puppet Facts navigate to **Monitor > Facts**.

For more information on ERB syntax see Template Writing Reference in *Managing Hosts*.

18. Click **Submit**.

Setting a single attribute as a matcher key

You can set a single attribute as a matcher key.

For example, to supply the **test** value of a parameter to the Puppet master for any host within the **Default_Location** location, complete the following steps:

1. To the **Order** field, append **location**.

2. Click **Add Matcher** and from the **Attribute type** list, select **location**.

3. In the field next to the attributes, enter **Default_Location**.

4. In the **Value** field, enter **test**.

5. Click **Submit**.

Setting multiple attributes as a matcher key

You can set multiple attributes as a matcher.

For example, to supply the test value of a parameter to the Puppet master for any host within the **Default_Location** location and the **development** environment, complete the following steps:

1. To the **Order** field, append **location,environment**.

2. Click **Add Matcher** and from the **Attribute type** list, select **location,environment**.

3. In the field next to the attributes, enter **Default_Location,development**.

4. In the **Value** field, enter **test**.

5. Click **Submit**.

Precedence for Matching Overview

Satellite uses the following precedence when matching hosts:

1. Satellite searches the matcher in host attributes.

2. If there is no match in host attributes, Satellite searches the matcher in host parameters, which are inherited according to the parameter hierarchy.

3. If there is no match in host parameters, Satellite searches the matcher in host Facts.

3.4. APPLYING FACTS USING A PARENT HOST GROUP

You can use a parent host group to apply facts to members of multiple host groups.

Prerequisites

- You have host groups with hosts assigned.

- You have a parent host group with host groups assigned.

Applying facts to members of multiple host groups

1. In the Satellite web UI, navigate to **Configure** > **Classes**.

2. Select the Puppet class you want to use.

3. Click the **Smart Class Parameter** tab.

4. Select a parameter from the left-hand list you want to override.

5. Optional: Edit the **Description** field to add any plain text notes.

6. Select **Override** to grant Satellite control over this variable.

7. Select the **Parameter type** of data to pass. This is most commonly a string, but other data types are supported.

8. Optional: Enter a **Default Value** for the parameter to send to the Puppet master if no host match occurs.

9. Optional: Select **Omit** not to send a value to the Puppet master unless an override match occurs.

10. Optional: Select **Hidden Value** if the field contains data you do not want to be displayed while you are working.

11. Optional: Use the **Optional Input Validator** section to restrict the allowed values for the parameter. Select a **Validator Type**, which is either a list of comma-separated values or a regular expression, **regexp**, and add the allowed values or regular expression code in the **Validator rule** field.

12. In the **Prioritize Attribute Order** area, set the **Order** of precedence in which to evaluate the host attributes or Facts against the matchers by arranging the entries in the list. You can add to the default list. To create a logical **AND** condition between matchers, arrange the matchers on one line as a comma-separated list.

13. In the **Specify Matchers** area, click **Add Matcher** to add a conditional argument.

14. Set the **Attribute type** to **hostgroup**.

15. After the **=** sign, enter the host group that you want to match against. In this example, the parent host group.

16. In the **Value** field, enter the value that you want to send to the Content Hosts that belong to the parent host group.

17. Click **Submit**.

If the parent host group is part of a hierarchy of host groups, enter all the parents as follows for the matcher value: ***top_host_group*/*intermediate_host_group*/*parent_host_group***.

3.5. USING MULTIPLE CUSTOM FACTS

The following is an example of creating and using multiple custom facts and their values in a Puppet Smart class parameter matcher.

Prerequisites

- You have imported Puppet environments and have a Puppet Module with smart classes in Satellite.

- You have a client provisioned and registered to Satellite.

- For a client not provisioned by Satellite, check that the client is configured as described in Section 6.2, "Applying Configuration to Existing Clients" .

    1. On the client, create two or more custom facts and assign values to them. For example:

       ```
       # vi /etc/facter/facts.d/my_custom_facts
       #! /bin/bash
       echo example_fact1=myfact1
       echo example_fact2=myfact2
       ```

    2. On the client, configure the file permissions:

       ```
       # chmod a+x /etc/facter/facts.d/my_custom_facts
       ```

    3. On the client, check the facts and their respective values:

       ```
       # facter | grep example
        example_fact1 => myfact1
        example_fact2 => myfact2
       ```

    4. In the Satellite web UI:

       a. Navigate to **Configure** > **Classes** and select the Puppet class you want to configure.

       b. Click the **Smart Class Parameter** tab and select the parameter you want to override.

       c. In the **Default Behavior** area, select the **Override** check box.

       d. In the **Prioritize Attribute Order** area, in the **Order** field, add the example facts to the end of the list. To create a logical AND condition between the two facts, add them as a comma separated list **example_fact1,example_fact2**.

       e. Select **Add Matcher**.

       f. From the **Attribute type** menu, select **example_fact1,example_fact2** and in the box after the **=** sign, enter **myfact1,myfact2**.

       g. In the **Value** field, enter the value you want to send to the Content Host when a match is made for the two attributes and their values.

h. Click **Submit**.

5. On the client, send the facts from the client to the Puppet master:

```
# puppet agent --test
```

6. In the Satellite web UI:

    a. Navigate to **Hosts** > **Content Hosts** and select the name of the Content Host.

    b. Click **YAML** and locate the **classes** section. Check that the parameter has the value you want.

3.6. CONFIGURING SMART VARIABLES

The following procedure configures Smart Variables to override a value in a Puppet class.

Configuring Smart Variables

1. Click **Configure > Puppet Classes**

2. Select a class from the list.

3. Click the **Smart Variables** tab. This displays a new screen. The left section contains a list of possible parameters the class supports. The right section contains the configuration options for the parameter selected. Click the **Add Variable** to add a new parameter. Otherwise, select a parameter from the left-hand list.

4. Type a name for the parameter in the **Key** field.

5. Edit the **Description** text box to add any plain text notes.

6. Select the **Parameter Type** of data to pass. This is most commonly a string, but other data types are supported.

7. Enter a **Default Value** for the parameter to be sent to the Puppet master if no host match occurs.

8. Optional: Select **Hidden Value** if the field contains data you do not want to be displayed while you are working.

9. Use the **Optional Input Validator** section to restrict the allowed values for the parameter. Choose a **Validator Type** (either a **list** of comma separated values or a regular expression, **regexp**) and input the allowed values or regular expression code in the **Validator rule** field.

10. The **Prioritize Attribute Order** section provides options for overriding values for specific hosts based upon conditional arguments. The attribute type and its value is known as a **matcher**.

    a. Set the **Order** of precedence in which the host attributes or Facts are to be evaluated against the matchers by arranging the entries in the list. You can add to the default list. To create a logical AND condition between matchers, arrange them on one line as a comma separated list.

    b. Click **Add Matcher** to add a conditional argument. The attributes to match against should correspond to the entries in the **Order** list. If no matcher is configured then only the default value can be used for the override feature.

For example, if the desired value of the parameter to be supplied to the Puppet master is **test** for any host with a fully qualified domain name of **server1.example.com**, then specify the matcher as **fqdn=server1.example.com** and the **Value** as **test**.

The precedence for matching is as follows:

i. If the matcher is a host attribute, use that.

ii. If there are no attributes with that name, look for a matching host parameter (which is inherited according to the parameter hierarchy).

iii. If there is still no match, check the host Facts.
It is recommended to use an attribute that is present in Facter and cannot be confused with a host attribute. Host attributes can be either host parameters or associations to the host, such as host group, domain, and organization. The matcher must only be something the host has one of, for example config group cannot be used because the host can have many config groups but a host only has one location so location is a valid matcher.

Dynamic data is possible by using parameters and Puppet Facts in the **Value** field in **Embedded Ruby** (ERB) template syntax. For example, to use a Puppet Fact as part of the value:

```
<%= @host.facts['network_eth0'] %>
```

To list available Puppet Facts navigate to **Monitor > Facts**.

11. Click **Submit** to save your changes.

For more information on ERB syntax see Template Writing Reference in *Managing Hosts*.

3.7. IMPORTING PARAMETERIZED CLASSES FROM A PUPPET MASTER

The following procedure imports parameterized classes from your Puppet master.

> **NOTE**
>
> The import of parameterized classes happens automatically if your Puppet modules are managed via a Product and a Content View.

Importing Parameterized Classes

1. In the Satellite web UI, select from the context menu **Any Organization** and **Any Location**.

2. Click **Configure > Puppet Classes**.

3. Click **Import from** *Host Name* to import parameterized classes from your Puppet master.

4. The **Puppet Classes** page displays with the new classes listed.

3.8. USING THE SMART VARIABLE TOOL

Smart Variables are a tool for providing global parameters to the Puppet master for use with Puppet classes that do not contain Smart Class Parameters. The same Smart Matcher rules are used for both Smart Variables and Smart Class Parameters.
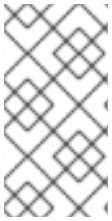
> **NOTE**
>
> The Smart Variables tool was introduced as an interim measure before Puppet modules supported Smart Class Parameters. If you do not have a specific reason to use Smart Variables, Red Hat recommends that you use Smart Class Parameters instead as described in Section 3.3, "Configuring Smart Class Parameters".

Before Smart Class Parameters were introduced, users who wanted to override a parameter where asked to rewrite their Puppet code to use a global parameter. For example:

```
class example1 {
  file { '/tmp/foo': content => $global_var }
}
```

For the above example, **$global_var** is set in the Smart Variables section of the web UI and the value is associated with the "example1" class. Although it is recommend to precede global variables with **::** to restrict Puppet to search the global scope, their absence does not mean a variable is not a global variable.

With the introduction of Smart Class Parameters, the following form could be used:

```
class example2($var="default") {
  file { '/tmp/foo': content => $var }
}
```

For the above example, **$var** is set in the Smart Class Parameters section of the web UI and the value is associated with the "example2" class. If you see a variable defined in the class header, as in the above **class example2($var="default")**, then you can be sure that **$var** is a class parameter and you should use the Smart Class Parameter function to override the variable.

As demonstrated, Smart Variables require custom-designed modules using global-namespace parameters, rather than standard modules from the Puppet community, and the result is the same: text placed in '/tmp/foo' in the examples above. Consequently, there is no longer a reason to use Smart Variables except to support legacy modules.

Although Smart Variables are global variables, they are associated with a Puppet class and will only be sent to a host that has that specific Puppet class assigned in Satellite. You can create a Smart Variable with any name, no validation is done in Satellite, but unless the associated Puppet module applied has a matching variable in its code, the Smart Variable will not be used.

Satellite adds the variable you create in Satellite to the Host YAML file. This file can be viewed in the web UI by navigating to **Hosts** > **All Hosts**, selecting the name of the host, and then clicking on the **YAML** button. Satellite sends the Host YAML file to the *external node classifier* (ENC), a function included with the Puppet master. When the Puppet master queries the ENC about a host, the ENC returns a YAML document describing the state of the host. This YAML document is based on data taken from Puppet Manifests, but is subject to Smart Class Parameter overrides and any Smart Variables.

Applying a Smart Variable to a Host

As Smart Variables should only be used to support your custom Puppet modules previously modified to include a global parameter, the following example uses a simple example called **anothermodule**. The **anothermodule** Puppet Manifest is as follows:

```
class anothermodule {
  file { '/tmp/motd':
    ensure  => file,
    content => $::content_for_motd,
  }
}
```

This example will supply a value for the **$::content_for_motd** parameter.

1. In the web UI, navigate to **Configure** > **Classes**.

2. Select the name of the Puppet class from the list.

3. Click the **Smart Variables** tab. This displays a new screen. The left section contains a list of previously created parameters, if any. The right section contains the configuration options.

4. Click **Add Variable** to add a new parameter.

5. Enter the parameter in the **Key** field. In this example, **content_for_motd**.

6. Edit the **Description** text box, for example **Testing /tmp motd Text**.

7. Select the **Parameter Type** of data to pass. Select **string**.

8. Type a **Default Value** for the parameter. For example, **No Unauthorized Use**.

9. Use the **Optional Input Validator** section to restrict the allowed values for the parameter. Choose a **Validator Type** (either a **list** of comma separated values or a regular expression, **regexp**) and input the allowed values or regular expression code in the **Validator rule** field.

10. Use the **Prioritize Attribute Order** section to set the order of precedence in which the host attributes or Facts are to be evaluated against the matchers (configured below). You can rearrange the entries in the list and add to the default list. To create a logical AND condition between matchers, arrange the names of the matchers on one line as a comma separated list.

11. In the **Specify Matchers** section, click **Add Matcher** to add a conditional argument. The attributes to match against should correspond to the entries in the **Order** list above. If no matcher is configured, then only the default value can be used. For example, if the desired value of the parameter is **This is for Server1** for any host with a fully qualified domain name of **server1.example.com**, then specify the **Match** as **fqdn=server1.example.com** and the **Value** as **This is for Server1**.

12. Click **Submit** to save your changes.

CHAPTER 4. STORING AND MAINTAINING HOST INFORMATION

Red Hat Satellite 6 uses a combination of applications to gather information about managed hosts and to ensure that those hosts are maintained in the desired state. These applications include:

- Foreman: Provides for the provisioning and life cycle management of physical and virtual systems. Foreman automatically configures these systems using various methods, including kickstart and Puppet modules.

- Puppet: A client/server architecture for configuring hosts, consisting of the Puppet master (server) and the Puppet agent (client).

- Facter: Puppet's system inventory tool. Facter gathers basic information (facts) about hosts such as hardware details, network settings, OS type and version, IP addresses, MAC addresses, SSH keys, and more. These facts are then made available in Puppet manifests as variables.

The use of Puppet, Facter, and facts is discussed in more detail below.

4.1. THE PUPPET ARCHITECTURE

Puppet usually runs in an agent/master (also known as a client/server) architecture, where a Puppet Server controls important configuration information, and managed hosts (clients) request only their own configuration catalogs. Puppet configures hosts in two steps:

- It compiles a catalog.

- It applies that catalog to the appropriate host.

In the agent/master setup, the Puppet client sends facts gathered by Facter and other information to the Puppet master. The Puppet master compiles a catalog based on these facts, and then sends this catalog to the client. The client sends a report of all the changes it made, or would have made if the **--noop** parameter had been used, to the Puppet master, which in turn sends the results to Foreman. This catalog describes the desired state for one specific host. It lists the resources to manage on that host, including any dependencies between those resources. The agent applies the catalog to the host.

This communication between master and agent occurs every 30 minutes by default. You can specify a different value in the **/etc/puppetlabs/puppet/puppet.conf** file using the **runinterval** parameter. You can also run **puppet agent apply** to initiate communication manually.

4.2. USING FACTER AND FACTS

Facter is Puppet's system inventory tool, and includes a large number of built-in facts. You can run Facter at the command line on a local host to display fact names and values. You can extend Facter with custom facts, and then use these to expose site-specific details of your hosts to your Puppet manifests. You can also use the facts provided by Facter to inform conditional expressions in Puppet.

Puppet determines a system state based on resources; for example, you can tell Puppet that the **httpd** service should always be running and Puppet knows how to handle that. If you are managing different operating systems, you can use the **osfamily** fact to create conditional expressions to tell Puppet which service to watch or which package to install. You can use the **operatingsystemmajrelease** and **versioncmp** parameters to create conditional expressions based on different versions of the same operating system. The following example shows the use of conditional expressions with Facts.

Using Conditional Expressions with Facts

```
if $::osfamily == 'RedHat' {
  if $::operatingsystemmajrelease == '6' {
   $ntp_service_name = 'ntpd'
   }

  elseif versioncmp($::operatingsystemmajrelease, '7') &gt;= 0 {
   $ntp_service_name = 'chrony'
   }
  }
```

> **NOTE**
>
> This example uses the expression **versioncmp($::operatingsystemmajrelease, '7') >= 0** to test for version 7 or later of Red Hat Enterprise Linux. Do not use the expression **$::operatingsystemmajrelease >= '7'** to perform this test. See https://docs.puppetlabs.com/references/latest/function.html#versioncmp for more information about this and other Puppet functions.

Puppet also sets other special variables that behave a lot like facts. See Special Variables Added by Puppet and Core Facts for more information.

4.2.1. Displaying Facts for a Particular Host

Puppet can access Facter's built-in core facts as well as any custom or external facts present in your Puppet modules. You can view available facts from the command line (**facter -p**) and also from the web UI (**Monitor > Facts**). You can browse the list of facts or use the **Search** box to search for specific facts. For example, type "facts." to display a list of available facts.

> **NOTE**
>
> The list of available facts is very long. The UI only displays 20 facts at a time. The list of facts gradually filters as you enter more details. For example, type "facts.e" to display all facts that begin with the letter "e."

Viewing Facts for a Particular Host

1. On the main menu, click **Hosts > All Hosts** and then click the name of the host that you want to inspect.

2. In the **Details** pane, click **Facts** to display all known facts about the host.

> **NOTE**
>
> - For any fact listed on this page, you can click **Chart** to display a chart of the distribution of this fact name over all managed hosts.
>
> - You can bookmark a search to make it easier to use in the future. When you have refined your search, click the drop-down arrow next to the **Search** button, and click **Bookmark this search**. Bookmarked searches appear in the **Search** drop-down list, and also under **Administer > Bookmarks** on the main menu.

4.2.2. Searching for Hosts based on Facts

You can use Facter information to search for specific hosts. This means that you can search for all hosts that match specific fact criteria, such as **facts.architecture = x86_64**.

Searching for Hosts based on Facts

1. On the main menu, click **Monitor > Facts** to display the **Fact Values** page.

2. In the **Search** field, start typing the name of the fact that you want to filter by. You can search by specific name, name/value pairs, and so on.

3. Click **Search** to retrieve the list of matching hosts.

4.2.3. Custom Fact Reporting

Obtaining custom information from managed hosts is fully supported with Red Hat Satellite 6. This section illustrates using a Puppet module obtained from Puppet Forge, but the principle applies equally for other sources of Puppet modules.

The number of facts reported via the standard Facter interface can be extended. For example, to gather a fact for use as a variable in modules. If a fact that describes the packages installed was available, you could search this data and make informed configuration management decisions based on the information.

To obtain a report on the packages installed on a host the process is as follows:

- The manifest **pkginventory** is obtained from Puppet Forge and saved to the base system.

- The Puppet module is added to a Content View and then this is promoted to a system and deployed to that system.

- The facts for the system are then queried using a package name. In this example, for a host called *hostname* and using a Satellite user with credentials *username* and *password*, the following API query would return the facts that matched the search string "bash":

  ```
  curl -u username:password -X GET http://localhost/api/hosts/:hostname/facts?search=bash
  {"hostname":{"pkg_bash":"4.2.45-5.el7_0.4"}}
  ```

  The search returns the package version. This could then be used to populate an external database.

4.2.3.1. Adding the pkginventory Puppet Module

To add the **pkginventory** Puppet module to the Red Hat Satellite Server application, download the module from https://forge.puppetlabs.com/ody/pkginventory to the base system where the Satellite Server application is installed and then follow the procedures below.

Puppet modules are usually stored in a custom repository named Puppet Modules. The following procedure assumes you have made a custom repository with that name. If you have not yet made a custom repository for Puppet Modules, see Creating a Custom Product in the *Quick Start Guide*.

Uploading a Puppet Module to a Repository

1. Download the Puppet module to the base system. Modules that are downloaded will have a **.tar.gz** extension.

2. Click **Content > Products** and then click the product name in the **Name** field associated with the Puppet module repository. For example, **Custom Products**.

3. On the **Repositories** tab, select the Puppet Modules repository you want to modify. For example, **Puppet Modules**.

4. In the **Upload Puppet Module** section, click **Browse**, and navigate to the module that you downloaded.

5. Click **Upload**.

To distribute a Puppet module to clients, content hosts, the module must be applied to a Content View and published. Follow this procedure to add a module to a Content View.

Adding a Module to a Content View

1. Click **Content > Content Views** and then select a Content View from the **Name** menu.

2. On the **Puppet Modules** tab, click **Add New Module**. A list of installed modules appears.

3. From the **Actions** column, click **Select a Version** to select the module you want to add. A table of available versions appears.

4. Click **Select Version** next to the version of the module that you want to add.

5. Click **Publish New Version** to create the new Content View.

6. Optionally add a description and click **Save**.

CHAPTER 5. CLIENT AND SERVER SETTINGS FOR CONFIGURATION MANAGEMENT

An important part of Red Hat Satellite 6's configuration process is making sure the Puppet clients (called Puppet agents) can communicate with the Puppet Server (called Puppet master) on either the internal Satellite Capsule or an external Satellite Capsule. This chapter examines how Red Hat Satellite 6 configures both the Puppet master and the Puppet agent.

5.1. CONFIGURING PUPPET ON THE RED HAT SATELLITE SERVER

Red Hat Satellite 6 controls the main configuration for the Puppet master on all Satellite Capsules. No additional configuration is required and it is recommended to avoid manually modify these configuration files. For example, the main **/etc/puppetlabs/puppet/puppet.conf** configuration file contains the following **[master]** section:

```
[master]
    autosign       = $confdir/autosign.conf { mode = 664 }
    reports        = foreman
    external_nodes = /etc/puppetlabs/code/node.rb
    node_terminus  = exec
    ca             = true
    ssldir         = /var/lib/puppet/ssl
    certname       = sat6.example.com
    strict_variables = false


    manifest       = /etc/puppetlabs/code/environments/$environment/manifests/site.pp
    modulepath     = /etc/puppetlabs/code/environments/$environment/modules
    config_version =
```

This section contains variables (such as **$environment**) that Satellite 6 uses to create configuration for different environments.

Some Puppet configuration options appear in the Satellite 6 web UI. Navigate to **Administer** > **Settings** and choose the **Puppet** subtab. This page lists a set of Puppet configuration options and a description of each.

5.2. CONFIGURING PUPPET AGENT ON PROVISIONED SYSTEMS

As part of the provisioning process, Satellite 6 installs Puppet to the system. This process also installs the **/etc/puppetlabs/puppet/puppet.conf** file that configures Puppet as an agent of the Puppet master on a chosen Capsule. This configuration file is stored as a provisioning template snippet in Satellite 6. Navigate to **Hosts** > **Provisioning templates** and click the **puppet.conf** snippet to view it.

The default **puppet.conf** snippet contains the following agent configuration:

```
[agent]
pluginsync      = true
report          = true
ignoreschedules = true
daemon          = false
ca_server       = <%= @host.puppet_ca_server %>
```

```
certname       = <%= @host.certname %>
environment    = <%= @host.environment %>
server         = <%= @host.puppetmaster %>
```

This snippet contains some template variables, which are:

- **@host.puppet_ca_server** and **@host.certname** — The certificate and certificate authority for securing Puppet communication.

- **@host.environment** — The Puppet environment on the Satellite 6 Server to use for configuration.

- **@host.puppetmaster** — The host containing the Puppet master. This is either the Satellite 6 Server's internal Capsule or an external Satellite Capsule.

CHAPTER 6. APPLYING CONFIGURATION ON CLIENTS

At this point, Satellite 6 Server's Puppet ecosystem is configured and contains the **mymodule** module. We now aim to apply this module's configuration to a registered system.

6.1. APPLYING CONFIGURATION ON CLIENTS DURING PROVISIONING

We first define a new host's Puppet configuration using the following procedure. This procedure uses the uploaded **mymodule** as an example.

Applying Configuration on Clients During Provisioning

1. Navigate to **Hosts** > **New host**.

2. Click the **Host** tab. Enter a **Name** for the host and choose the organization and location for the system. Choose the **Lifecycle Environment** and its promoted **Content View**. This defines the Puppet environment to use for the configuration. Also choose a **Puppet CA** and **Puppet Master** from the **Capsule Settings**. The chosen capsule acts as the server that controls the configuration and communicates with the agent on the new host.

3. Click the **Puppet Classes** tab and from the **Available Classes** section choose the Puppet classes that contain the configuration to apply. In our example, choose:

   - **mymodule**

   - **mymodule:httpd**

   - **mymodule:app**

4. Choose the necessary options from the **Operating System** tab. These options depend on your own Satellite 6 infrastructure. Make sure the **Provisioning templates** option includes the **Satellite Kickstart Default** kickstart template. This template contains installation commands for the Puppet agent on the new host.

5. Click the **Parameters** tab and provide any custom overrides to our Puppet class parameters. See Section 3.3, "Configuring Smart Class Parameters" to enable this feature.

6. After completing all provisioning options, click **Submit**.

The provisioning process begins. Satellite 6 installs the required configuration tools as part of the **Satellite Kickstart Default** provisioning template. This provisioning template contains the following:

```
<% if puppet_enabled %>
# and add the puppet package
yum -t -y -e 0 install puppet

echo "Configuring puppet"
cat > /etc/puppetlabs/puppet/puppet.conf << EOF
<%= snippet 'puppet.conf' %>
EOF

# Setup puppet to run on system reboot
/sbin/chkconfig --level 345 puppet on
```

```
/usr/bin/puppet agent --config /etc/puppetlabs/puppet/puppet.conf -o --tags no_such_tag <%=
@host.puppetmaster.blank? ? " : "--server #{@host.puppetmaster}" %> --no-daemonize
<% end -%>
```

This section performs the following:

- Installs the **puppet** package from the Red Hat Satellite Tools 6.10 repository.

- Installs the Puppet configuration snippet to the system at
  /etc/puppetlabs/puppet/puppet.conf.

- Enables the Puppet service to run on the system.

- Runs Puppet for the first time and initializes the node.

After the provisioning and configuration processes complete on the new host, access the host at the user-defined port in your web browser. For example, navigate to **http://newhost.example.com:8120/**. The following message should appear in your browser:

> Congratulations
>
> Your puppet module has correctly applied your configuration.

6.2. APPLYING CONFIGURATION TO EXISTING CLIENTS

You might aim to have Puppet configuration applied to an existing client not provisioned through Red Hat Satellite 6. In this situation, install and configure Puppet on the existing client after registering it to Red Hat Satellite 6.

Register your existing system to Red Hat Satellite 6. For information on registering existing hosts, see Registering Hosts in the *Managing Hosts* guide.

IMPORTANT

The **puppet** package is part of the Red Hat Satellite Tools 6.10 repository. Ensure you enable this repository before you proceed.

To Install and Enable the Puppet Agent:

1. Open a terminal console and log in as root.

2. Install the Puppet agent:

   ```
   # yum install puppet
   ```

3. Configure the Puppet agent to start at boot:

   - On Red Hat Enterprise Linux 6:

     ```
     # chkconfig puppet on
     ```

   - On Red Hat Enterprise Linux 7:

     ```
     # systemctl enable puppet
     ```

Configuring the Puppet Agent

1. Configure the Puppet agent by changing the **/etc/puppetlabs/puppet/puppet.conf** file:

   ```
   # vi /etc/puppetlabs/puppet/puppet.conf
   ```

   ```
   [main]
       # The Puppet log directory.
       # The default value is '$vardir/log'.
       logdir = /var/log/puppet

       # Where Puppet PID files are kept.
       # The default value is '$vardir/run'.
       rundir = /var/run/puppet

       # Where SSL certificates are kept.
       # The default value is '$confdir/ssl'.
       ssldir = $vardir/ssl

   [agent]
       # The file in which puppetd stores a list of the classes
       # associated with the retrieved configuratiion.  Can be loaded in
       # the separate puppet executable using the --loadclasses
       # option.
       # The default value is '$confdir/classes.txt'.
       classfile = $vardir/classes.txt
       pluginsync = true
       report = true
       ignoreschedules = true
       daemon = false
       ca_server = satellite.example.com
       server = satellite.example.com
       environment = KT_Example_Org_Library_RHEL6Server_3

       # Where puppetd caches the local configuration.  An
       # extension indicating the cache format is added automatically.
       # The default value is '$confdir/localconfig'.
       localconfig = $vardir/localconfig
   ```


   IMPORTANT

   Set the **environment** parameter to the host's Puppet environment from the Satellite Server. The Puppet environment label contains the organization label, life-cycle environment, content view name, and the content view ID. To see a list of Puppet environments in the Satellite 6 web UI, navigate to **Configure** > **Environments**.

2. Run the Puppet agent on the host:

   ```
   # puppet agent -t --server satellite.example.com
   ```

3. Sign the SSL certificate for the Puppet client through the Satellite Server web interface:

   a. Log in to the Satellite Server through the web interface.

b. Select **Infrastructure** > **Capsules**.

c. Click **Certificates** to the right of the required host.

d. Click **Sign**.

e. Rerun the **puppet agent** command:

```
# puppet agent -t --server satellite.example.com
```

> **NOTE**
>
> When the Puppet agent is configured on the host it will be listed under **All Hosts** but only when **Any Context** is selected as the host will not be assigned to an organization or location.

CHAPTER 7. USING CONFIG GROUPS TO MANAGE PUPPET CLASSES

Red Hat Satellite includes the concept of *Config Groups* and *Host Groups* to enable building and managing groups of systems in a modular way.

Config Groups are a collection of Puppet classes that you create to form building blocks for use in configuring Hosts and Host Groups. Config Groups are analogous to the community Puppet concept of Profiles, a Puppet class that includes a collection of Puppet classes to form a building block. Config Groups can be created and managed in the Satellite web UI.

Host Groups are both a collection of host servers and a container for the definition of a system, with a Content View, an assigned life cycle, and a set of Puppet modules. Host Groups are analogous to the community Puppet concept of Roles, a Puppet class that includes a number of Profiles to build a system with a specific business role. Host Groups can be created and managed in the Satellite web UI, using the Hammer CLI, and by using the API.

Creating a Config Group

From the left-hand side context drop-down menu, select **Any Organization** and **Any Location**.

Navigate to **Configure** > **Config groups**.

Select **New Config Group** and enter a name, for example, *TestConfGroup*.

Select one or more **Puppet Classes** from the list of available classes.

Select **Submit** to apply the changes.

Once you have created a Config Group it will be available for selection on the Puppet Classes tab when configuring a Host or Host Group. For more information on creating Host Groups see Creating a Host Group on Satellite Server in the *Red Hat Satellite Provisioning Guide* .

CHAPTER 8. REVIEWING PUPPET REPORTS IN RED HAT SATELLITE 6

Puppet generates a report each time it applies a configuration. Provisioned hosts send this report to the Red Hat Satellite 6 Server. View these reports on the hosts details page.

Reviewing Puppet Reports in Red Hat Satellite 6

1. Navigate to **Hosts** > **All hosts**.

2. Click the **Name** of your desired host.

3. Click the **Reports** button.

4. Select a report to view.

Each report shows the status of each Puppet resource and its configuration applied to the host.