Using Puppet Locally:

Puppet is usually used in a client-server structure, but we don’t have to follow that kind of structure, in order to use it, we can also use puppet rules locally in our computer as a stand-alone application running it from the command line.

Installing Puppet:

We can install puppet from the command line available in the OS, or we can download it from the official website.

The puppet files are called manifests, and the end with the extension ‘. pp’. Inside of these files, is where we write the rules that we want out system to follow to get to the desired state:

package { 'htop': ensure => present, }

Here we are applying a rule that ensures that the package ‘htop’ is installed. Next step is going to the prompt and write the next line of code, that will apply this rule to the computer:

user@ubuntu:~$ sudo puppet apply -v tools.pp

In this example, we are using -v, to tell the process that we want verbose output, that means that we will have a constant information flow of what it’s going on inside the installation.

Info: Loading Facts Notice: Compiled catalog for <user> in environment production in 0.02 seconds Info: Applying configuration version ‘15756519819’ Notice: /Stage[main]/Main/Package[htop] /ensure: created Notice: Applied catalog in 2.32 seconds user@ubuntu:~$

What is a **catalog**?

A **catalog** is the list of rules that are generated for one specific computer once the server has evaluated all variables, conditionals, and functions.

Managing Resource Relationship:

Usually, the puppet manifest that is used in our computers for applying the rules has many resources that are related to each other. You can not configure a Package that is not installed, is not logic, neither you can start a Service which needs both a Package and a configuration File without these two resources being present. We can have control of the relationship that exists between these resources using the **Resource Relationship** as follows in the next example:

**class ntp { package { ‘ntp’: ensure => latest, } file { ‘/ect/ntp.conf’: source => “puppet:///modules/ntp/ntp.config”, replace => true, require => Package[‘ntp’], notify => Service[‘ntp’], } service {‘ntp’: enable => true, ensure => running, require => File[‘/ect/ntp.conf’], } } include ntp**

This way puppet knows that before starting the service, both the configuration file and the package must be present, also establishes a relationship between them, so in the future if both the package or the configuration file are modified, the service will be notified and will change depending on what changes were made.

Here we can see that there is a slight difference, when we are declaring a resource, this is written in lowercase, and the when the resource is called for establish a relationship, it is Capitalized. This does not have a specific reason to be, is just the way that the puppet language is written.

Finally, we can see that there is a line at the bottom of the file that says **include ntp**, as you can imagine we have been just declaring a function, but not calling it, this line of the code is the one that applies the rules described in the ntp class (is the same as creating a function, it won’t work if you don’t call it)

Puppet Modules:

A module is a collection of manifests and associated data. We create them because sometimes, the configuration required for a type of device is different to the configuration required for another one, so it’s manifest will be different, but we still can store them all together in a puppet module that contains them all.

Also, all manifests are stored in a directory called Manifests, the files, like the ntp.config file we saw earlier is stored in a directory called ‘Files’, also, there is a ‘Template’ directory that contains files that are pre-processed before getting to the client machine

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

There are more directories that can be part of a module depending on what the module does, but these are the fundamental ones. The same way as in a python module, we must create an init file, in this case an init.pp file, and inside, it should define a class with the name of the module we are creating, inside of the ‘Manifest’ directory. Then any files that your rules are going to use, need to be stored inside of the ‘Files’ directory or the ‘Templates’ directory depending on how you need to pre-process your files or not.

Since it has been some time now since the puppet language was developed, there are a lot of pre-determined modules that are already created and that have the same rules that we might need for the deploying of our system, and we can just copy them, and use them.

We can install modules as it follows:

user@ubuntu:~$ sudo apt install puppet-module-puppetlabs-apache

This is How a module directory looks like:Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

And this is what there is inside if the Manifest directory:

Puppet Nodes:

We can get puppet to apply general rules to all our nodes, and the make it aplly specufuc rules to some nodes, because some devices have diferent functioins from one another, like a web server from a e mail server for example. The rules that need to be aplly to one, is not the same for the other.

When setting up puppet we usually have a default node definition that lists classes that should be included for all the nodes.

node default { class { ‘sudo’: } class{ ‘ntp’: servers => [‘ntp.example.com’, ‘ntp.example.com’] } }

So here as we can see we have a default node, this means that for all the computers in the fleet, these rules will be applied, and this node as we can see, has two classes in it, so the rules applied will be those inside of those classes.

Up next, we can stablish what specific rules we want to apply for specifics devices:

node webserver.example.com { class { ‘sudo’: } class{ ‘ntp’: servers => [‘ntp.example.com’, ‘ntp.example.com’] }

Class { ‘apache’: } }

This information about which rules are applied to which servers, are stored in a file called site.pp which does not belongs to a module, it just defines what classes belong to what nodes.

Puppet’s Certificate Infrastructure:

It’s a way of certificate that the machine’s identity is the same that we are receiving, it’s true and not someone else, we can do that by using CA or Certificates of Authenticity.

Setting up Puppet Clients as Servers:

This is never to be done in an actual system, we do this to skip the authentication part, is dangerous to do it in a real system.

user@ubuntu:~$ sudo puppet config –section master set autosign true

Up next, we are going to connect to each machine, and we’ll install either the agent, or the master as it follows:

user@ubuntu:~$ ssh webserver user@webserver’s password: user@ubuntu:~$ sudo apt install puppet [sudo] password for user:

Now we must set to which server is going to be linked this agent, and then in the next line we test the connection to make sure that everything went well

user@ubuntu:~$ sudo puppet config set server ubuntu.example.com user@ubuntu:~$ sudo puppet agent -v –test

Modifying and Testing Manifests:

As we said already when we change the manifest modifying a setting that’s already managed by puppet, Puppet applies this change to the notes, the puppet agent does what ever is needed to bring the nodes to the desired state. This way you can make a small change in your manifests and have that modify in all the devices.

This can be very helpful, but at the same time very dangerous, so it’s common to first test the change to be made, so we know for sure what we are doing, and then proceed to modify the manifest.

Testing the manifests is easy in puppet since it comes with a tool for that baked in. First you want to use a parser tool called **puppet parser validate** which tells us if we have any typo error, syntax error or that sort of things. Then we can run the rules using the **--noop** parameter, which comes from no operation, and means exactly that; meaning that puppet will just run a simulation of what would happen if we applied the rules written in the manifest.

Another option we may also think of, if the change is going to be very big or very complex is to have a test machine only for testing. But these are manual ways of testing, instead, we can use a tool to automatically test our rules, the name of the tool we use is named **rspec tests.**

**Example:**

**describe ‘gksu’ , :type => :class do let (:facts) { { ‘is\_virtual’ => ‘false’ } } it { should contain\_package(‘gksu’).with\_ensure(‘latest’)} end**

Here we are setting the ‘is\_virtual’ fact to false, and then we make sure that the structure contains the ‘gksu’ package and we ensure that the version is the latest. In these lines we are making sure that the rules are written correctly, but we are not testing id=f they do what they are supposed to.

Rolling Changes and Validating Them:

Once you've prepared and tested the changes that you want to make, it's time to roll them out, but not so fast. Even if you've tested the change on your computer or on a test computer and it worked just fine, it doesn't mean that the change will work correctly on all machines running in production.

Production:

Production are the parts of the infrastructure where a service is executed and served to its users.

Once we know the changes are all corrects, we are ready to push the changes to the nodes. When we are doing that, we are going to do it by chunks and not by all the machines on the fleet at the same time. We do this so if there is a problem only a very little part of the users may see it, and not all of them, creating a bigger problem. Once we have waited one day at least, and our users haven’t reported any issues, we are ready to deploy the change in all the machines