DevOps

Puppet For Configuration Mgmt(2)

Trainer: Abhijith V G – AWS, eCommerce, Mobile & DevOps Architect

Magento Certified DEVELOPER PLUS

amazon web services | Certified

Developer - Associate

Puppet: Installation and Configuration - Configuring Puppet Master and Ag

1. Create Repos for Puppet Labs
    1. This is to setup software repositories to get the latest versions
    2. Update the repository
2. Install Puppet-Agent on Nodes
3. Install PuppetServer on the Puppet Server
4. Sign the Node's SSL Certificates on the Puppet Server

Puppet: Installation of Puppet Server and Node(s) using Vagrant

1. git clone https://github.com/schogini/vagrant-puppetserver-ubuntu.git
2. git clone https://github.com/schogini/vagrant-puppetnode-ubuntu.git

Puppet: Puppet Master - Puppet configuration tree

/opt/puppetlabs/puppet/

```
/opt/puppetlabs/puppet/
|-- VERSION
|-- bin
|-- cache
|-- include
|-- lib
|-- modules
|-- share
`-- ssl
```

/etc/puppetlabs/code/

```
/etc/puppetlabs/code/
|-- environments
|   `-- production
|       |-- environment.conf
|       |-- hiera.yaml
|       |-- hieradata
|       |-- manifests
|       |   `-- site.pp
|       `-- modules
|           |-- motd
|           |-- registry
|           `-- stdlib
`-- modules
```

Puppet: Puppet Master - Puppet configuration tree

The main configuration directory is /etc/puppetlabs/puppet

```
/etc/puppetlabs/puppet
|-- auth.conf
|-- hiera.yaml
|-- puppet.conf
`-- ssl
    |-- ca
    |   |-- ca_crl.pem
    |   |-- ca_crt.pem
    |   |-- ca_key.pem
    |   |-- ca_pub.pem
    |   |-- inventory.txt
    |   |-- private
    |   |-- requests
    |   |-- serial
    |   `-- signed
    |       |-- a9678380ab81.pem
    |       |-- puppet.pem
    |       `-- puppetnode1.pem
    |-- certificate_requests
    |-- certs
    |   |-- a9678380ab81.pem
    |   |-- ca.pem
    |   `-- puppet.pem
    |-- crl.pem
    |-- private
    |-- private_keys
    |   |-- a9678380ab81.pem
    |   `-- puppet.pem
    `-- public_keys
        |-- a9678380ab81.pem
        `-- puppet.pem
```

Puppet: Puppet Master - Puppet configuration file

```
root@puppet:/# puppet config print modulepath
/etc/puppetlabs/code/environments/production/modules:
/etc/puppetlabs/code/modules:/opt/puppetlabs/puppet/modules

root@puppet:/# cat /etc/puppetlabs/puppet/puppet.conf
# This file can be used to override the default puppet settings.
# See the following links for more details on what settings are available:
# - https://docs.puppetlabs.com/puppet/latest/reference/config_important_settings.html
# - https://docs.puppetlabs.com/puppet/latest/reference/config_about_settings.html
# - https://docs.puppetlabs.com/puppet/latest/reference/config_file_main.html
# - https://docs.puppetlabs.com/puppet/latest/reference/configuration.html
[master]
vardir = /opt/puppetlabs/server/data/puppetserver
logdir = /var/log/puppetlabs/puppetserver
rundir = /var/run/puppetlabs/puppetserver
pidfile = /var/run/puppetlabs/puppetserver/puppetserver.pid
codedir = /etc/puppetlabs/code
```

Puppet: Puppet Master – Puppet important folders

```
/etc/puppetlabscode                 # Puppet code. Don't modify directly
/etc/puppetlabscodepuppetserver     # PuppetServer configuration
/etc/puppetlabscodepxp-agent        # Orchestration Agent configuration
/etc/puppetlabscodepuppetdb         # PuppetDB configuration
/etc/puppetlabscodepuppet           # Puppet Agent configuration
/etc/puppetlabscodemcollective      # Mcollective configuration

/opt/puppetlabs                     # All the internal Puppet stuff, binaries, etc
/var/log/messages                   # The Puppet Agent logs here on *nix systems
/var/log/puppetlabs                 # All other logging

C:\Program Files\Puppet Labs        # /opt/puppetlabs equivalent
C:\ProgramData\PuppetLabs           # /etc/puppetlabs equivalent
```

Puppet: Puppet Terms

- **Puppet Master**: the master server that controls configuration on the nodes
- **Puppet Agent Node**: a node controlled by a Puppet Master
- **Manifest**: a file that contains a set of instructions to be executed
- **Resource**: a portion of code that declares an element of the system and how its state should be changed. For instance, to install a package we need to define a *package* resource and ensure its state is set to "installed"
- **Module**: a collection of manifests and other related files organized in a pre-defined way to facilitate sharing and reusing parts of a provisioning
- **Class**: just like with regular programming languages, classes are used in Puppet to better organize the provisioning and make it easier to reuse portions of the code
- **Facts**: global variables containing information about the system, like network interfaces and operating system
- **Services**: used to trigger service status changes, like restarting or stopping a service

Puppet provisionings are written using a custom DSL (domain specific language) that is based on Ruby.

Puppet: Resources

Common Syntax

<Resource Type> { '<title>'
    name: '<string>' (optional)
    <attribute name>: <attribute value>
}

- '<Resource Type>' is the type of resource
- '<title>' is a unique name for the resource declaration.
- 'name' is the actual resource (e.g. for example package resource type can have 'name', 'curl', 'wget' etc; for service resource type can have 'name', 'apache2', 'nginx' and so on)
- If 'name' is missing then, title is used.

Both the package resource declarations will install apache2 package.
The first declaration uses the resource 'title' since, 'name' is missing.

```
package { 'apache2'
    ensure => installed
}
```

```
package { 'install-apache'
    name => 'apache2'
    ensure => installed
}
```

Puppet: Resources

Main resources: package, service, file, user, exec

https://docs.puppet.com/puppet/latest/cheatsheet_core_types.html

Inline documentation on types

Types reference documentation

We can check all the types documentation via the command line:

```
puppet describe <type>
```

For a more compact output:

```
puppet describe -s <type>
```

To show all the available resource types:

```
puppet describe -l
```

Inspect existing resource types

To interactively inspect and modify our system's resources

```
puppet resource <type> [name]
```

Remember we can use the same command to CHANGE our resources attibutes:

```
puppet resource <type> <name> [attribute=value] [attribute2=value2]
```

Managing packages

Installation of packages is managed by the **package** type.

The main arguments:

```
package { 'apache':
  name     => 'httpd',   # (namevar)
  ensure   => 'present' # Values: 'absent', 'latest', '2.2.1'
  provider => undef,     # Force an explicit provider
}
```

Managing services

Management of services is via the **service** type.

The main arguments:

```
service { 'apache':
  name       => 'httpd',   # (namevar)
  ensure     => 'running' # Values: 'stopped', 'running'
  enable     => true,      # Define if to enable service at boot (true|false)
  hasstatus  => true,      # Whether to use the init script' status to check
                           # if the service is running.
  pattern    => 'httpd',   # Name of the process to look for when hasstatus=false
}
```

Managing files

Files are the most configured resources on a system, we manage them with the **file** type:

```
file { 'httpd.conf':
    # (namevar) The file path
    path       => '/etc/httpd/conf/httpd.conf',
    # Define the file type and if it should exist:
    # 'present','absent','directory','link'
    ensure     => 'present',
    # Url from where to retrieve the file content
    source     => 'puppet://[puppetfileserver]/<share>/path',
    # Actual content of the file, alternative to source
    # Typically it contains a reference to the template function
    content    => 'My content',
    # Typical file's attributes
    owner      => 'root',
    group      => 'root',
    mode       => '0644',
    # The sylink target, when ensure => link
    target     => '/etc/httpd/httpd.conf',
    # Whether to recursively manage a directory (when ensure => directory)
    recurse    => true,
}
```

Executing commands

We can run plain commands using Puppet's **exec** type. Since Puppet applies it at every run, either the command can be safely run multiple times or we have to use one of the **creates**, **unless**, **onlyif**, **refreshonly** arguments to manage when to execute it.

```
exec { 'get_my_file':
    # (namevar) The command to execute
    command   => "wget http://mysite/myfile.tar.gz -O /tmp/myfile.tar.gz',
    # The search path for the command. Must exist when command is not absolute
    # Often set in Exec resource defaults
    path      => '/sbin:/bin:/usr/sbin:/usr/bin',
    # A file created by the command. It if exists, the command is not executed
    creates   => '/tmp/myfile.tar.gz',
    # A command or an array of commands, if any of them returns an error
    # the command is not executed
    onlyif    => 'ls /tmp/myfile.tar.gz && false',
    # A command or an array of commands, if any of them returns an error
    # the command IS executed
    unless    => 'ls /tmp/myfile.tar.gz',
}
```

Managing users

Puppet has native types to manage users and groups, allowing easy addition, modification and removal. Here are the main arguments of the **user** type:

```
user { 'joe':
    # (namevar) The user name
    name       => 'joe',
    # The user's status: 'present','absent','role'
    ensure     => 'present',
    # The user's  id
    uid        => '1001',
    # The user's primary group id
    gid        => '1001',
    # Eventual user's secondary groups (use array for many)
    groups     => [ 'admins' , 'developers' ],
    # The user's password. As it appears in /etc/shadow
    # Use single quotes to avoid unanted evaluation of $* as variables
    password   => '$6$ZFS5JFFRZc$FFDSvPZSSFGVdXDlHe',
    # Typical users' attributes
    shell      => '/bin/bash',
    home       => '/home/joe',
    mode       => '0644',
}
```

Puppet: Resources List

```
root@puppet:/# puppet resource --types| wc -l
53
root@puppet:/# puppet resource --types|head
anchor
augeas
computer
cron
exec
file
file_line
filebucket
group
host
```

Puppet: Resources Built-In Help

puppet describe <resource name>

```
root@puppet:/etc/puppetlabs/code/environments/production# puppet describe file

file
====
Manages files, including their content, ownership, and permissions.
The `file` type can manage normal files, directories, and symlinks; the
type should be specified in the `ensure` attribute.
File contents can be managed directly with the `content` attribute, or
downloaded from a remote source using the `source` attribute; the latter
can also be used to recursively serve directories (when the `recurse`
attribute is set to `true` or `local`). On Windows, note that file
contents are managed in binary mode; Puppet never automatically translates
line endings.
**Autorequires:** If Puppet is managing the user or group that owns a
file, the file resource will autorequire them. If Puppet is managing any
parent directories of a file, the file resource will autorequire them.


Parameters
----------

- **backup**
    Whether (and how) file content should be backed up before being
    replaced.
    This attribute works best as a resource default in the site manifest
    (`File { backup => main }`), so it can affect all file resources.
```

Puppet: Resources Help Provides Example

```
EXAMPLE
-------
This example uses `puppet resource` to return a Puppet configuration for
the user `luke`:

    $ puppet resource user luke
    user { 'luke':
     home => '/home/luke',
     uid => '100',
     ensure => 'present',
     comment => 'Luke Kanies,,,',
     gid => '1000',
     shell => '/bin/bash',
     groups => ['sysadmin','audio','video','puppet']
    }
```

Puppet: Resource Dependency

When writing manifests, it is important to keep in mind that Puppet doesn't evaluate the resources in the same order they are defined. This is a common source of confusion for those who are getting started with Puppet. Resources must explicitly define dependency between each other, otherwise there's no guarantee of which resource will be evaluated, and consequently executed, first.

As a simple example, let's say you want execute a command, but you need to make sure a dependency is installed first:

```
package { 'python-software-properties':
    ensure => 'installed'
}

exec { 'add-repository':
    command => '/usr/bin/add-apt-repository ppa:ondrej/php5 -y'
    require => Package['python-software-properties']
}
```

The `require` option receives as parameter a reference to another resource. In this case, we are referring to the *Package* resource identified as `python-software-properties`.
It's important to notice that while we use `exec`, `package`, and such for declaring resources (with lowercase), when referring to previously defined resources, we use `Exec`, `Package`, and so on (capitalized).

Puppet: Resource Dependency (continued)

Now let's say you need to make sure a task is executed **before** another. For a case like this, we can use the `before` option instead:

```
package { 'curl':
    ensure => 'installed'
    before => Exec['install script']
}

exec { 'install script':
    command => '/usr/bin/curl http://example.com/some-script.sh'
```

Puppet: Manifests are Puppet Programs

Puppet programs are called manifests. Manifests are composed of puppet code and their filenames use the `.pp` extension. The default main manifest in Puppet installed via apt is `/etc/puppet/manifests/site.pp`.

Puppet: Manifests

Manifests are basically a collection of resource declarations, using the extension `.pp`. Below you can find an example of a simple playbook that performs two tasks: updates the `apt` cache and installs `vim` afterwards:

```
exec { 'apt-get update':
    command => '/usr/bin/apt-get update'
}

package { 'vim':
    ensure => 'installed'
    require => Exec['apt-get update']
}
```

Puppet: Manifests - Variables

Variables can be defined at any point in a manifest. The most common types of variables are strings and arrays of strings, but other types are also supported, such as booleans and hashes.

The example below defines a string variable that is later used inside a resource:

```
$package = "vim"

package { $package:
   ensure => "installed"
}
```

Puppet: Manifests - Loops

Loops are typically used to repeat a task using different input values. For instance, instead of creating 10 tasks for installing 10 different packages, you can create a single task and use a loop to repeat the task with all the different packages you want to install.

The simplest way to repeat a task with different values in Puppet is by using arrays, like in the example below:

```
$packages = ['vim', 'git', 'curl']

package { $packages:
   ensure => "installed"
}
```

As of version 4, Puppet supports additional ways for iterating through tasks. The example below does the same thing as the previous example, but this time using the `each` iterator. This option gives you more flexibility for looping through resource definitions:

```
$packages.each |String $package| {
  package { $package:
    ensure => "installed"
  }
}
```

Puppet: Manifests - Conditionals

Conditionals can be used to dynamically decide whether or not a block of code should be executed, based on a variable or an output from a command, for instance.

Puppet supports most of the conditional structures you can find with traditional programming languages, like `if/else` and `case` statements. Additionally, some resources like `exec` will support attributes that work like a conditional, but only accept a command output as condition.

Let's say you want to execute a command based on a *fact*. In this case, as you want to test the value of a variable, you need to use one of the conditional structures supported, like `if/else`:

```
if $osfamily != 'Debian' {
 warning('This manifest is not supported on this OS.')
}
else {
 notify { 'Good to go!': }
}
```

Puppet: Manifests – Conditionals (continued)

Another common situation is when you want to condition the execution of a command based on the output from another command. For cases like this you can use `onlyif` or `unless`, like in the example below. This command will only be executed when the output from `/bin/which php` is successful, that is, the command exits with status **0**:

```
exec { "Test":
 command => "/bin/echo PHP is installed here > /tmp/test.txt",
 onlyif => "/bin/which php"
}
```

Similarly, `unless` will execute the command all times, except when the command under `unless` exits successfully:

```
exec { "Test":
 command => "/bin/echo PHP is NOT installed here > /tmp/test.txt",
 unless => "/bin/which php"
}
```

Puppet: Manifests - Templates

Templates are typically used to set up configuration files, allowing for the use of variables and other features intended to make these files more versatile and reusable. Puppet supports two different formats for templates: Embedded Puppet (EPP) and Embedded Ruby (ERB). The EPP format, however, works only with recent versions of Puppet (starting from version 4.0).

Below is an example of an ERB template for setting up an Apache virtual host, using a variable for setting up the document root for this host:

```
<VirtualHost *:80>
    ServerAdmin webmaster@localhost
    DocumentRoot <%= @doc_root %>

    <Directory <%= @doc_root %>>
        AllowOverride All
        Require all granted
    </Directory>
</VirtualHost>
```

Puppet: Manifests – Templates (continued)

In order to apply the template, we need to create a `file` resource that renders the template content with the `template` method. This is how you would apply this template to replace the default Apache virtual host:

```
file { "/etc/apache2/sites-available/000-default.conf":
    ensure => "present",
    content => template("apache/vhost.erb")
}
```

Puppet makes a few assumptions when dealing with local files, in order to enforce organization and modularity. In this case, Puppet would look for a `vhost.erb` template file inside a folder `apache/templates`, inside your modules directory.

Puppet: Manifests – Defining and Triggering Services

Service resources are used to make sure services are initialized and enabled. They are also used to trigger service restarts.

Let's take into consideration our previous template usage example, where we set up an Apache virtual host. If you want to make sure Apache is restarted after a virtual host change, you first need to create a *service* resource for the Apache service. This is how such resource is defined in Puppet:

```
service { 'apache2':
    ensure => running,
    enable => true
}
```

Now, when defining the resource, you need to include a `notify` option in order to trigger a restart:

```
file { "/etc/apache2/sites-available/000-default.conf":
    ensure => "present",
    content => template("vhost.erb"),
    notify => Service['apache2']
}
```

Puppet: Classes

In Puppet, classes are code blocks that can be called in a code elsewhere. Using classes allows you reuse Puppet code, and can make reading manifests easier.

Class Definition

A class definition is where the code that composes a class lives. Defining a class makes the class available to be used in manifests, but does not actually evaluate anything.

Here is how a class **definition** is formatted:

```
class example_class {
  ...
  code
  ...
}
```

The above defines a class named "example_class", and the Puppet code would go between the curly braces.

Puppet: Classes – Declaration Normal Style

A class declaration occurs when a class is called in a manifest. A class declaration tells Puppet to evaluate the code within the class. Class declarations come in two different flavors: normal and resource-like.

A **normal class declaration** occurs when the `include` keyword is used in Puppet code, like so:

```
include example_class
```

This will cause Puppet to evaluate the code in *example_class*.

Puppet: Classes – Declaration Resource Style

A **resource-like class declaration** occurs when a class is declared like a resource, like so:

```
class { 'example_class': }
```

Using resource-like class declarations allows you to specify *class parameters*, which override the default values of class attributes.

```
node 'host2' {
  class { 'apache': }              # use apache module
  apache::vhost { 'example.com':   # define vhost resource
    port    => '80',
    docroot => '/var/www/html'
  }
}
```

Puppet: Puppet Language Basics - Features

- Variables
- Data types
- Conditional statements
- Parameters
- Functions
- Etc.

Puppet: Puppet Language Basics – Data Types

- String
- Integer, Float, and Numeric
- Boolean
- Array
- Hash
- Regexp (regular expression)
- Undef (undefined)
- Default

Puppet: Puppet Language Basics – Comparison Operators

==	equality
!=	non-equality
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=~	regex or data type match
!~	regex or data type non-match
in	e.g. 'eat' in 'eaten' resolves to true

Puppet: Puppet Language Basics – Conditionals

if	Execute if expression is true
unless	Execute if expression is false
case	Execute based on value of expression
selector	Return value based on value of expression

Puppet: Puppet Language Basics - Resource Collectors

https://docs.puppet.com/puppet/5.1/lang_collectors.html

Syntax

```
User <| title == 'luke' |> # Will collect a single user resource whose title is 'luke'
User <| groups == 'admin' |> # Will collect any user resource whose list of supplementa
Yumrepo['custom_packages'] -> Package <| tag == 'custom' |> # Will create an order rela
```

Puppet: Puppet Language Basics - Relationships and Ordering

https://docs.puppet.com/puppet/5.2/lang_relationships.html

Syntax: Relationship metaparameters

```
package { 'openssh-server':
  ensure => present,
  before => File['/etc/ssh/sshd_config'],
}
```

Puppet: Puppet Language Basics - Class Parameters

```
# A class with parameters
class apache (String $version = 'latest') {
  package {'httpd':
    ensure => $version, # Using the class parameter from above
    before => File['/etc/httpd.conf'],
  }
  file {'/etc/httpd.conf':
    ensure  => file,
    owner   => 'httpd',
    content => template('apache/httpd.conf.erb'), # Template from a module
  }
  service {'httpd':
    ensure    => running,
    enable    => true,
    subscribe => File['/etc/httpd.conf'],
  }
}
```

How does Puppet know about your system ?

- Using the Ruby library Facter
- Facter supports a large numbers of predefined facts
- Customs facts can be defined

```
# facter
architecture => x86_64
bios_vendor => Seabios
bios_version => 0.5.1
blockdevices => vda,vdb
interfaces => eth0,lo
ipaddress => 172.16.27.44
ipaddress_eth0 => 172.16.27.44
is_virtual => true
kernel => Linux
kernelmajversion => 2.6
kernelrelease => 2.6.32-431.el6.x86_64
kernelversion => 2.6.32
etc, ...
```

Facts and Built-In Variables

Puppet has many built-in variables that you can use in your manifests. For a list of these, see the page on facts and built-in variables.

```
if $facts['os']['family'] == 'redhat' {
  # ...
}
```

https://docs.puppet.com/puppet/5.1/lang_facts_and_builtin_vars.html

Assignment

```
$content = "some content\n"
```

Resolution

```
file {'/tmp/testing':
  ensure  => file,
  content => $content,
}

$address_array = [$address1, $address2, $address3]
```

Puppet: Puppet Language Advanced - Conditional statement

An “unless” statement:

```
unless $facts['memory']['system']['totalbytes'] > 1073741824 {
  $maxclient = 500
}
```

Puppet: Puppet Language Advanced - Conditional statement

A case statement:

```
case $facts['os']['name'] {
  'Solaris':           { include role::solaris }
  'RedHat', 'CentOS':  { include role::redhat  }
  /^(Debian|Ubuntu)$/: { include role::debian  }
  default:             { include role::generic }
}
```

Puppet: Puppet Language Advanced - Conditional statement

A selector:

```
$rootgroup = $facts['os']['family'] ? {
  'Solaris'           => 'wheel',
  /(Darwin|FreeBSD)/ => 'wheel',
  default             => 'root',
}

file { '/etc/passwd':
  ensure => file,
  owner  => 'root',
  group  => $rootgroup,
}
```

Puppet: Puppet Language Advanced - If Else

```
if $facts['is_virtual'] {
  # Our NTP module is not supported on virtual machines:
  warning('Tried to include class ntp on virtual machine; this node might b
}
elsif $facts['os']['name'] == 'Darwin' {
  warning('This NTP module does not yet work on our Mac laptops.')
}
else {
  # Normal node, include the class.
  include ntp
}
```

REGEX

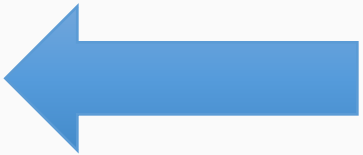
```
if $trusted['certname'] =~ /^www(\d+)\./ {
  notice("Welcome to web server number $1.")
}
```

Puppet: Templates (EPP – Embedded Puppet/ERB – Embedded Ruby)

https://docs.puppet.com/puppet/5.1/lang_template_epp.html
https://docs.puppet.com/puppet/5.1/lang_template_erb.html

EPP

- Stands for "Embedded Puppet"
- Language is consistent with Puppet
- Supports parameters



ERB

- Stands for "Embedded Ruby"
- Language is different from Puppet
- Does not support parameters

Puppet: Templates (EPP – Embedded Puppet/ERB – Embedded Ruby)

Template Language	File	String
Embedded Puppet (EPP)	`epp`	`inline_epp`
Embedded Ruby (ERB)	`template`	`inline_template`

With a template file: `template` and `epp`

You can put template files in the `templates` directory of a module. EPP files should have the `.epp` extension, and ERB files should have the `.erb` extension.

To use a template file, evaluate it with the `template` (ERB) or `epp` function:

```
# epp(<FILE REFERENCE>, [<PARAMETER HASH>])
file { '/etc/ntp.conf':
  ensure  => file,
  content => epp('ntp/ntp.conf.epp', {'service_name' => 'xntpd', 'iburst_enable' => tru
  # Loads /etc/puppetlabs/code/environments/production/modules/ntp/templates/ntp.conf.e
}

# template(<FILE REFERENCE>, [<ADDITIONAL FILES>, ...])
file { '/etc/ntp.conf':
  ensure  => file,
  content => template('ntp/ntp.conf.erb'),
  # Loads /etc/puppetlabs/code/environments/production/modules/ntp/templates/ntp.conf.e
}
```

Puppet: Templates

- Expression-printing tags

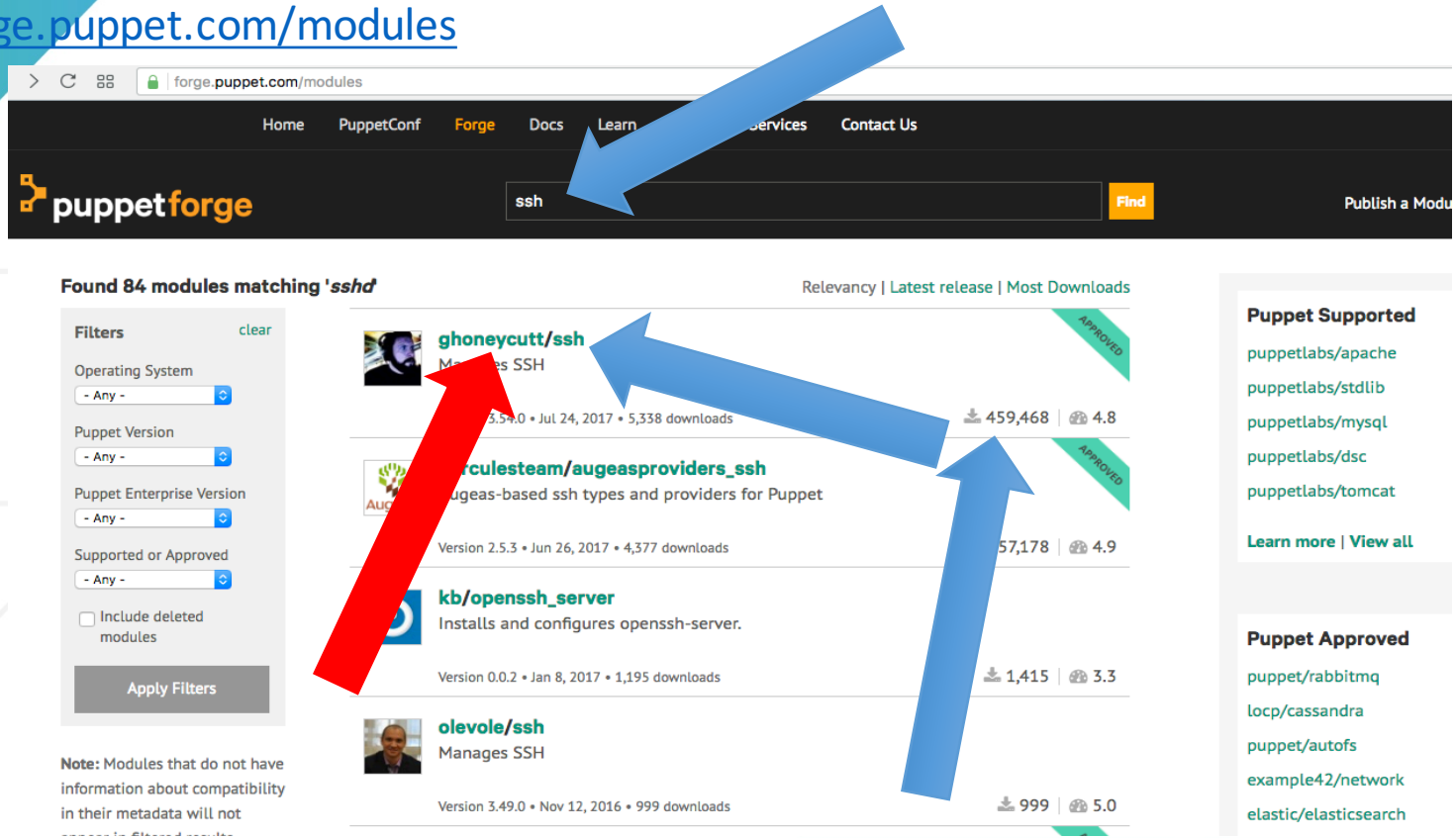
```
<%= EXPRESSION %>
```

- Non-printing tags

```
<% EXPRESSION %>
```

Puppet: Example Code Manifests/Modules - NTP Module

https://forge.puppet.com/modules



Puppet: Example Code Manifests/Modules - NTP Module

https://forge.puppet.com/puppetlabs/ntp/6.0.0

puppetlabs/ntp

by: Puppet Project URL Report issues RSS Feed

Installs, configures, and manages the NTP service.

This module has been optimized for Puppet 4. Since version 6.0.0, this module takes advantage of the latest Puppet 4 features. For more information on upgrading, check out why you should upgrade to Puppet 4.

Latest version is compatible with:

- Puppet Enterprise 2017.2.x, 2017.1.x, 2016.5.x, 2016.4.x, 2016.2.x
- Puppet >= 4.5.0 < 5.0.0
- RedHat, Ubuntu, Debian, Solaris, SLES, Scientific, CentOS, OracleLinux, AIX, Fedora, Amazon, Archlinux

Tags: debian, ubuntu, ntp, time, rhel, ntpd, archlinux, centos, gentoo, aix, ntpserver

To use this module, add this declaration to your Puppetfile:

```
mod 'puppetlabs-ntp', '6.0.0'
```

Learn more about managing modules with a Puppetfile download latest tar.gz

18,324,039
Latest version: 43,972

To manually install this module with puppet module tool:

```
puppet module install puppetlabs-ntp --version 6.0.0
```

Puppet: Example Code Manifests/Modules - NTP Module

https://forge.puppet.com/puppetlabs/ntp Module description

The ntp module installs, configures, and manages the NTP service across a range of OS

Beginning with ntp

`include '::ntp'` is enough to get you up and running. If you wish to pass in parameters specifying which servers to use, then:

```
class { '::ntp':
  servers => [ 'ntp1.corp.com', 'ntp2.corp.com' ],
}
```

Usage

All interaction with the ntp module can be done through the main ntp class. This means you can simply toggle the options in `::ntp` to have full functionality of the module.

I just want NTP, what's the minimum I need?

```
include '::ntp'
```

I just want to tweak the servers, nothing else.

```
class { '::ntp':
  servers => [ 'ntp1.corp.com', 'ntp2.corp.com' ],
}
```

Puppet: Example Code Manifests/Modules - Sudo

https://forge.puppet.com/abstractit/sudo

puppet module install abstractit-sudo --version 1.2.2

Usage

```
sudo::rule { "extra_rule":
  ensure   => present,
  who      => 'bob',
  runas    => 'root', # default runas user is root, please change to override.
  commands => "/usr/sbin/systemctl",
  nopass   => false, #
  setenv   => false,
  comment  => "what ever you like",
}
```

Puppet language style

Puppet language as an official Style Guide which defines recommended rules on how to write the code.

The standard de facto tool to check the code style is puppet-lint.

Badly formatted example:

```
file {
 "/etc/issue":
      content => "Welcome to $fqdn",
      ensure => present,
       mode => 644,
      group => "root",
      path => "${issue_file_path}",
      }
```

Corrected style:

https://puppet.com/docs/puppet/5.3/style_guide.html

```
file { '/etc/issue':
  ensure  => present,
  content => "Welcome to ${fqdn}",
  mode    => 0644,
  group   => 'root',
  path    => $issue_file_path,
}
```

Puppet: Puppet Language Basics - Manifests

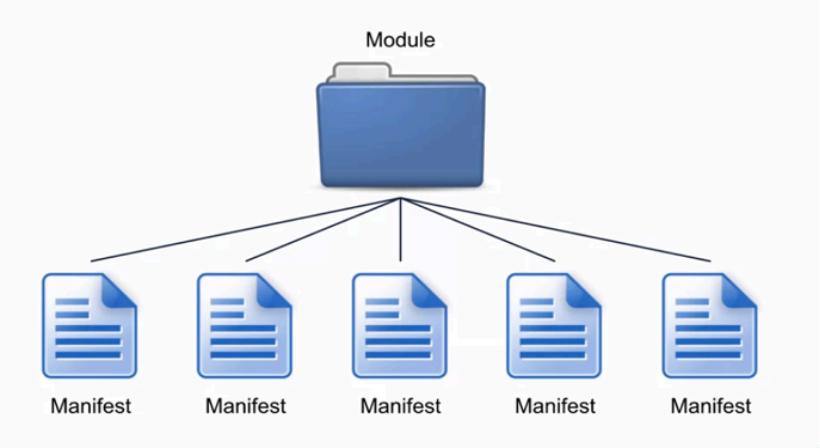
A manifest in Puppet can have one or more classes

Puppet: Modules (can have one or more manifests)

A module is a collection of manifests and data (such as facts, files, and templates), and they have a specific directory structure. Modules are useful for organizing your Puppet code, because they allow you to split your code into multiple manifests. It is considered best practice to use modules to organize almost all of your Puppet manifests.

To add a module to Puppet, place it in

/etc/puppetlabs/code/environments/production/modules/



Puppet: Manifests – Coding a Complete Manifest

cd to production folder

nano manifests/web.pp

Puppet: Manifests – Example 1 (Install Apache+PHP via a Single Manifest)

```
exec {'apt-update':
  command => '/usr/bin/apt-get update'
}

package {'apache2':
  require => Exec['apt-update'],
  ensure => installed,
}

service {'apache2':
  ensure => running
}

package {'php5':
  require => Exec['apt-update'],
  ensure => installed,
}

package {'libapache2-mod-php5':
  ensure => installed,
}

file {'/etc/apache2/mods-available/php5.conf':
  content => '<IfModule mod_php5.c>
        <FilesMatch "\.php$">
            SetHandler application/x-httpd-php
        </FilesMatch>
</IfModule>
',
  require => Package['apache2'],
  notify => Service['apache2'],
}

file {'/var/www/html/i.php':
  ensure => 'file',
  content => "<?php phpinfo(); ?>",
  require => Package['apache2'],
}
```

cd to production folder
nano manifests/web.pp

puppet apply -t manifests/web.pp

curl localhost/i.php

Puppet: Module – Example 2 (Install Apache+PHP using a New Module)

```
root@puppet:/etc/puppetlabs/code/environments/production/modules# puppet module generate --skip-interview sree-web
```

```
root@puppet:/etc/puppetlabs/code/environments/production/modules# tree web
web
|-- Gemfile
|-- README.md
|-- Rakefile
|-- examples
|   `-- init.pp
|-- manifests
|   `-- init.pp
|-- metadata.json
`-- spec
    |-- classes
    |   `-- init_spec.rb
    `-- spec_helper.rb
```

Puppet: Module – Example 2 (Install Apache+PHP using a New Module)

```
class web {

exec { 'apt-update':
         command => '/usr/bin/apt-get update'
       }

       package { 'apache2':
         require => Exec['apt-update'],
         ensure => installed,
       }

       service {'apache2':
         ensure => running,
       }

       package {'php5':
         require => Exec['apt-update'],
         ensure => installed,
       }

       package {'libapache2-mod-php5':
          ensure => installed,
       }

       file {'/etc/apache2/mods-available/php5.conf':
           content => '<IfModule mod_php5.c>
           <FilesMatch "\.php$">
               SetHandler application/x-httpd-php
           </FilesMatch>
       </IfModule>',
         require => Package['apache2'],
         notify => Service['apache2'],
       }

       file {'/var/www/i.php':
         ensure => file,
         content => "<?php echo 7+3; ?><br>\n",
         require => Package['apache2'],
       }


}
```

```
puppet apply -e "include web"
```

```
curl localhost/i.php
```

Puppet: Module – Example 2 (Install Apache+PHP using a New Module)

Here is the output of the web sever with php calculated in this case 7+3

```
root@puppet:/etc/puppetlabs/code/environments/production/modules# puppet apply -e "include web"
Notice: Compiled catalog for puppet in environment production in 0.59 seconds
Notice: /Stage[main]/Web/Exec[apt-update]/returns: executed successfully
Notice: Applied catalog in 30.11 seconds
root@puppet:/etc/puppetlabs/code/environments/production/modules# curl localhost/i.php
10<br>
root@puppet:/etc/puppetlabs/code/environments/production/modules#
```

Puppet: Applying a Module to Nodes

https://docs.puppet.com/puppet/5.1/lang_node_definitions.html

Syntax

```
# <ENVIRONMENTS DIRECTORY>/<ENVIRONMENT>/manifests/site.pp
node 'www1.example.com' {
  include common
  include apache
  include squid
}
node 'db1.example.com' {
  include common
  include mysql
}
```

Puppet: Applying a Module to Nodes

https://docs.puppet.com/puppet/5.1/lang_node_definitions.html

Multiple names

You can use a comma-separated list of names to create a group of nodes with a single node statement:

```
node 'www1.example.com', 'www2.example.com', 'www3.example.com' {
  include common
  include apache, squid
}
```

This example creates three identical nodes: `www1.example.com`, `www2.example.com`, and `www3.example.com`.

Puppet: **Applying a Module to Nodes**

https://docs.puppet.com/puppet/5.1/lang_node_definitions.html

Regular expression names

Regular expressions (regexes) can be used as node names. This is another method for writing a single node statement that matches multiple nodes.

> **Note:** Make sure all of your node regexes match non-overlapping sets of node names. If a node's name matches more than one regex, Puppet makes no guarantee about which matching definition it will get.

```
node /^www\d+$/ {
  include common
}
```

The above example would match `www1`, `www13`, and any other node whose name consisted of `www` and one or more digits.

Puppet: Applying a Module to Nodes

Edit the main manifest site.php

```
root@puppet:/etc/puppetlabs/code/environments/production# cat  manifests/site.pp
node 'puppetnode1' {
 include motd
 include web
}

node 'puppet' {
 include motd
 include web
}

node default {
}
```

```
root@puppetnode1:/# puppet agent -t
```

```
curl localhost/i.php
```

Puppet: Puppet Master - Puppet module structure

https://docs.puppet.com/puppet/5.1/modules_fundamentals.html

```
/etc/puppetlabs/code/environments/production/modules/motd
├── CHANGELOG.md
├── checksums.json
├── Gemfile
├── LICENSE
├── manifests
│   └── init.pp
├── metadata.json
├── Rakefile
├── README.md
├── spec
│   ├── acceptance
│   │   ├── motd_spec.rb
│   │   └── nodesets
│   │       ├── default.yml
│   │       └── ubuntu.yml
│   ├── classes
│   │   └── motd_spec.rb
│   ├── spec_helper_acceptance.rb
│   ├── spec_helper.rb
│   └── spec.opts
├── templates
│   ├── motd.erb
│   └── spec.erb
```

Puppet: Puppet Extra Points to note

You don't revert actions Puppet has carried out by removing the resource. By removing the resource from your manifests, you're only telling Puppet not to manage it (anymore).

Also, Puppet simply doesn't remember/know about the previous states, so it can't revert to that. It just tries to change the system to a state you define in the manifests. One way to revert is:

```
file {
   '/etc/rsyslog.d/60-custconfig.conf':
   ensure => absent,
}
```

You have to explicitly write code that will "undo" any specific actions that you want to revert.

For example, if you have a my_apache module that install a few packages, configures several files, and ensures a certain state, you will have to write another class inside this module, lets call it my_apache::uninstall, that undoes each of the things that your other module did. The undo class does not have to be inside, or part of the original module.

It can be a completely separate class.

```
file {'remove_directory':
   ensure => absent,
   path => '/your/directory',
   recurse => true,
   purge => true,
   force => true,
  }
```

Be careful. This will remove the whole directory.

Puppet: Puppet Extra Points to note

Connect client (node) to server

Once we can ping each server make sure the Puppet Master is running

puppet agent --server puppetmaster --waitforcert 60 --test

You should see:

info: Creating a new SSL certificate request for puppetclient

Now on the server we can see the request that has come in with

puppet cert --list

You should see '*puppetclient*'. Now we can sign the certificate with

puppet cert --sign puppetclient

Hostname should match the server certificate. If it does not then, you can change it by setting this explicitly in */etc/puppetlabs/puppet/puppet.conf*

[master]

***certname**=puppetmaster*

Now you should have both servers talking to each other!

Puppet: Puppet Extra Points to note

Validate your manifest file:
puppet parser validate init.pp

Manually start/enable a service type resource (on the nodes)
puppet resource service <NAME> ensure=running enable=true

Show puppet configuration values
puppet apply --configprint all ➔ Show all configuration values
puppet apply --configprint manifest ➔ Show the configuration value for the key 'manifest'
puppet config print modulepath --section master --environment production ➔ Show the configuration value for the key 'modulepath' under the section master and environment production

Creating and assigning environments and adding nodes to them.
https://docs.puppet.com/puppet/5.2/environments_creating.html
Puppet uses four config sections:
https://docs.puppet.com/puppet/5.1/config_file_main.html
main is the global section used by all commands and services. It can be overridden by the other sections. master is used by the Puppet master service and the Puppet cert command, agent is used by Puppet agents, last is user.

Puppet: Steps to disconnect a node

1. SERVER: puppet cert clean puppetnode1
2. NODE: find /etc/puppetlabs/puppet/ssl –name puppetnode1.pem –delete

Now, the node will need to make a certificate signing request again when the puppet agent is executed.

Puppet: HW-2

1. Install Apache Webserver + PHP
2. Use site.pp and puppet agent to deploy
3. Build and experiment with custom and official modules


Cognixia
THANK YOU
average 45%