zPuppet is not just limited to the default resources provided with it. It is highly customizable and provides methods to configure resources to meet organizational needs. In this section, we will learn about tools to manage dependencies, work with custom modules, facts and types. We will also learn to implement orchestration using mcollective and Hiera for data management.

# 

# Using r10k and librarian

We will learn to use r10k and librarian to manage module dependencies

in our Project and how they can be useful to develop the Puppet infrastructure for MPLI Productions.

# Working with public modules

We will learn how to use already developed public modules and

reuse them for tasks in MPLI Productions.

Public modules are listed on the forge, there are a number of puppet supported ones. These are actively maintained by puppet.

Puppet approved have been reviewed by puppet and are endorsed but still maintained outside of puppet.

Click through forge to supported, listed by top downloads, stdlib will always be on the top because it's referenced by many other modules. Parts of stdlib have been moved into the core of puppet.

To use stdlib, just make sure it is in your load path, there isn't anything in the manifest

file\_line {'no root login':

ensure => present,

line => 'PermitRootLogin no',

match => '^PermitRootLogin\s+',

path => '/tmp/etc/ssh/sshd\_config',

}

file\_line {'nothing':

ensure => absent,

match => "^Port",

line => 'Port',

path => '/tmp/etc/ssh/sshd\_config',

match\_for\_absence => true,

multiple => true,

replace => false,

}

This function of stdlib shows how useful it is to be able to manage parts of a file but not the whole file, there are a few other modules on the forge that have this ability. We'll look at two of them now.

Next concat

puppet module install puppetlabs-concat

concat {'bashrc':

path => '/tmp/etc/bashrc',

}

$local = @(LOCAL/L)

export PATH=$PATH:/usr/local/bin:/usr/local/sbin

| LOCAL

concat::fragment {'local':

target => 'bashrc',

order => '001',

content => $local,

}

$umask = @(UMASK/L)

if [ $UID -gt 199 ] && [ "`id -gn`" = "`id -un`" ]; then

umask 002

else

umask 022

fi

| UMASK

concat::fragment {'umask':

target => 'bashrc',

order => '002',

content => $umask,

}

Then inifile

puppet module install puppetlabs-inifile

ini\_setting {'puppet-server':

section => 'main',

path => '/home/thomas/.puppet/puppet.conf',

setting => 'server',

value => 'puppet.mpli.packtpub.com',

}

ini\_setting {'puppet-server':

section => 'main',

path => '/home/thomas/.puppet/puppet.conf',

setting => 'server',

value => 'puppet.mpli.packtpub.com',

}

ini\_subsetting {'puppet-reports-store':

section => 'agent',

path => '/home/thomas/.puppet/puppet.conf',

setting => 'reports',

subsetting\_separator => ',',

subsetting => 'store',

}

ini\_subsetting {'puppet-reports-logstash':

section => 'agent',

path => '/home/thomas/.puppet/puppet.conf',

setting => 'reports',

subsetting\_separator => ',',

subsetting => 'logstash',

}

# Writing custom facts

There are two types of custom facts, the facts that are written in Ruby and synced as part of the pluginsync mechanism. These are placed in /opt/puppetlabs/puppet/cache/lib/facter. The other form of facts are much simpler to write, they are just scripts, text files or yaml files placed in the /etc/facter/facts.d directory or the /opt/puppetlabs/facter/facts.d directory.

When you are writing facts, know that the custom facts inside modules, placed in the lib/facter subdirectory of the module, are available on the first puppet run. This makes them very useful if you are using the custom fact value in your conditional statements.

We'll start by making a simple custom fact.

~/.puppetlabs/opt/puppet/cache/lib/facter/test.rb

Facter.add('test') do

setcode do

42

end

end

The interesting thing is that we can use this in a manifest right away.

if $::test == 42 {

notify {"so long and thanks for all the fish": }

} else {

notify {"Don't Panic!": }

}

You can also access facts in your facter code:

require 'ipaddr'

def getRegion(ip)

regions = {

'local' => IPAddr.new('192.168.0.0/16'),

'remote' => IPAddr.new('10.0.0.0/8'),

}

for zone in regions.keys do

if regions[zone].include?(ip)

return zone

end

end

return 'undefined'

end

ip = IPAddr.new(Facter.value('ipaddress'))

Facter.add('region') do

setcode do

getRegion(ip)

end

end

We've used the value of ipaddress to calculate another fact which we can then use in our manifests.

to have this synced down to our systems, we would place this code in the lib/facter subdirectory of a module.

The module need only be in your modulepath for the custom fact to be available to your nodes. The module itself does not need to be included in your catalog.

The last type of custom fact we'll talk about is the one created from the facts.d directories.

These are very simple to write and use, but are not available on the initial puppet run (unless they are placed there by some other means before the puppet run, this is also a useful paradigm is to place files on there during your provisioning process.

cd ~/.facter/facts.d

test.yaml

---

one=two

test.txt

two=three

test.sh

three=four

chmod 755 test.sh

./test.sh

# Writing custom types

custom types are used to extend puppet to deal with new resource types.

There are again two solutions to this problem. The easiest method is to use defined types, which are created from the define function. This can be used to simplify your code and make it more supportable by making reusable sections of code. The argument is the same as that of using subroutines in any programming language.

We'll start with a simple define that creates a notify resource

define greet {

notify { "Hi $title!": }

}

greet { "Ricky": }

greet { "Julian": }

Now you can add a file resource.

file {"/tmp/$title":

content => @("GREET")

Greetings $title,

First, I must solicit your strictest confidence in this transaction. this is by virtue of its nature as being utterly confidential and 'top secret'. I am sure and have confidence of your ability and reliability to prosecute a transaction of this great magnitude involving a pending transaction requiring maximum confidence.

|GREET

}

}

The next type of custom type is done by extending puppet, we'll make a module to do this, and allow puppet to utilize the pluginsync method to stick the files in the appropriate place.

puppet module generate gitrepo

Puppet::Type.newtype(:gitrepo) do

ensurable

newparam(:source) do

isnamevar

end

newparam(:path)

end

--

require 'fileutils'

Puppet::Type.type(:gitrepo).provide(:git) do

commands :git => "git"

def create

git "clone", resource[:source], resource[:path]

end

def exists?

File.directory? resource[:path]

end

def destroy

FileUtils.rm\_r resource[:path]

end

end

now we'll use plugin download to get the files in the right places.

# Understanding and using Hiera

We will understand what is Hiera and use it for data management for MPLI Productions Puppet environment.

# Orchestrating with mcollective

We will configure mcollective on the Puppet client nodes in MPLI

Productions so that we can run remote commands and modules on the nodes using the same.