What is configuration management?

Unmanaged configuration refers to manually install the configuration of a device individually.

Managed configuration means to use a configuration management system to handle all the configuration of all the devices in your fleet, also known as nodes (the devices).

Typically, you define a set of rules that must apply to all nodes, and then start a process that verifies that all settings all true in each one of the nodes, meaning, that the config file in the configuration management system is equal to the config file in each device(node).

The more servers you must deploy, the most time it will take for you to configurate each device using the unmanaged configuration, while using the managed configuration, you can deploy the configuration in all devices, at once.

Whenever you need to make a modification in the rules of the fleet you only need to change the rules inside of the configuration management file, in the configuration management system, and then, the new rules will apply for all the devices in the fleet.

The CMS often have an automatic error correction built in, so that they can recover from certain types of error by themselves. This kind of system is used to monitor all the changes in all the configurations in all the apps in all devices, so if for example, a user does a change in a configuration in a determined app, which is not the one that have for default in you CMS, the CMS is going to go back to your previously set configuration, using a Version Control System works.

In this course we are going to use a configuration management system called **Puppet**.

What is infrastructure as code?

The paradigm of storing all the configuration for the managed devices in version-controlled files known as Infrastructure as Code or IaC. We say we are using an infrastructure as code or IaC, when all the configuration necessary to deploy and manage a node in the infrastructure is stored in a version control system, as Git, for example.

Is very useful to have a managed configuration system, even If we have only one device in our fleet, this is because if by some reason our device stops working, or the OS, or any important application in the device brakes, we can use the CMS for installing the new device at the same place in time as we had it before the fatal event happened.

What is Puppet?

Puppet is an open-source project, a cross-platform tool created in 2005. We typically deploy Puppet using a client-server architecture, the client is known as the puppet agent, and the server is known as the puppet master.

The puppet agent connects to the master and send a lot of facts that describe the computer to the master, then the master processes this information and generates the list of rules to be applied on this device and send it to the agent. Once the agent receives the list of rules, is the agent the one in charge of making any changes in the computer.

**Puppet Agent**-------facts----→**Puppet Master**----↓ ↑←-----------------rules←------------------------------←

Since puppet is cross-platform we can use the same rules for a very broad range of computers, but, what are the rules?

This a small block as example:

**class sudo {**

**package {‘sudo’: ensure => present,**

**}**

**}**

This one for example means that is going to check is the package sudo is installed in the computer where the rule is applied. This means that if, for instance, we have 100 computers, and in any of those computers, the package sudo is not installed, by rule, the package will be installed.

There are various installation tools available for each OS, Puppet will automatically detect which is the OS we are using, and then select the corresponding tool for it. For ***Linux***, the package management system’s names are **APT** **Yum** or **DNF**, on ***MacOS*** it depends of where is the package coming from, for example, the **Apple** **Provider** is used for packages that are part of the OS, while the MacPorts provider is used for the **MacPorts** projects, for ***Windows*** is a bit different because we are going to need to add an extra attribute to our rule, which is the location of the installer file, if its in the local deck or a network mounted resource.

Puppet Resources:

In puppet, resources are the basic unit for modeling the configuration that we want to manage, in other words, each resource specifies one configuration that we are trying to manage, like a service, a package, or a file.

In the next example, in line 3, we are defining a file resource, and in line 4, we are ensuring that the resource is a directory.

EX:

**1 class sysctl {**

**2 *#Make sure the directory exists, come distros don’t have it***

**3 file {‘/etc/sysctl.d’: 4 ensure => directory,**

**}**

**}**

Example 2:

**1 class timezone {**

**2 *#It makes sure the file exists, sets the content of the file to the UTC timezone, and set replaces the file even if it exists already***

**3 file {‘/etc/timezone’: 4 ensure => file, content => “UTC\n”, replace => true,**

**}**

**}**

There are a lot more of attributes we can use, for example, the file permissions, the file owner, the file modification time.

This is the link to the official documentation of the Puppet attributes:

<https://puppet.com/docs/puppet/latest/lang_resources.html>

<https://puppet.com/blog/deploy-packages-across-your-windows-estate-with-bolt-and-chocolatey>

But how exactly do we turn these rules we are writing into actual changes in a computer?

When we declare a resource inside of a rule, we are defining the desire state of that resource inside of the system, once the puppet agent has received the list of rules, is the Provider the one in charge of making the changes happen.

When the puppet agent processes a resource, it first decides which provider it needs to use, then passes along the attributes that we configured in the resource to that provider. The code of each provider oversees (is in charge of) making our computer reflect the state requested in the resource.

Puppet Classes:

In the examples we have seen this far, we have declared classes that contain one resource, we use these classes to collect the resources that are needed to achieve a goal in a single place.

For example, you could have a class that installs a package, sets the contents of a configuration file, and starts the service provided by that package, as appears in the next example:

**1 class timezone { package {‘ntp’: ensure => latest, } file {‘/ect/ntp.config’: source => “puppet:///modules/ntp/ntp.config” replace => true, } service {‘ntp’: enable => true, ensure => running, } }**

Here we can see that in this class we have three resources, a package, a file, and a service. All these resources are related to something called ntp, which stands for Network Time Protocol which is the mechanism our computers use to synchronize our clocks.

In this case our rules are making sure that the ntp package is upgraded to the latest version. We are setting the contents of the configuration file using the source attribute, which means that the agent will read the required contents from the specified location. Also, we are saying that we want the NTP service to be enabled and running.

By grouping all the resources related to NTP in the same class, we only need a quick glance to understand how the service is configured and how is supposed to work. Also, it will be very easy for someone else or us to make changes in the future if all the settings are related to one service are included in one class.

What is a domain-specific language?

Until now we have seen how to use a resource in order go generate a desired state inside of a system, but there is much more that we can do with the power of Puppet, and that is because puppet has its own DSL, or Domain Specific language, but what is a DSL?

Typical programing languages like Python, Java or Go, are GPL, or General Purposes programing Languages, which means they can be used to write a lot of different applications, with different goals, and use cases. On the other hand, a Doman Specific Language is a programing language that is more limited, usually learning a DSL is much easier than learning a GPL because there is a lot less to cover, like the syntaxis or keywords for example.

In the case of Puppet, the DSL is limited to when and how we apply a configuration management rule to a device. On top of the resource type that we have already seen, Puppet’s DSL includes **variables**, **conditional statements,** and **functions.**

Another important characteristic of puppet are **Facts**.

**Facts:** Variables that represent the characteristics of the system. When the puppet agent runs, it calls a program called factor which analyzes the current system, storing the information that gathers, in these facts. Once this is done, it sends these values for these facts to the server, the puppet master, which uses them to calculate the rules that should be applied.

This means that the puppet master receives a file named fact, with a list of all the files, including configuration files of course, that should be on the system, if these files are not present, or they are modified or outdated, the puppet master will send the rules, which are all the code lines that saw above that are passed to the Provider, and this is ultimately who realizes the changes in the system’s files.

What information can the **facts** have?

The **fact** files can have different types of information on them, for example, what the **current OS** is, how much **memory** the computer has, whether is a virtual machine or not, or what the current **IP address** is. But what happen when the information that we need is not extractable by using of the predefined facts, well the solution is so simple as writing a script that gathers the information that we need, and then create a new kind of fact ourselves. But how does a fact look like?

**If** **$facts**[‘is\_virtual’] { package {‘smartmoontools’: ensure=> purged, } } **else** { package {‘smartmooltools’: ensure=> installed, }

}

In this example we are using the baked in fact called is\_virtual. This fact returns True or False, depending on if the machine virtual or not, and what we are doing here is checking that because the smartmoontools package is used to monitor hard drives, and since virtual machines don’t have a hard drive, it doesn’t make sense to have it installed, so if it’s a virtual machine it ‘purges’ the package, and if is not, it ‘ensures’ it.

Here we can see the word **$facts** which is a variable inside of the DSL of puppet. The **$fact** variable is also known as hash, remember that in other languages, a hash is the same we call dictionary in python which means that as in a dictionary, we can access to its elements using their keys, the same way we access them in a python dictionary. In this case we can see that we have written $facts[‘is\_virtual’], which means that we are accessing the whatever it is inside of the value inside of the $facts ‘dictionary’, in its ‘is\_virtual’ value.

Also, we can see the conditional statements **‘if’** and **‘else’**. This kind of bocks, no matter the language, must have a defined indentation of any sort, in this case it’s a little confusing because we have to kind of curly braces, we have the ones that are used to indent each conditional block from one another, and we have the ones that we use inside of puppet for defining its resources.

What is the syntax of Puppet? >>Because its already a bit confusing to continue without knowing **(-\_-)**

Every resource must start with the type of resource defined, in this case we are defining a **package** **resource**, right after we declare the kind of **resource**, we have to also declare what is the content of this **resource** and we do that by opening a curly brace, then a **title**, followed by a colon, an then, we type the attributes that are being set. We use => to assign values to the attributes and then each attribute ends with a come.

**If** **$facts**[‘is\_virtual’] { **package** {**‘smartmoontools’**: ensure=> purged, } } **else** { **package** {‘smartmooltools’: ensure=> installed, }

}

Principles of configuration Management:

We have seen that puppet is a language that is very easy to use, since there is only that to the language, and that is because of its first principle, Puppet is a **Declarative language**, this means that we only write the state that we want to achieve instead of writing the actual procedure that the computer has to make to get to the goal, as in python that is what we call a **Procedure Language**. Here we have already got to the point where wea re using what others have already create, is we want a package installed, we don’t care what procedures is the computer following to install it, we only care about the fact that the package is installed.

Other aspect that we care here is that operations are idempotent. In this context an idempotent action can be performed repeatedly without changing the system after the first time the action was performed, and with no unintended side effects.

This means, for example what we want a determined file, in a determined directory, and we are using puppet for it, the result of running the configuration management system will be the same, no matter is the file exists, doesn’t exist, or if exists and is different, will be changed, the result will be the same no matter what, this is what an idempotent action me.

Another important aspect of how configuration management works is the test and repair paradigm. This means that actions will only be taken when needed to achieve a goal, or what is the same, puppet will first test if the resource that is being managed, needs to be modified.