**Puppet [ Configuration Management Tool ]**

**Puppet Architecture**

1. Puppet architecture follows a client-server model, where the Puppet Server acts as the master and nodes with Puppet Agents act as clients.
2. Puppet Server contains the configuration data, referred to as "manifests," which define the desired state of the nodes.
3. Nodes with Puppet Agent installed communicate with the Puppet Server to fetch configurations.
4. Puppet Server uses a Certificate Authority (CA) to manage SSL certificates for secure communication between the server and agents.
5. When a Puppet Agent is run for the first time, it generates a Certificate Signing Request (CSR) and sends it to the Puppet Server.
6. The Puppet Server signs the agent's CSR (has to be done manually by the administrator) and establishes a secure SSL connection for future communication.
7. Puppet Agent periodically requests for the catalog from the Puppet Server for updates, typically every 30 minutes by default.
8. Puppet Server compiles a catalog for the agent, which is a document describing the desired state of resources on the agent's system.
9. The catalog is sent back to the Puppet Agent, which applies it to enforce the desired configuration.
10. Puppet Agent reports back the status of the configuration application to the Puppet Server.

**Note**

* Puppet uses declarative language for defining configurations, ensuring idempotence by applying changes only when necessary.
* Resources, such as files, packages, and services, are managed through Puppet's Resource Abstraction Layer (RAL).
* Fact values, gathered by Facter, provide system information to Puppet for conditional configurations.
* Hiera is used as a key-value lookup tool to separate configuration data from code, enabling data-driven configurations.
* Puppet supports a wide range of modules for managing various applications, services, and operating systems.
* The architecture is scalable and can manage thousands of nodes using tools like PuppetDB for storing configuration data and reports.

**Steps to set up Puppet Server**

1. Install the Puppet Server package on the master node.

*apt install puppetserver -y*

1. Configure the Puppet Server memory allocation by editing the /etc/default/puppetserver or /etc/sysconfig/puppetserver file.

*JAVA\_ARGS="-Xms2g -Xmx2g" # Adjust memory settings as needed*

1. Set the Puppet Server hostname in /etc/puppet/puppet.conf under the [main] section.

*[main]*

*server = <hostname of puppet server>*

*certname = <hostname of puppet server>*

1. Start and enable the Puppet Server service on the master node.

*systemctl start puppetserver*

*systemctl enable puppetserver*

1. Allow Puppet's default port (8140) through the firewall.

*ufw allow 8140/tcp # For UFW*

*firewall-cmd --add-port=8140/tcp --permanent # For firewalld*

*firewall-cmd --reload*

1. Test the Puppet Server status to ensure it is running.

*systemctl status puppetserver*

**Steps to set up Puppet Agent**

1. Install the Puppet Agent package.

*apt install puppet-agent -y*

1. Configure the Puppet Agent by editing the /etc/puppet/puppet.conf file under the [main] section.

*[main]*

*server = <hostname of the puppet server>*

*certname = <certificate name of the puppet agent>*

1. Start and enable the Puppet Agent service on the agent node.

*systemctl start puppet*

*systemctl enable puppet*

1. Verify the Puppet Agent installation using the following command.

*puppet --version*

1. Manually run the Puppet Agent to generate a Certificate Signing Request (CSR) and send it to the Puppet Server ( or you can skip this as Puppet Agent when started, automatically generates CSR and sends it to the Puppet Server).

*puppet agent --test*

1. Now on the Puppet Server, list pending certificate requests

*puppetserver ca list*  –all

1. Sign the Puppet Agent certificate on the Puppet Server.

*puppetserver ca sign --certname <agent\_certname>*

*(or)*

*puppetserver ca sign –all (sign all certificates)*

1. Re-run the Puppet Agent to apply configurations from the Puppet Server

*puppet agent --test*

**Note**

* Add the fully qualified domain name (hostname) of the puppet server in /etc/hosts along with its IP address to avoid DNS resolving conflicts in the Puppet Agent node.

**Puppet Manifests**

A **Puppet manifest** is a file that contains the **declarative code** written in Puppet's domain-specific language (DSL). It describes the desired state of resources on a system, such as files, packages, services, and users. Manifests define **how resources should be configured**, and Puppet ensures that the actual state of the system matches this desired state.

**Key Points**

1. **File Extension**: Manifests are written in files with the .pp extension (e.g., site.pp, init.pp).
2. **Declarative Syntax**: Manifest code describes the desired state of resources rather than specifying step-by-step instructions.
3. **Resources**: Resources are the fundamental building blocks of manifests, such as files, services, and packages. Example resource types:
   * **file**: Manage files and directories.
   * **package**: Install or remove software packages.
   * **service**: Start, stop, or enable services.
4. **Modules**: Manifests are often organized into **modules**, which are reusable and shareable units of configuration.
5. **Main Manifest**: The main manifest (site.pp) is usually located in /etc/puppet/code/environments/production/manifests on the Puppet Server and is the entry point for configurations.

**Example**

# Ensure the Apache package is installed

package { ‘apache2’ :

ensure => ‘installed’ ,

}

**Puppet Classes**

In Puppet, **classes** are reusable, declarative blocks of code that define a set of configurations or resources. Classes allow you to organize and manage configurations for complex systems by grouping related resources together.

**Types of classes**

1. **Non-Parameterized Class**

class *<class\_name>* { # Declaring class

*<resource>* { ‘*<value>*’ :

*<argument>* => ‘*<value>*’ ,

}

}

node default {

include *<class\_name>* # Calling the defined class

}

**Example**

class ApacheInstallation {

package { ‘apache2’ :

ensure => ‘present’ ,

}

}

node default {

include ApacheInstallation

}

1. **Parameterized Class**

class *<class\_name>* (*parameters*) { # Defining class

*<resource>* { ‘*<value>*’ :

*<argument>* => ‘*<value>*’ ,

}

}

node default {

class { *<class\_name>* : # Calling the defined

*<parameters>* => ‘*<value>*’ ,

}

}

**Example**

class Apache( $packagename, $servicename) {

package { $packagename :

ensure => ‘installed’ ,

}

service { $servicename :

Ensure => ‘running’,

Enabled => ‘true’ ,

}

}

node default {

class { ‘Apache’ :

packagename => ‘httpd’ ,

servicename => ‘httpd’,

}

}

**Puppet Relationship and Ordering**

In Puppet, relationships and ordering are key concepts used to define the sequence in which resources are applied to ensure configurations are done in the correct order. Puppet is declarative by default, meaning resources can be applied in any order unless you explicitly specify their relationships. There are two ways to do so, they are as follows

1. **Relationship Metaparameters**

Puppet provides special metaparameters to define relationships between resources. These ensure that resources are applied in the desired order.

* **Before**

Ensures a resource is applied before another resource.

file { '/tmp/config':

ensure => present,

**before => Package['nginx'],**

}

package { 'nginx':

ensure => installed,

}

* **Require**

Ensures a resource is applied after another resource is successfully applied.

package { 'nginx':

ensure => installed,

}

service { 'nginx':

ensure => running,

**require => Package['nginx'],**

}

* **Notify**

Triggers an action on another resource whenever the notifying resource is changed.

file { '/etc/nginx/nginx.conf':

ensure => file,

content => template('nginx/nginx.conf.erb'),

**notify => Service['nginx'],**

}

service { 'nginx':

ensure => running,

subscribe => File['/etc/nginx/nginx.conf'],

}

* **Subscribe**

The inverse of notify. It ensures a resource reacts to changes in another resource.

service { 'nginx':

ensure => running,

**subscribe => File['/etc/nginx/nginx.conf'],**

}

1. **Arrow Syntax for Ordering**

You can also use arrow syntax (-> or ~>) for defining resource relationships in a simpler, more readable way.

* **->**

Represents a strong ordering relationship between resources. Ensures the first resource is applied before the second resource. There is no notification if the first resource changes.

file { '/tmp/config':

ensure => present,

} **->**

package { 'nginx':

ensure => installed,

} **->**

service { 'nginx':

ensure => running,

}

* **~>**

It represents a notification or weak ordering relationship. Ensures the first resource is applied before the second resource. If the first resource changes, it notifies the second resource, which may trigger an action (e.g., restart).

file { '/etc/nginx/nginx.conf':

ensure => file,

content => template('nginx/nginx.conf.erb'),

} **~>**

service { 'nginx':

ensure => running,

}

**Puppet Module**

A Puppet module is a collection of reusable code that simplifies the management of resources and configurations in Puppet. It is a structured and standardized way to organize and distribute Puppet code, making it easy to manage complex configurations.

In simple words, The Puppet module is a folder that contains everything you need to set up or manage something on your system, like installing a web server (e.g., Nginx) or creating users.

This folder includes:

* The instructions for what to do (code).
* Optional files or templates to copy to the system (like configuration files).
* Metadata (just a small description of the module).

The structure of the module follows below diagram

mymodule/

├── manifests/ # Contains the main Puppet code

│ └── init.pp # Entry point for the module

├── files/ # Contains static files to be deployed to nodes

├── templates/ # Contains dynamic templates (e.g., .erb or .epp files)

├── lib/ # Contains custom facts and functions (optional)

├── examples/ # Contains usage examples for the module (optional)

├── spec/ # Contains tests for the module (optional)

├── metadata.json # Contains metadata about the module (author, version, dependencies)

├── README.md # Documentation for the module

**Example**

nginx/

├── manifests/

│ └── init.pp

├── files/

│ └── nginx.conf

├── templates/

│ └── virtualhost.erb

└── metadata.json

**Steps to create and use Puppet Module**

1. Create a directory with the name of your module inside /etc/puppet/code/environment/production/modules
2. Inside the module directory, create a manifests folder.
3. In the manifests folder, create an init.pp file.
4. Write the main class for the module in the init.pp file.
5. Optionally, create a files folder to store static files that the module can deploy.
6. Optionally, create a templates folder to store dynamic templates for configuration files.
7. Optionally, create a metadata.json file to define module metadata like name, version, and dependencies.
8. Use the include <module\_name> statement to call the module in your manifest.
9. If you are using any module from Puppet Forge (repo for predefined modules) you have to download it in your puppet server and include it in your manifest file.

*puppet module install <module\_name>*

**Puppet Facts**

Puppet facts are basically pieces of information about your system that Puppet gathers automatically when it runs. These facts are used to make decisions about how to configure your system based on its specific properties (e.g., what operating system it is using, how much memory it has, etc.).

*facter <facts\_to\_view> (or) facter os (or) facter os.family*

**Default Facts**

Puppet automatically collects a set of default facts when the agent runs. These include system information such as:

* Operating system (e.g., os, osfamily, operatingsystem)
* CPU architecture (e.g., architecture)
* Memory (e.g., memorytotal, memoryfree)
* Hostname (e.g., hostname)
* IP address (e.g., ipaddress)
* Domain name (e.g., domain)

**Custom Facts**

These are user-defined facts that are written using ruby language. To create a custom fact follow the below given steps

* Create a directory - **/etc/puppet/code/environments/ production/modules/<module\_name>/lib/facter** for custom facts inside your module
* Create a Ruby file - **/etc/puppet/code/environments/ production/modules/<module\_name>/lib/facter/my\_custom\_fact.rb** for the custom fact in the facter directory.
* Open the Ruby file (my\_custom\_fact.rb) and define the custom fact

Facter.add('my\_custom\_fact') do

setcode do

'custom\_value'

end

end

* Save the file, the Puppet will now recognize your custom fact.

**PuppetDB**