



Red Hat Satellite 6.2

Puppet Guide

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

Red Hat Satellite 6.2 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

Copyright © 2018 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution-Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

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	3
1.1. DEFINING THE PUPPET WORKFLOW	3
1.2. USING PUPPET ON SATELLITE 6	3
1.3. PUPPET PERFORMANCE AND SCALABILITY ON SATELLITE 6	4
CHAPTER 2. BUILDING PUPPET MODULES FROM SCRATCH	5
2.1. EXAMINING THE ANATOMY OF A PUPPET MODULE	5
2.2. SETTING UP A PUPPET DEVELOPMENT SYSTEM	6
2.3. GENERATING A NEW MODULE BOILERPLATE	6
2.4. INSTALLING A HTTP SERVER	7
2.5. RUNNING THE HTTP SERVER	8
2.6. CONFIGURING THE HTTP SERVER	9
2.7. CONFIGURING THE FIREWALL	11
2.8. CONFIGURING SELINUX	13
2.9. COPYING A HTML FILE TO THE WEB HOST	14
2.10. FINALIZING THE MODULE	15
CHAPTER 3. ADDING PUPPET MODULES TO RED HAT SATELLITE 6	16
3.1. CREATING A CUSTOM PRODUCT	16
3.2. CREATING A PUPPET REPOSITORY IN A CUSTOM PRODUCT	16
3.3. UPLOADING A PUPPET MODULE TO A REPOSITORY	17
3.4. REMOVING A PUPPET MODULE FROM A REPOSITORY	17
3.5. ADDING PUPPET MODULES FROM A GIT REPOSITORY	17
3.6. PUBLISHING A CONTENT VIEW	19
3.7. PUPPET ENVIRONMENTS	20
3.8. CONFIGURING SMART CLASS PARAMETERS FROM PUPPET CLASSES	21
3.9. USING THE SMART VARIABLE TOOL	23
CHAPTER 4. CLIENT AND SERVER SETTINGS FOR CONFIGURATION MANAGEMENT	26
4.1. CONFIGURING PUPPET ON THE RED HAT SATELLITE SERVER	26
4.2. CONFIGURING PUPPET AGENT ON PROVISIONED SYSTEMS	26
CHAPTER 5. APPLYING CONFIGURATION ON CLIENTS	28
5.1. APPLYING CONFIGURATION ON CLIENTS DURING PROVISIONING	28
5.2. APPLYING CONFIGURATION TO EXISTING CLIENTS	29
CHAPTER 6. USING CONFIG GROUPS TO MANAGE PUPPET CLASSES	32
CHAPTER 7. REVIEWING PUPPET REPORTS IN RED HAT SATELLITE 6	33

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

Satellite 6 uses Puppet in several ways:

- Satellite 6 imports Puppet modules used to define the system configuration. This includes control over module versions and their environments.
- 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.
- 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 Red Hat Knowledgebase article [Satellite 6 Component Versions](#).

1.3. PUPPET PERFORMANCE AND SCALABILITY ON SATELLITE 6

The performance of Puppet in Satellite 6 is driven 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 Satellite 6.2, the maximum number of concurrent Puppet runs is approximately 320. This should improve significantly in Satellite 6.3 due to Puppet Server replacing the Apache Passenger module and Puppet master.

Importing many Puppet modules simultaneously can take a long time. For example, on a notebook with 8GB RAM and 4 cores, importing 2000 modules takes approximately 4 minutes.

The following performance brief in the Red Hat Knowledgebase provides examples of performance tests made for Satellite 6.2: [Performance & Scale Tuning of Satellite 6.2 and Capsules](#) . The performance of Satellite 6.3 is expected to be noticeably better than indicated in the performance brief for Satellite 6.2.

For information on recommended storage and hardware requirements, see [Hardware Requirements](#) in the *Red Hat Satellite Installation Guide*

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.

Manifests

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.

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. It is recommended to use a new system with a Red Hat Enterprise Linux 6 or 7 subscription.

After installing the new system and registering your version of Red Hat Enterprise Linux, enable the Red Hat Satellite 6 Tools repository. For example, for Red Hat Enterprise Linux 7:

```
# subscription-manager repos --enable=rhel-7-server-satellite-tools-6.2-rpms
```

After enabling the repository, install the `puppet` package:

```
# yum install puppet
```

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 Puppet to create a boilerplate for your module:

```
# cd /etc/puppet/modules
# puppet module generate user_name-module_name
```

An interactive wizard appears and guides you through populating the module's `metadata.json` file with metadata.



IMPORTANT

The `puppet module generate` command requires the module name take the format of `user_name-module_name` to comply with Puppet Forge specifications. However, to test our tutorial module and use it with Satellite 6 we need to rename the module directory to remove the user name. For example, for `user_name-mymodule` you would run:

```
# puppet module generate user_name-mymodule
# mv user_name-mymodule mymodule
```

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 a `tests` 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/tests/init.pp --noop
```

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. The output should resemble the following:

```
Notice: Compiled catalog for puppet.example.com in environment production
in 0.59 seconds
Notice: /Stage[main]/Mymodule::Httpd/Package[httpd]/ensure: current_value
absent, should be present (noop)
Notice: Class[Mymodule::Httpd]: Would have triggered 'refresh' from 1
events
Notice: Stage[main]: Would have triggered 'refresh' from 1 events
Notice: Finished catalog run in 0.67 seconds
```

The highlighted line 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.

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: Compiled catalog for puppet.example.com in environment production
in 0.56 seconds
Notice: /Stage[main]/Mymodule::Httpd/Package[httpd]/ensure: current_value
absent, should be present (noop)
Notice: /Stage[main]/Mymodule::Httpd/Service[httpd]/ensure: current_value
stopped, should be running (noop)
Notice: Class[Mymodule::Httpd]: Would have triggered 'refresh' from 2
events
Notice: Stage[main]: Would have triggered 'refresh' from 1 events
Notice: Finished catalog run in 0.41 seconds
```

The highlighted line 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
Warning: Config file /etc/puppet/hiera.yaml not found, using Hiera
defaults
Notice: Compiled catalog for puppet.example.com in environment production
in 0.84 seconds
Notice: /Stage[main]/Mymodule::Httpd/File[/var/www/myserver]/ensure:
current_value absent, should be directory (noop)

```

```

Notice: /Stage[main]/Mymodule::Httpd/Package[httpd]/ensure: current_value
absent, should be present (noop)
Notice:
/Stage[main]/Mymodule::Httpd/File[/etc/httpd/conf.d/myserver.conf]/ensure:
current_value absent, should be file (noop)
Notice: /Stage[main]/Mymodule::Httpd/Service[httpd]/ensure: current_value
stopped, should be running (noop)
Notice: Class[Mymodule::Httpd]: Would have triggered 'refresh' from 4
events
Notice: Stage[main]: Would have triggered 'refresh' from 1 events
Notice: Finished catalog run in 0.51 seconds

```



NOTE

The warning for the `hiera.yaml` file is safe to ignore.

The highlighted lines 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
&&& iptables-save > /etc/sysconfig/iptables",
    path => "/sbin",
    refreshonly => true,
    subscribe => Package['httpd'],
  }
  service { 'iptables':
    ensure => running,
    enable => true,
    hasrestart => true,
    subscribe => Exec['iptables'],
  }
}
elseif $operatingsystemmajrelease == 7 {
  exec { 'firewall-cmd':
    command => "firewall-cmd --zone=public --add-port=${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.

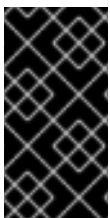
Run the `puppet apply` command again to test the changes to our module. The following example is a test of Red Hat Enterprise Linux 6:

```

# puppet apply mymodule/tests/init.pp --noop
Warning: Config file /etc/puppet/hiera.yaml not found, using Hiera
defaults
Notice: Compiled catalog for puppet.example.com in environment production
in 0.82 seconds
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
...

```

The highlighted lines 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 `polycoreutils-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['polycoreutils-python'],
  before => Service['httpd'],
  subscribe => Package['httpd'],
  refreshonly => true,
}
package { 'polycoreutils-python':
  ensure => installed,
}
```

This code performs the following:

- The `require => Package['polycoreutils-python']` attribute makes sure the `polycoreutils-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[polycoreutils-
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 `polycoreutils-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
Warning: Config file /etc/puppet/hiera.yaml not found, using Hiera
defaults
```

```

Notice: Compiled catalog for puppet.example.com in environment production
in 0.66 seconds
Notice: /Stage[main]/Mymodule::Httpd/Exec[iptables]/returns: current_value
notrun, should be 0 (noop)
Notice: /Stage[main]/Mymodule::Httpd/Package[policycoreutils-
python]/ensure: current_value absent, should be present (noop)
Notice: /Stage[main]/Mymodule::Httpd/Service[iptables]: Would have
triggered 'refresh' from 1 events
Notice: /Stage[main]/Mymodule::Httpd/File[/var/www/myserver]/ensure:
current_value absent, should be directory (noop)
Notice: /Stage[main]/Mymodule::Httpd/Package[httpd]/ensure: current_value
absent, should be present (noop)
Notice:
/Stage[main]/Mymodule::Httpd/File[/etc/httpd/conf.d/myserver.conf]/ensure:
current_value absent, should be file (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)
Notice: Class[Mymodule::Httpd]: Would have triggered 'refresh' from 8
events
Notice:
/Stage[main]/Mymodule::App/File[/var/www/myserver/index.html]/ensure:
current_value absent, should be file (noop)
Notice: Class[Mymodule::App]: Would have triggered 'refresh' from 1 events
Notice: Stage[main]: Would have triggered 'refresh' from 2 events
Notice: Finished catalog run in 0.74 seconds

```

The highlighted line 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/puppet/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 6

Puppet modules form a part of a Satellite Product in Red Hat Satellite 6. This means you must create a custom Product and then upload the modules that form the basis of that Product. For example, a custom Product might consist of a set of Puppet modules required to setup an HTTP server, a database, and a custom application. Custom Products can also include repositories with RPM packages that apply to your application.

3.1. CREATING A CUSTOM PRODUCT

The first step to adding our Puppet module is to create a custom Product.

Creating a Custom Product

1. Login to your Red Hat Satellite 6 Server.
2. Navigate to **Content > Products**.
3. Click **+ New Product**.
4. Provide your custom Product with a **Name**. In this example, use **MyProduct** as the name.
5. The **Label** field automatically populates with a label based on the **Name**.
6. Provide a **GPG Key**, **Sync Plan**, and a **Description** if required. For our example, leave those fields blank.
7. Click **Save**.

Satellite now has a new Product called **MyProduct**.

3.2. CREATING A PUPPET REPOSITORY IN A CUSTOM PRODUCT

The next procedure creates a Puppet repository in our custom Product.

Creating a Custom Puppet Repository

1. On the **Products** page, click on the custom Product created previously (**MyProduct**).
2. Navigate to the **Repositories** subtab.
3. Click **Create Repository**.
4. Provide the repository with a **Name**. This example uses the name **MyRepo**.
5. The **Label** field automatically populates with a label based on the **Name**.
6. Select **puppet** as the repository **Type**.
7. Leave the **URL** field blank. This field is used for remote repositories, but in our case Satellite 6 creates its own repository.
8. Click **Save**.

The custom Product now contains a repository to store our Puppet modules.

3.3. UPLOADING A PUPPET MODULE TO A REPOSITORY

Now we upload our `mymodule` module to the newly created repository, which adds it to our custom Product.

1. Click the **Name** of the newly created repository.
2. In the **Upload Puppet Module** section, click **Browse** and select the `mymodule` archive.
3. Click **Upload**.

You can upload more modules to this repository. For our example, we only need to upload the `mymodule` module.

Our Puppet module is now a part of your Red Hat Satellite 6 environment. Next we publish the module as part of a content view.

3.4. REMOVING A PUPPET MODULE FROM A REPOSITORY

If you aim to remove redundant modules from custom repositories in the future, use the **Manage Puppet Modules** feature.

1. On the **Products** page, click on the custom Product containing the module to remove.
2. Click the **Name** of the repository containing the module to remove.
3. Click **Manage Puppet Modules**. The screen displays a list of Puppet modules contained within the repository.
4. Select the modules to remove.
5. Click **Remove Puppet Modules**.

Satellite removes the chosen modules from your repository.

3.5. ADDING PUPPET MODULES FROM A GIT REPOSITORY

As an alternative to manually uploading modules, Red Hat Satellite 6 includes a utility called `pulp-puppet-module-builder`. This tool checks out repositories containing a set of modules, builds the modules, and publishes them in a structure for Satellite 6 to synchronize. This provides an efficient way to manage module development in Git and include them in the Satellite 6 workflow.

The use of `Modulefile` is deprecated since Puppet version 3. If you build any Puppet module containing a `Modulefile` when using Puppet version 3.X, a deprecation warning is printed. Satellite 6.3 will include Puppet version 4 which will ignore the `Modulefile` data. The data in the `Modulefile` must be moved to a `metadata.json` file. You can convert a module to use a `metadata.json` file as follows:

1. Run `puppet module build Module_Directory` once.
2. Delete the `Modulefile`.
3. Check the updated `metadata.json` file into the revision control repository.

**NOTE**

You can also install the `pulp-puppet-module-builder` tool on other machines using the `pulp-puppet-tools` package.

One common method is to run the utility on the Satellite 6 Server itself and publish to a local directory.

Publishing a Git Repository to a Local Directory

1. Create a directory on the Satellite Server to synchronize the modules:

```
# mkdir /var/www/puppet-modules
# chmod 755 /var/www/puppet-modules
```

Store your modules under the `/var/www/` directory, otherwise SELinux will block the repository synchronization. If required to use another directory, you can use the `httpd_sys_r_content_t` or `pulp_tmp_t` SELinux type. Using the `httpd_sys_r_content_t` SELinux type allows the webserver to read the files. See the [SELinux User's and Administrator's Guide](#) for information on setting SELinux file types.

2. Run the `pulp-puppet-module-builder` and checkout the Git repository:

```
# pulp-puppet-module-builder --output-dir=/var/www/puppet-modules \
--url=git@mygitserver.com:mymodules.git --branch=develop
```

This checks out the `develop` branch of the Git repository from `git@mygitserver.com:mymodules.git` and publishes the modules to `/var/www/puppet-modules/`.

The same procedure applies to publishing modules to an HTTP server.

Publishing Git Repository to a Web Server

1. Create a directory on the web server to synchronize the modules.

```
# mkdir /var/www/html/puppet-modules
# chmod 755 /var/www/html/puppet-modules
```

2. Run the `pulp-puppet-module-builder` and checkout the Git repository.

```
# pulp-puppet-module-builder \
--output-dir=/var/www/html/puppet-modules \
--url=git@mygitserver.com:mymodules.git --branch=develop
```

This checks out the `develop` branch of the Git repository from `git@mygitserver.com:mymodules.git` and publishes the modules to `/var/www/html/puppet-modules/`.

In the Satellite 6 web UI, create a new repository with the URL set to the location of your published modules.

Creating a Repository for Puppet Modules from Git

1. On the **Products** page, click on the custom Product created previously (**MyProduct**).

2. Navigate to the **Repositories** subtab.
3. Click **Create Repository**.
4. Provide the repository with a **Name**. This example uses the name **MyGitRepo**.
5. The **Label** field automatically populates with a label based on the **Name**.
6. Select **puppet** as the repository **Type**.
7. In the URL field, set the location you defined earlier. For example, local directories on the Satellite 6 Server use the **file://** protocol:

```
file:///modules
```

A remote repository uses the **http://** protocol:

```
http://webserver.example.com/modules/
```

8. Click **Save**.
9. Click **Sync Now** to synchronize the repository.

The Puppet modules in the Git repository are now included in your Satellite 6 Server.

3.6. PUBLISHING A CONTENT VIEW

The final step to getting our Puppet module ready for consumption is to publish it as part of a content view. You can add this module to an existing view but for our example we will create a new view.

Publishing a Content View

1. Navigate to **Content > Content Views**.
2. Click **+ Create New View**.
3. Provide your view with a **Name**. In this example, we use **MyView** as the name.
4. The **Label** field automatically populates with a label based on the **Name**.
5. Make sure **Composite Content View** is not selected.
6. Click **Save**.
7. Select the **Name** of your newly created view.
8. Navigate to **Content > Repositories**.
9. Add the required Red Hat Enterprise Linux repositories, including a base Red Hat Enterprise Linux Server RPM collection and a Red Hat Satellite Tools RPM collection for the same version. The Tools RPM collection contains the packages to set up our remote Puppet configuration on provisioned systems.
10. Navigate to **Puppet Modules**.
11. Click **+ Add New Module**.

12. Scroll to your module and click **Select a Version**.
13. Scroll to the module version **Use Latest** and click **Select Version**.
14. Our module is now a part of the content view. Navigate to **Versions** to publish and promote a new version of the content view.
15. Click **Publish New Version**. On the **Publish New Version** page, click **Save**. This publishes the content view with our module.
16. Scroll to the new version of our view and click **Promote**. Choose a life-cycle environment and click **Promote Version**. This makes the view a part of the chosen life-cycle environment.

Our content view is now published. As a part of the content view creation, Red Hat Satellite 6 creates a new Puppet environment for use in the provisioning process. This Puppet environment contains our module. You can view this new Puppet environment on the **Configure > Environments** page.

3.7. 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 6 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

Navigate to **Configure > Environments** and click **New Puppet Environment**. 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.

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 on 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.

3.8. CONFIGURING SMART CLASS PARAMETERS FROM PUPPET CLASSES

Some module classes contain complex parameters, called Smart Class Parameters, that define a value for a key just as a simple parameter does, but allow conditional arguments, validation, and overrides for specific object types. Satellite 6 has the ability to import these parameterized classes and allow modification of such smart parameters. For more information on the types of parameters in Puppet, see [Parameters](#) in the *Red Hat Satellite Host Configuration Guide*

For example, **mymodule** contains a parameter for the HTTP port of our web server. This parameter, **httpd_port**, is set to a default of 8120. However, a situation might occur where we need to use a different port for a provisioned system. Satellite 6 can override the **httpd_port** parameter during configuration. This provides an easy way to change the HTTP port on our web server.

This procedure requires the **mymodule** module to be uploaded to a Product and added to a Content View. This is because we need to edit the classes in the resulting Puppet environment. Alternatively, you can download and install another module as described in [Chapter 3, Adding Puppet Modules to Red Hat Satellite 6](#).

1. Navigate to **Configure > Smart class parameters**.
2. A table appears listing all Smart Class Parameters from the classes in your Puppet modules. Enter **httpd_port** in the search field. Click on the **httpd_port** parameter.
3. The options for the Smart Class Parameter appears. To allow overriding this parameter during provisioning, select the **Override** option.
4. Selecting the **Override** option allows us to change the **Key type** and **Default value**. This is useful if we aim to globally change this value for all future configurations.
The following key types are available:

String

The value is interpreted as a plain text string. For example, if your Smart Class Parameter sets the host name, the value is interpreted as a string:

```
myhost.example.com
```

Boolean

The value is interpreted and validated as a true or false value. Examples include:

```
True
true
1
```

Integer

The value is interpreted and validated as an integer value. Examples include:

```
8120
-8120
```

Real

The value is interpreted and validated as a real number value. Examples include:

```
8120
-8120
8.12
```

Array

The value is interpreted and validated as a JSON or YAML array. For example:

```
["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"]
```

Hash

The value is interpreted and validated as a JSON or YAML hash map. For example:

```
{"Weekdays": ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"],
 "Weekend": ["Saturday", "Sunday"]}
```

YAML

The value is interpreted and validated as a YAML file. For example:

```
email:
  delivery_method: smtp
  smtp_settings:
    address: smtp.example.com
    port: 25
    domain: example.com
    authentication: none
```

JSON

The value is interpreted and validated as a JSON file. For example:

```
{
  "email": [
    {
      "delivery_method": "smtp"
      "smtp_settings": [
        {
```

```

    "address": "smtp.example.com",
    "port": 25,
    "domain": "example.com",
    "authentication": "none"
  }
]
}

```

For this example, leave the default as 8120.

5. Selecting the **Override** option also exposes **Optional Input Validator**, which provides validation for the overridden value. For example, we can include a regular expression to make sure `ht t p d _ p o r t` is a numerical value. For our example, leave this section blank.
6. Selecting the **Override** option also exposes **Prioritize attribute order**, which defines a hierarchical order of system facts, and **Specify matchers**. The matcher-value combinations determine the right parameter to use depending on an evaluation of the system facts. For our example, leave these sections with the default settings.
7. Click **Submit**.

We now have an override value for the Smart Class Parameter `ht t p d _ p o r t`.

3.9. 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 3.8, “Configuring Smart Class Parameters from Puppet Classes”](#).

Before Smart Class Parameters were introduced, users who wanted to override a parameter were 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 module 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 **Key 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. 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.

4.1. 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/puppet/puppet.conf` configuration file contains the following `[master]` section:

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

  manifest          =
/etc/puppet/environments/$environment/manifests/site.pp
  modulepath        = /etc/puppet/environments/$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.

4.2. 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/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 5. 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.

5.1. 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. For example, modify the `httpd_port` from the `mymodule` to set your own custom port.
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/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/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 6 Tools RPMs repository.
- Installs the Puppet configuration snippet to the system at **/etc/puppet/puppet.conf**.
- Enables the Puppet service to run on the system.
- Run Puppet for the first time and apply the system configuration.

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.

5.2. 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 a Host](#) in the *Red Hat Satellite Host Configuration Guide*



IMPORTANT

The **puppet** package is part of the Red Hat Satellite 6 Tools 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/puppet/puppet.conf` file:

```
# vi /etc/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 configuration. 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.

- 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 6. 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](#) in the *Red Hat Satellite Host Configuration Guide* and [Creating a Host Group on Satellite Server](#) in the *Red Hat Satellite Provisioning Guide*

CHAPTER 7. 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.