

Finish a Puppet deployment | Qwiklabs

Qwiklabs

17-21 minutes

Introduction

You want to automatically manage the computers in your company's fleet, including a number of different machines with different operating systems. You've decided to use Puppet to automate the configurations on these machines. Part of the setup is already done, but there are more rules that need to be added, and more operating systems to consider.

Prerequisite

Understands the basics of Puppet, including:

- How to create Puppet classes and rules
- How Puppet interacts with different OSs
- How to use the DSL to create complex rules

You'll have 90 minutes to complete this lab.

Puppet rules

The goal of this exercise is for you to see what Puppet looks like in action. During this lab, you'll be connecting to two different VMs. The VM named **puppet** is the Puppet Master that has the Puppet rules that you'll need to edit. The VM named **linux-instance** is a client VM that you'll use to test that your catalog was applied successfully.

The manifests used for the production environment are located in the directory `/etc/puppet/code/environments/production/manifests`, which contains a `site.pp` file with the node definitions that will be used for this deployment. On top of that, the `modules` directory contains a bunch of modules that are already in use. You'll be extending the code of this deployment to add more functionality to it.

Install packages

There's a module named **packages** on the **Puppet VM instance** that takes care of installing the packages that are needed on the machines in the fleet. Use the command to visit the module:

```
cd /etc/puppet/code/environments/production/modules/packages
```

This module already has a resource entry specifying that **python-requests** is installed on all machines. You can see the `init.pp` file using the **cat** command on the **Puppet VM instance**.

```
cat manifests/init.pp
```

Output:

```
student-02-1c5894ca6729@puppet:~$ cat /etc/puppet/code/environments/production/modules/packages/manifests/init.pp
class packages {
    package { ['python-requests']:
        ensure => installed,
    }
}
```

Now, add an additional resource in the same `init.pp` file within the path `/etc/puppet/code/environments/production/modules/packages`, ensuring the **golang** package gets installed on all machines that belong to the Debian family of operating systems (which includes Debian, Ubuntu, LinuxMint, and a bunch of others).

This resource will be very similar to the previous **python-requests** one. Add edit permission to the file before moving forward using:

```
sudo chmod 646 manifests/init.pp
```

To install the package on Debian systems only, you'll need to use the **os family** fact, like this:

```
if $facts[os][family] == "Debian" {
    # Resource entry to install golang package
}
```

Now, open the file using nano editor and add the resource entry specifying golang package to be installed on all machines of Debian family after the previous resource entry.

The snippet would now look like this:

```
if $facts[os][family] == "Debian" {
    package { ['golang']:
        ensure => installed,
    }
}
```

The complete **init.pp** file would now look similar to the below file:

```
class packages {
    package { ['python-requests']:
        ensure => installed,
    }
}
```

```

    if $facts[os][family] == "Debian" {
        package { 'golang':
            ensure => installed,
        }
    }
}

```

After this, we will also need to ensure that the **nodejs** package is installed on machines that belong to the RedHat family. Refer to the below snippet for this.

```

if $facts[os][family] == "RedHat" {
    #Resource entry
}

```

Complete the above snippet just like the previous one.

The complete **init.pp** file should now look like this:

```

class packages {
    package { 'python-requests':
        ensure => installed,
    }
    if $facts[os][family] == "Debian" {
        package { 'golang':
            ensure => installed,
        }
    }
    if $facts[os][family] == "RedHat" {
        package { 'nodejs':
            ensure => installed,
        }
    }
}

```

Once you've edited the file and added the necessary resources, you'll want to check that the rules work successfully. We can do this by connecting to another machine in the network and verifying that the right packages are installed.

We will be connecting to **linux-instance** using its external IP address. To fetch the external IP address of **linux-instance**, use the following command:

```

gcloud compute instances describe linux-instance --zone=us-central1-a
--format='get(networkInterfaces[0].accessConfigs[0].natIP) '

```

This command outputs the external IP address of **linux-instance**. Copy the **linux-instance** external IP address, open another terminal and connect to it. Follow the instructions given in

the section Accessing the virtual machine by clicking on Accessing the virtual machine from the navigation pane at the right side.

Now manually run the Puppet client on your **linux-instance** VM instance terminal:

```
sudo puppet agent -v --test
```

This command should run successfully and the catalog should be applied.

Output:

```
gcpstaging100622_student@linux-instance:~$ sudo puppet agent -v --test
Info: Using configured environment 'production'
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for linux-instance.c.qwiklabs-gcp-ff98c2c54ac75e3a.internal
Info: Applying configuration version '1578928759'
Notice: /Stage[main]/Packages/Package[golang]/ensure: created
Notice: Applied catalog in 22.66 seconds
```

Now verify whether the **golang** package was installed on this instance. This being an machine of the Debian family should have golang installed. Use the following command to verify this:

```
apt policy golang
```

Output:

```
gcpstaging100622_student@linux-instance:~$ apt policy golang
golang:
  Installed: 2:1.7~5
  Candidate: 2:1.7~5
  Version table:
     2:1.11~1~bpo9+1 100
        100 http://deb.debian.org/debian stretch-backports/main amd64 Packages
*** 2:1.7~5 500
        500 http://deb.debian.org/debian stretch/main amd64 Packages
        100 /var/lib/dpkg/status
```

With this, you've seen how you can use Puppet's facts and package resources to install specific packages on machines within your fleet.

Fetch machine information

It's now time to navigate to the **machine_info** module in our Puppet environment. In the **Puppet VM terminal**, navigate to the module using the following command:

```
cd /etc/puppet/code/environments/production/modules/machine_info
```

The **machine_info** module gathers some information from the machine using **Puppet** facts and then stores it in a file. Currently, the module is always storing this information in `/tmp/machine_info`.

Let's check this out:

```
cat manifests/init.pp
```

Output:

```
student-04-107cacf4ec98@puppet:/etc/puppet/code/environments/production/modules/machine_info$ cat manifests/init.pp
class machine_info {
    file { ['/tmp/machine_info.txt':
        content => template('machine_info/info.erb'),
    ]
}
```

You can view the path in the above file. This path doesn't work for Windows machines. So, you need to adapt this rule for Windows.

Add edit permission to the file using the following command before we adapt the rule.

```
sudo chmod 646 manifests/init.pp
```

Now we will be using `$facts[kernel]` fact to check if the kernel is "windows". If so, set a **\$info_path** variable to `"C:\Windows\Temp\Machine_Info.txt"`, otherwise set it to `"/tmp/machine_info.txt"`. To do this, open the file using nano editor and add the below rule after the default path within the class `machine_info`.

```
if $facts[kernel] == "windows" {
    $info_path = "C:\Windows\Temp\Machine_Info.txt"
} else {
    $info_path = "/tmp/machine_info.txt"
}
```

The file should now look similar to:

```
class machine_info {
    file { ['/tmp/machine_info.txt':
        content => template('machine_info/info.erb'),
    ]
    if $facts[kernel] == "windows" {
        $info_path = "C:\Windows\Temp\Machine_Info.txt"
    } else {
        $info_path = "/tmp/machine_info.txt"
    }
}
```

By default the file resources are stored in the path defined in the name of the resource (the string in the first line) within the class. We can also set different paths, by setting the path attribute.

We will now be renaming the resource to "machine_info" and then use the variable in the path attribute. The variable we are using to store the path in the above rule is **\$info_path**.

Remove the following part from the file **manifests/init.pp**.

```
file { ['/tmp/machine_info.txt':  
    content => template('machine_info/info.erb'),  
}
```

And add the following resource after the rule within the class definition:

```
file { 'machine_info':  
    path => $info_path,  
    content => template('machine_info/info.erb'),  
}
```

The complete manifests/init.pp file should now look like this:

```
class machine_info {  
    if $facts[kernel] == "windows" {  
        $info_path = "C:\Windows\Temp\Machine_Info.txt"  
    } else {  
        $info_path = "/tmp/machine_info.txt"  
    }  
    file { 'machine_info':  
        path => $info_path,  
        content => template('machine_info/info.erb'),  
    }  
}
```

Puppet Templates

Templates are documents that combine code, data, and literal text to produce a final rendered output. The goal of a template is to manage a complicated piece of text with simple inputs.

In Puppet, you'll usually use templates to manage the content of configuration files (via the content attribute of the file resource type).

Templates are written in a templating language, which is specialized for generating text from data. Puppet supports two templating languages:

- **Embedded Puppet (EPP)** uses Puppet expressions in special tags. It's easy for any Puppet user to read, but only works with newer Puppet versions. (≥ 4.0 , or late 3.x versions with future parser enabled.)
- **Embedded Ruby (ERB)** uses Ruby code in tags. You need to know a small bit of Ruby to read it, but it works with all Puppet versions.

Now, take a look at the template file using the following command.

```
cat templates/info.erb
```

Puppet templates generally use data taken from Puppet variables. Templates are files that are pre-processed, some values gets replaced with variables. In this case, the file currently includes the values of three facts. We will be adding a new fact in this file now.

Add edit permissions to the file using templates/info.erb using the following command:

```
sudo chmod 646 templates/info.erb
```

Now open the file using nano editor and add the following fact just after the last fact within the file:

```
Network Interfaces: <%= @interfaces %>
```

The template should now look like this:

```
Machine Information
-----
Disks: <%= @disks %>
Memory: <%= @memory %>
Processors: <%= @processors %>
Network Interfaces: <%= @interfaces %>
}
```

To check if this worked correctly, return to **linux-instance** VM terminal and manually run the client on that machine using the following command:

```
sudo puppet agent -v --test
```

This command should run successfully and the catalog should be applied.

Now verify that the **machine_info** file has the required information using:

```
cat /tmp/machine_info.txt
```

Output:

```
student-04-0fd15a076522@linux-instance:~$ cat /tmp/machine_info.txt
Machine Information
-----
Disks:
Memory:
Processors: {"models"=>["Intel(R) Xeon(R) CPU @ 2.30GHz"], "count"=>1, "physicalcount"=>1}
Network Interfaces: eth0,lo
}
```

And with that, you've seen how you can fetch machine information and store it according to the operating system.

Reboot machine

For the last exercise, we will be creating a new module named **reboot**, that checks if a node has been online for more than **30 days**. If so, then reboot the computer.

To do that, you'll start by creating the module directory.

Switch back to **puppet** VM terminal and run the following command:

```
sudo mkdir -p /etc/puppet/code/environments/production/modules/reboot/manifests
```

Go to the manifests/ directory.

```
cd
/etc/puppet/code/environments/production/modules/reboot/manifests
```

Create an **init.pp** file for the reboot module in the manifests/ directory.

```
sudo touch init.pp
```

Open **init.pp** with nano editor using sudo.

```
sudo nano init.pp
```

In this file, you'll start by creating a class called reboot.

The way to reboot a computer depends on the OS that it's running. So, you'll set a variable that has one of the following reboot commands, based on the kernel fact:

- **shutdown /r** on windows
- **shutdown -r now** on Darwin (macOS)
- **reboot** on Linux.

Hence, add the following snippet in the file **init.pp**:

```
class reboot {
  if $facts[kernel] == "windows" {
    $cmd = "shutdown /r"
  } elsif $facts[kernel] == "Darwin" {
```



```

        $cmd = "shutdown -r now"
    } else {
        $cmd = "reboot"
    }
}

```

With this variable defined, we will now define an exec resource that calls the command, but only when the **uptime_days** fact is larger than 30 days.

Add the following snippet after the previous one within the class definition in the file **reboot/manifests/init.pp**:

```

if $facts[uptime_days] > 30 {
    exec { 'reboot':
        command => $cmd,
    }
}

```

The complete **reboot/manifests/init.pp** should now look like this:

```

class reboot {
    if $facts[kernel] == "windows" {
        $cmd = "shutdown /r"
    } elsif $facts[kernel] == "Darwin" {
        $cmd = "shutdown -r now"
    } else {
        $cmd = "reboot"
    }
    if $facts[uptime_days] > 30 {
        exec { 'reboot':
            command => $cmd,
        }
    }
}

```

Finally, to get this module executed, make sure to include it in the site.pp file.

So, edit `/etc/puppet/code/environments/production/manifests/site.pp` using the following command:

```

sudo nano
/etc/puppet/code/environments/production/manifests/site.pp

```

Add an extra line that includes the reboot module.

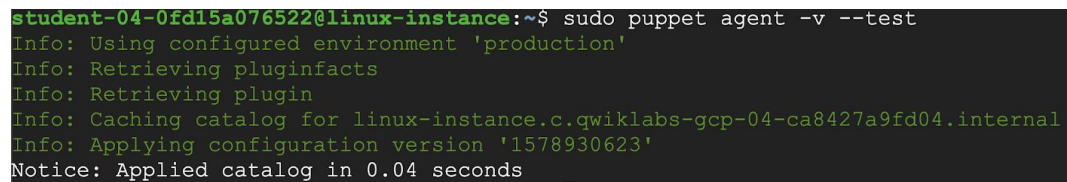
The file `/etc/puppet/code/environments/production/manifests/site.pp` should now look like this:

```
node default {  
  class { 'packages': }  
  class { 'machine_info': }  
  class { 'reboot': }  
}
```

Run the client on **linux-instance** VM terminal:

```
sudo puppet agent -v --test
```

Output:

A terminal window screenshot showing the output of the command 'sudo puppet agent -v --test'. The prompt is 'student-04-0fd15a076522@linux-instance:~\$'. The output consists of several lines: 'Info: Using configured environment 'production'', 'Info: Retrieving pluginfacts', 'Info: Retrieving plugin', 'Info: Caching catalog for linux-instance.c.qwiklabs-gcp-04-ca8427a9fd04.internal', 'Info: Applying configuration version '1578930623'', and 'Notice: Applied catalog in 0.04 seconds'.

```
student-04-0fd15a076522@linux-instance:~$ sudo puppet agent -v --test  
Info: Using configured environment 'production'  
Info: Retrieving pluginfacts  
Info: Retrieving plugin  
Info: Caching catalog for linux-instance.c.qwiklabs-gcp-04-ca8427a9fd04.internal  
Info: Applying configuration version '1578930623'  
Notice: Applied catalog in 0.04 seconds
```

And with that, you've added a whole new module to your deployment!