

Experiment 13 - Puppet Blocks

Roll No.	24
Name	Iyer Sreekesh Subramanian
Class	D15-A
Subject	DevOps Lab
LO Mapped	<p>LO1: To understand the fundamentals of DevOps engineering and be fully proficient with DevOps terminologies, concepts, benefits, and deployment options to meet your business requirements</p> <p>LO6: To Synthesize software configuration and provisioning using Ansible/Puppet.</p>

Aim: To learn Software Configuration Management and provisioning using Puppet Blocks(Manifest, Modules, Classes, Function)

Introduction:

PUPPET MANIFESTS

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.

Classes

In Puppet, classes are code blocks that can be called in a code elsewhere. Using classes allows you to 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.

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

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

Class Declaration

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 flavours: normal and resource-like.

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

This will cause Puppet to evaluate the code in example_class.

```
include example_class
```

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.

Example

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

Modules

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 the `/etc/puppet/modules` directory.

Applying a Manifest

One of Puppet's best features is the ease of testing your code. Puppet does not require you to set up complicated testing environments to evaluate Puppet manifests.

```
puppet apply <name_of_file>.pp
```

Puppet Apply does the following things:

- Builds (compiles) a Puppet catalog from the manifest.
- Uses dependency and ordering information to determine evaluation order.
- Evaluate the target resource to determine if changes should be applied.
- Creates, modifies, or removes the resource—a notification message is created.
- Provides verbose feedback about the catalog application.

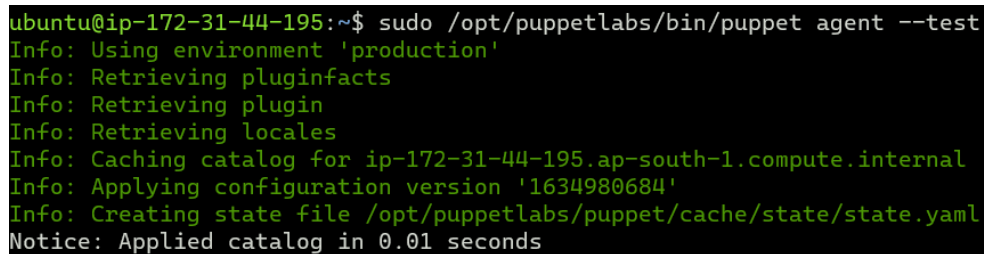
Demo:

To be able to run manifests, you need to have your cluster set up.

Steps:

1. To test your cluster setup, run this command -

```
sudo /opt/puppetlabs/bin/puppet agent --test
```



```
ubuntu@ip-172-31-44-195:~$ sudo /opt/puppetlabs/bin/puppet agent --test
Info: Using environment 'production'
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Retrieving locales
Info: Caching catalog for ip-172-31-44-195.ap-south-1.compute.internal
Info: Applying configuration version '1634980684'
Info: Creating state file /opt/puppetlabs/puppet/cache/state/state.yaml
Notice: Applied catalog in 0.01 seconds
```

If the output is normal, you can proceed.

2. Create new directories

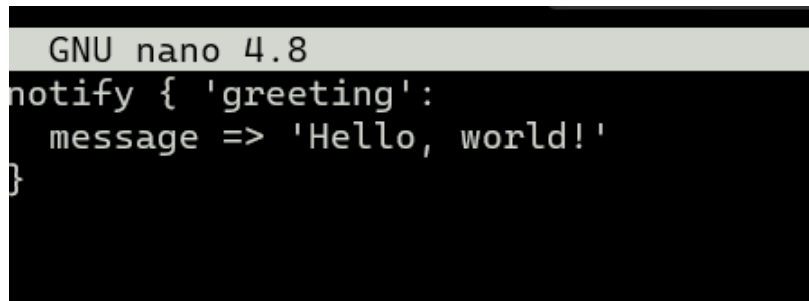
```
mkdir puppet_code
mkdir puppet_code/manifests
```

3. Create a new puppet manifest file and open it using nano.

```
nano /puppet_code/manifests/helloworld.pp
```

4. Enter the following contents inside the file.

```
notify { 'greeting':
  message => 'Hello, world!'
}
```



```
GNU nano 4.8
notify { 'greeting':
  message => 'Hello, world!'
}
```

5. Change directory to /opt/puppetlabs/puppet/bin
cd /opt/puppetlabs/puppet/bin
6. Now, apply the file.

```
./puppet apply /home/ubuntu/manifests/helloworld.pp
```

```
ubuntu@ip-172-31-44-195:/opt/puppetlabs/puppet/bin$ ./puppet apply /home/ubuntu/puppet_code/manifests/helloworld.pp
Notice: Compiled catalog for ip-172-31-44-195.ap-south-1.compute.internal in environment production in 0.01 seconds
Notice: Hello, world!
Notice: /Stage[main]/Main/Notify[greeting]/message: defined 'message' as 'Hello, world!'
Notice: Applied catalog in 0.01 seconds
ubuntu@ip-172-31-44-195:/opt/puppetlabs/puppet/bin$ |
```

Conclusion

Thus, we successfully learned about the syntax of manifest files and created our first puppet script to print a 'hello world' message.