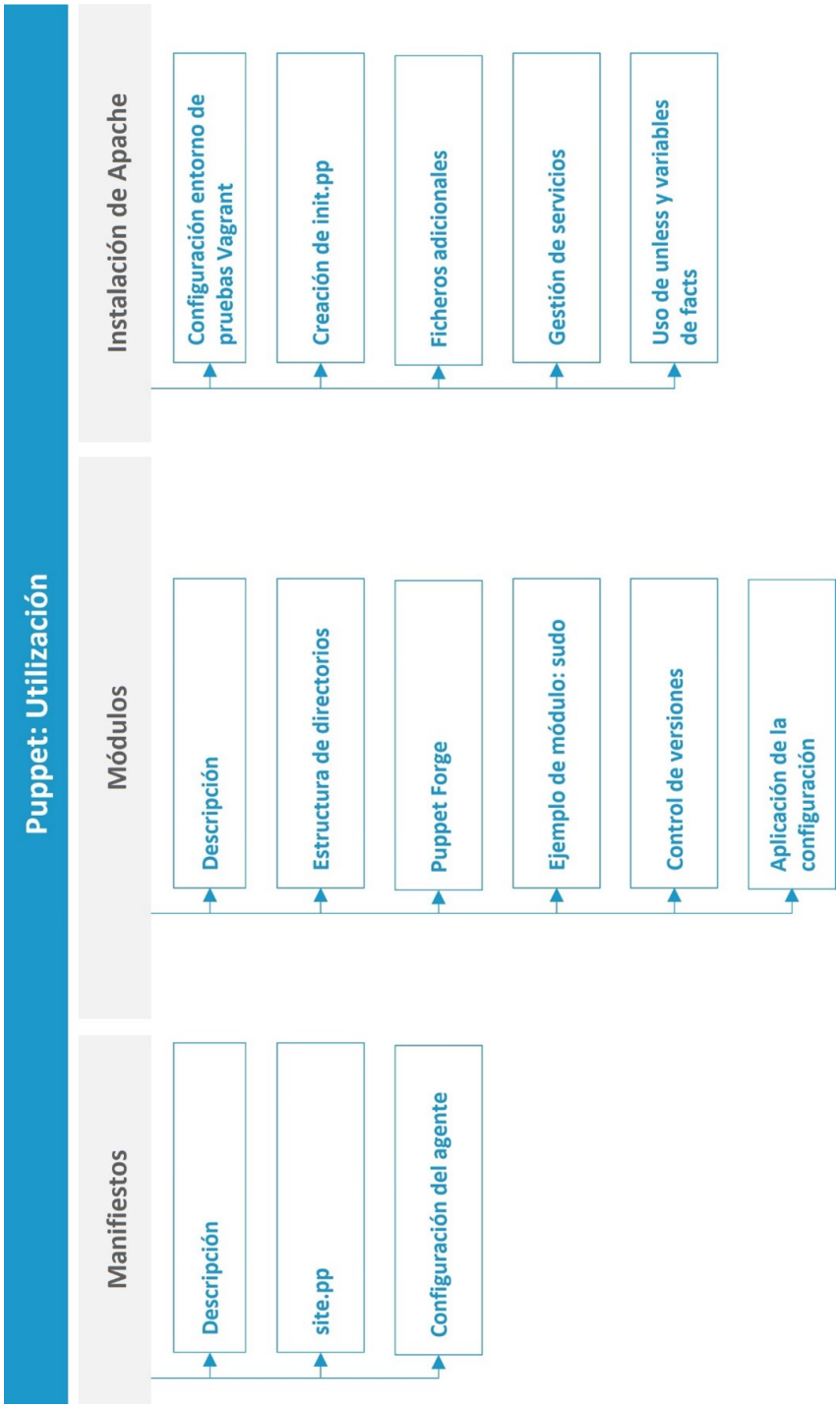
Herramientas de Automatización de Despliegues

Tema 2. Puppet. Utilización

Índice

Esquema



2.1. Introducción y objetivos

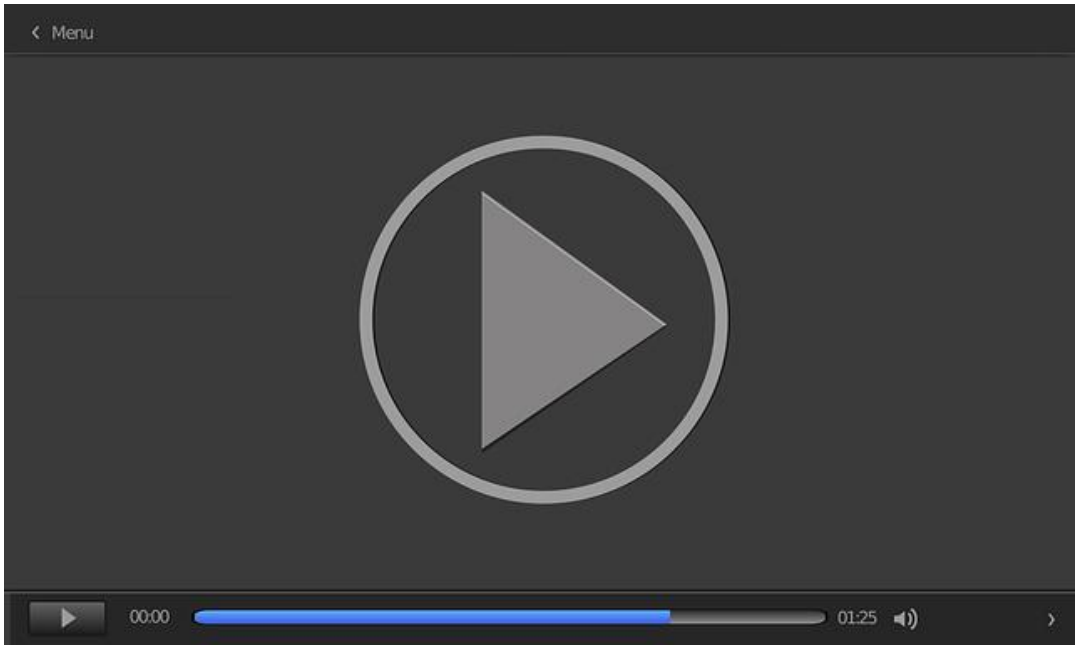
Una vez hecha la introducción a Puppet en el tema anterior y explicados los diversos modos de instalación y configuración inicial, vamos a revisar más a fondo su lenguaje, las capacidades de los componentes de Puppet y cómo definir los elementos de configuración.

Los objetivos que se pretenden conseguir en este tema son:

- Estudiar los archivos *manifest* y sus características.
- Configurar el agente.
- Trabajar con módulos mediante un ejemplo práctico.
- Estudiar el archivo `init.pp`
- Hacer una primera configuración de Puppet.
- Instalar Apache.

Ideas clave

A continuación, puedes ver el vídeo *Mejores prácticas en Puppet*:



Accede al vídeo:https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=679981f5-8327-4256-b24d-abb700b54b3d

2.2. Archivos de manifiesto (manifest)

Hemos visto anteriormente que Puppet describe los archivos que contienen información de configuración en la forma de *manifests*.

Estos se componen de una serie de elementos principales:

- **Recursos:** elementos de configuración individuales.
- **Archivos:** archivos físicos que pueden servir a los agentes.
- **Plantillas:** plantillas que se pueden utilizar para rellenar archivos.
- **Nodos:** especifican la configuración de cada agente.
- **Clases:** colecciones de recursos.
- **Definiciones:** colecciones compuestas de recursos.

Todos estos componentes se definen mediante un lenguaje de configuración que también proporciona la posibilidad de incluir variables, condicionales, matrices y otras funcionalidades.

Puppet también incorpora, además de los componentes ya mencionados, el concepto de módulo (*module*), que está compuesto por un conjunto portable y reutilizable de manifiestos que incorporan recursos, clases, definiciones, archivos y plantillas. Los módulos serán explicados más adelante.

Extensión del archivo `site.pp`

Lo primero que tenemos que hacer para crear el primer agente y definir la configuración es la extensión de archivo `site.pp`

```
Import 'nodes.pp'
```

```
$servidorpuppet = 'puppet.ejemplo.edu'
```

El comando import indica a Puppet que debe cargar un archivo llamado `nodes.pp`, ya que este comando nos permite importar otro manifiesto de Puppet dentro de nuestro archivo.

El siguiente ejemplo a continuación incluye un archivo denominado `resources.pp`, donde tendríamos definidos nuestros recursos:

```
import 'resources.pp'
```

Cuando Puppet se inicie, cargará y procesará primero el fichero `nodes.pp`, donde se encuentran definidas las configuraciones de nodo que serán aplicadas a cada agente que se conecte. También podríamos importar varios archivos usando comodines:

```
import 'nodes/*'
```

```
import 'classes/*'
```

En este ejemplo, el comando import se encargará de incluir todos los ficheros con extensión `.pp` que se encuentren en los directorios nodes y classes.

Por otro lado, `$servidorpuppet` declara una variable. Puedes definir una variable en Puppet con el signo dólar, seguido del identificador de la variable y su valor tras el signo igual (=), estas variables se pueden posteriormente referenciar desde la configuración de Puppet.

Cabe señalar que en los manifiestos de Puppet, tal como ocurre en muchos sistemas operativos, las cadenas de caracteres entre comillas dobles permiten la sustitución de las variables, mientras que las cadenas entre comillas simples, no. Si deseas utilizar el valor de una variable dentro de la cadena, debes declarar la cadena entre comillas, como por ejemplo: “Esto es una cadena `$variable`”. También se puede encerrar el nombre de la variable entre llaves, ${variable}, para definirla más claramente.

Si quieres conocer más sobre esta característica, puedes consultar la información en el siguiente enlace:
https://puppet.com/docs/puppet/latest/lang_data_string.html

Configuración del agente

A continuación, añadiremos la primera definición de un agente al archivo “`nodes.pp`” que hemos importado. En Puppet los agentes se definen en los manifiestos utilizando nodos. Primero creamos el archivo:

```
touch /etc/puppet/manifests/nodes.pp.
```

Luego incluimos la definición del nodo:

```
node 'nodo1.ejemplo.edu' {
 include sudo
}
```

Para la definición del nodo especificamos el nombre del nodo entre comillas simples y, entre llaves {}, añadimos la configuración que se le debe aplicar. Se puede utilizar el nombre de host o el nombre completo de dominio de la máquina como nombre del nodo o cliente, aunque también podemos utilizar una lista de nombres separados por comas.

En este punto, ya no se pueden utilizar expresiones de nodos con comodines, como por ejemplo `*.ejemplo.edu`, pero sin embargo se pueden utilizar expresiones regulares, tales como:

```
node /^www\d+\.ejemplo\.edu/ {
 include sudo
}
```

Este ejemplo de expresión regular coincidirá con todos los nodos del dominio "example.com" con el nombre de *host* `www1, www12, www123`, etc.

Para más información sobre el uso de expresiones regulares en Puppet, consulta: https://puppet.com/docs/puppet/latest/lang_data_regexp.html

A continuación, especificaremos una directiva include en nuestra definición de nodo. La directiva include nos sirve para especificar un conjunto de configuraciones que deseamos aplicar a la máquina, pudiéndose incluir dos tipos distintos de colecciones:

- **Clases:** colección básica de recursos.
- **Módulos:** colección avanzada y reutilizable de recursos que pueden incluir clases, definiciones y otras configuraciones de soporte.

Se pueden también incluir múltiples colecciones mediante el uso de varias directivas `include` o en una única directiva proporcionando una lista de colecciones separadas por comas:

```
include sudo
```

```
include apache
```

```
include vim, syslog-ng
```

También se pueden definir recursos individuales dentro de la propia definición de nodo, aparte de las colecciones:

```
node 'nodo1.ejemplo.edu' {
```

```
include sudo
```

```
package {'vim': ensure => present}
```

```
}
```

No obstante, es recomendable no definir recursos directamente dentro de las declaraciones de nodos, por lo que vamos a seguir esta recomendación y dejar únicamente la colección de recursos, el módulo sudo.

Para más información sobre nodos en Puppet, puedes consultar esta sección en la documentación oficial de referencia de la herramienta:
https://puppet.com/docs/puppet/latest/lang_node_definitions.html

2.3. Módulos

Un módulo en Puppet se refiere a una colección reutilizable de manifiestos, recursos, archivos, plantillas, clases y definiciones. Un módulo puede incorporar todo lo que es requerido para configurar por completo una aplicación, esto es, podría incluir toda la definición de recursos en ficheros *manifests*, archivos necesarios y configuraciones asociadas que se requieren para configurar Apache, por ejemplo, en una máquina.

Un módulo consta de una estructura de directorios concreta y un archivo principal llamado `init.pp`, que será el punto de entrada. Esta estructura permite que Puppet pueda cargar módulos de forma automática, para lo cual Puppet revisará los directorios que se hayan especificado mediante las rutas de módulos. Estas rutas de módulos se configuran en el fichero `puppet.conf` mediante la opción de configuración modulepath dentro de la sección `[main]`. Por defecto, Puppet comprobará los directorios `/etc/puppet/modules` y `/var/lib/puppet/modules` en busca de módulos, pero se pueden personalizar las rutas mediante la opción de configuración:

```
[main]
```

```
moduledir =
/etc/puppet/modules:/var/lib/puppet/modules:/opt/modules
```

Esta carga automática de módulos nos permite no tener que incluirlos explícitamente mediante la directiva import, a diferencia de lo que pasa con el fichero `nodes.pp`.

Estructura del módulo

En el siguiente fragmento crearemos un directorio de módulos junto con su estructura de archivos correspondiente. Para ello, tomaremos como base el directorio `/etc/puppet/modules`, pero lo puedes sustituir por cualquier otro. El nombre del módulo será sudo. Tanto los módulos como las clases deben tener un nombre que

solo incluya letras, números, subrayados y guiones.

```
# mkdir -p /etc/puppet/modules/sudo/{files,templates,manifests}
```

```
# touch /etc/puppet/modules/sudo/manifests/init.pp
```

El directorio de *manifests* será donde se encuentre el fichero `init.pp` y podría también contener cualquier otro manifiesto de configuración que fuera necesario. El archivo `init.pp` es el núcleo del módulo y define la clase principal, a la que nos referiremos al utilizar el nombre del módulo directamente. El directorio files almacenará cualquier archivo que necesitemos utilizar como parte de nuestro módulo, mientras que el directorio templates contendrá cualquier plantilla que requiera nuestro módulo.

Estos son solo algunos de los directorios que un módulo puede tener. A continuación, se enumeran todos los directorios posibles que un módulo puede incorporar, junto con una breve descripción de su cometido.

Estructura de directorios de un módulo Puppet	
Directorio	**Contenido**
`data`	Ficheros de datos que especifican valores por defecto de los parámetros
`examples`	Ejemplos que muestran como declarar las clases del módulo y los tipos definidos
`facts.d`	Hechos (facts) externos, que son una alternativa a los *custom facts* basados en Ruby. Se sincronizan a todos los agentes de nodo, por lo que pueden recopilar valores para esos hechos y mandarlos al Puppet Master
`files`	Archivos estáticos que los nodos gestionados pueden descargar
`functions`	Funciones personalizadas escritas en lenguaje Puppet
`lib`	*Plug-ins*, tales como *facts* (en el subdirectorio `facter/`) y tipos de recursos, funciones y proveedores (`puppet/`) personalizados
`locales`	Archivos relativos a la localización del módulo, para otros lenguajes que no sean el inglés
`manifests`	Los manifiestos del módulo: `init.pp` y cualquier otra clase. También se permite agrupar clases en subdirectorios
`plans`	Planes de tareas Puppet, que son conjuntos de tareas que pueden ser combinadas con otros elementos lógicos
`readmes`	Los archivos README del módulo localizados en otros lenguajes que no sean el inglés
`spec`	Pruebas de *spec* para cualquier plugin en el directorio lib
`tasks`	Tareas Puppet, que pueden estar escritas en cualquier lenguaje que entienda el nodo destino
`templates`	Plantillas que los manifiestos del módulo pueden usar para generar contenido o valores de variables
`types`	Alias de tipos de recursos

Tabla 1. Estructura de directorios de un módulo Puppet. Fuente: elaboración propia.

Puppet Forge

Como ya hemos indicado anteriormente, los módulos son colecciones portables y reutilizables, que permiten encapsular las configuraciones y compartirlas.

Puppet Forge es el repositorio que proporciona Puppet para tal fin, en el que podremos encontrar miles de módulos compartidos por desarrolladores de Puppet y por la propia comunidad. Todos estos módulos están disponibles para descargar e

instalar en nuestro entorno Puppet y aprovechar sus funcionalidades.

También puedes compartir en Puppet Forge tus propios módulos, sobre los que podrás recibir contribuciones, y podrás mantener y liberar versiones sucesivas.

Para acceder a Puppet Forge, visita desde el navegador:

https://forge.puppet.com

2.4. Ejemplo de módulo: sudo

En el siguiente paso vamos a crear el módulo sudo. Para ello, vamos a comenzar por analizar el archivo `init.pp`:

```
class sudo {

 package {sudo:

 ensure => present,

 }

 if $operatingsystem == "Ubuntu" {

 package {"sudo-ldap":

 ensure => present,

 require => Package["sudo"],

 }

 }

 file {"/etc/sudoers":

 owner =>"root",

 group =>"root",

 mode => 0440,

 source =>"puppet://$servidorpuppet/modules/sudo/etc/sudoers",
```

```
 require => Package["sudo"],
```

```
 }
```

```
}
```

Este fichero `init.pp` de nuestro módulo consta de una única clase, también denominada sudo, que contiene dos recursos de paquetes y uno de fichero.

El primero de los recursos que definimos del tipo `package` sirve para asegurar que el paquete sudo está instalado, mediante el atributo `ensure => present`, mientras que el segundo de los recursos utiliza la sintaxis if/else que nos proporciona Puppet para condicionar la instalación del paquete `sudo-ldap` al tipo de sistema operativo. Puppet también proporciona, aparte de la sentencia if/else, otros dos tipos de sentencias condicionales: case y selector.

Puedes ver más detalles de las sintaxis condicionales de Puppet en:
https://puppet.com/docs/puppet/latest/lang_conditional.html

Puppet verificará el valor del *fact* del sistema operativo para cada cliente que se conecte y, si el *fact* operatingsystem tiene el valor Ubuntu, entonces Puppet se encargará de instalar también el paquete `sudo-ldap`. Ya habíamos hablado de Facter y sus valores en el tema anterior. Cada *fact* está accesible mediante una variable, que tiene el mismo nombre que el *fact* añadiéndole el prefijo de un signo dólar ($), que se puede utilizar en los manifests de Puppet.

Continuando con el análisis del archivo `init.pp` del módulo `sudo`, vemos que para este mismo recurso también hemos añadido un nuevo atributo, require. El atributo require es un metaparámetro, que son los incluidos en el *framework* de Puppet, en vez de ser propios de un tipo concreto, y sirven para realizar acciones sobre los recursos, pudiendo ser especificados para cualquiera de sus tipos.

El metaparámetro require, en nuestro caso, sirve para definir una relación de dependencia entre el recurso `Package[“sudo-ldap”]` y el recurso `Package[“sudo”]`, por lo que le decimos a Puppet que el recurso `Package[“sudo”]` es prerrequisito (requerido) por `Package[“sudo-ldap”` y, por ello, el recurso `Package[“sudo”]` se debe instalar en primer lugar. La utilización del metaparámetro require nos sirve para definir el flujo y precedencia de ejecución de los diferentes recursos que hayamos definido.

Las relaciones en Puppet son una parte fundamental de la definición de la configuración, ya que reflejan las relaciones existentes en el mundo real entre los distintos componentes de configuración de las máquinas. Por ejemplo, pensemos en la configuración de red, de la que dependen una gran cantidad de los recursos que podemos configurar en las máquinas, tales como un servidor web o un agente de transferencia de correo (MTA), que necesitan que la red esté configurada y activa antes de poder ejecutarse. Las definiciones de relaciones entre estos recursos nos permiten especificar que, por ejemplo, los que se utilizan para configurar la red, puedan procesarse antes que los que se encargan de configurar el servidor web o MTA.

El empleo de las relaciones en Puppet va más allá, al permitirnos también definir relaciones desencadenantes entre recursos, tal como indicarle a Puppet que reinicie un recurso de servicio cuando se ha producido un cambio en un recurso de fichero que afecta a ese servicio. Esto es, puedes modificar el fichero de configuración de un servicio y definir la relación para que el cambio desencadene un reinicio del propio servicio y asegurarnos así de que se ejecuta con la última configuración.

En el siguiente enlace puedes ver una lista completa de los metaparámetros de Puppet: https://puppet.com/docs/puppet/latest/metaparameter.html

---

El último recurso que hemos incorporado a la clase sudo es un recurso de archivo, `File["/etc/sudoers"]`, que gestiona el archivo “`/etc/sudoers`”. Los tres primeros

atributos permiten definir respectivamente el propietario (en este caso, el propietario es el usuario `root`), el grupo (`root`) y los permisos del fichero (modo 0440, en notación octal).

El cuarto atributo, `source`, especifica la ruta del fichero origen que queremos enviarle al cliente, por lo que Puppet recuperará ese fichero del servidor y lo enviará al nodo. Este atributo tiene como valor el nombre del servidor de ficheros Puppet y la ruta junto con el nombre del fichero que se quiere copiar:

```
puppet://$servidorpuppet/modules/sudo/etc/sudoers
```

Este valor se compone de las siguientes partes: la parte `puppet://` indica a Puppet que debe usar su protocolo de servidor de ficheros para obtener el fichero, mientras que la variable `$servidorpuppet`, que habíamos definido previamente en el fichero site.pp, tiene como valor el nombre de host del Puppet Master. En vez de hacer referencia a la variable, se podría haber usado directamente el nombre del host del servidor de ficheros:

```
puppet://puppet.ejemplo.edu/modules/sudo/etc/sudoers
```

Una manera útil de abreviar esto es simplemente omitiendo el nombre del servidor, con lo que haremos que Puppet utilice el mismo servidor al que el cliente esté conectado actualmente:

```
puppet:///modules/sudo/etc/sudoers
```

La siguiente parte del valor `source` indica a Puppet cuál es la ubicación del fichero dentro del servidor, que equivale a la ruta de acceso a un recurso compartido en un servidor de ficheros de red. El primer directorio de la ruta es modules, que indica que el fichero en cuestión pertenece a un módulo, cuyo nombre será lo que aparece a continuación en la ruta, que en nuestro caso es sudo, para finalmente especificar la ubicación dentro del módulo donde se encuentra el fichero.

En Puppet, los ficheros que se incluyen en los módulos se almacenan siempre en el subdirectorio `files` del módulo, que es considerado la 'raíz' de las ubicaciones de ficheros, por lo que no hay que incluirlo en la ruta de acceso a un fichero del módulo (nótese que en nuestro caso el directorio `files` no se especifica en la ruta). En nuestro ejemplo, estaríamos accediendo al fichero `sudoers` dentro del subdirectorio `etc` que se encuentra dentro del directorio `files`. Así crearíamos estos recursos:

```
puppet$ mkdir –p /etc/puppet/modules/sudo/files/etc
```

```
puppet$ cp /etc/sudoers /etc/puppet/manifests/files/etc/sudoers
```

Control de versiones

En la práctica profesional, la configuración suele hacerse más compleja con el paso del tiempo, y por ello deberías considerar incluirla en un sistema de control de versiones, tal como *Git*. Este sistema permite a los usuarios realizar cambios, guardarlos, y realizar un seguimiento de los cambios que otros usuarios han hecho en los archivos. Es una herramienta sumamente útil y provechosa y, sobre todo, muy utilizada por los desarrolladores de *software* para controlar las versiones del código fuente.

El control de versiones también es útil para la gestión de la configuración, ya que permite dar seguimiento a los cambios, volver a un estado previamente conocido o poder realizar cambios en una rama independiente sin afectar la configuración actual de ejecución. Esto es lo que se denomina **«infraestructura como código».**

Aplicación de la primera configuración

Una vez que hemos creado nuestro primer módulo Puppet, vamos a ver a continuación qué es lo que pasa cuando se conecta un agente que lo incorpora a su configuración.

- Se asegurará de que esté instalado el paquete `sudo`.
- En el caso de una máquina Ubuntu, también se asegurará de que esté instalado el paquete `sudo-ldap`.
- Copiará el archivo sudoers en la ruta `/etc/sudoers`.

Para pasar a la acción, ahora incluiremos nuestro módulo sudo en el agente que habíamos definido mediante la sentencia de nodo que especificamos para nodo1.ejemplo.edu. Lo incluimos en la sentencia de la siguiente forma:

```
node 'nodo1.ejemplo.edu' {

include sudo

}
```

Cuando el agente se conecte al servidor, detectará que su configuración ha cambiado y que ahora incluye el módulo `sudo`, por lo que aplicará los cambios de configuración correspondientes. Vamos a ejecutar el agente Puppet nuevamente para que esto ocurra, mediante:

```
puppet# puppet agent --server=puppet.ejemplo.edu --no-daemonize--
verbose --onetime
```

Nota: Puppet tiene un modo muy útil que se llama `noop`, que ejecuta Puppet sin realizar ningún cambio en el *host*, lo cual nos permite ver las acciones que realizaría Puppet, como un ensayo de la ejecución *(dry run)*. Si deseas ejecutar en este modo, debes especificar el parámetro `--noop` en la línea de comandos.

En el siguiente listado veremos una secuencia de cómo hemos ejecutado el agente de Puppet y conectado al Master. El agente lo hemos ejecutado en primer plano (no en modo *daemon*), con salida detallada y con la opción `--onetime`, para indicarle al

agente de Puppet que se ejecute una sola vez y a continuación se pare. Podemos ver el inicio de la salida de la ejecución de la configuración en nuestro *host*:

```
notice: Starting Puppet client version 6.4
```

```
info: Caching catalog for nodo1.ejemplo.edu
```

```
info: Applying configuration version '1272631279'
```

```
notice: //sudo/Package[sudo]/ensure: created
```

```
notice: //sudo/File[/etc/sudoers]/checksum: checksum changed
'{md5}9f95a522f5265b7e7945ff65369acdd2'to'{md5}d657d8d55ecdf88a2d11da7
3ac5662a4'
```

```
info: Filebucket[/var/lib/puppet/clientbucket]: Adding
```

```
/etc/sudoers(d657d8d55ecdf88a2d11da73ac5662a4)
```

```
info: //sudo/File[/etc/sudoers]: Filebucketed /etc/sudoers to
puppet with sum⊃
```

```
d657d8d55ecdf88a2d11da73ac5662a4
```

```
notice: //sudo/File[/etc/sudoers]/content: content changed
'{md5}d657d8d55ecdf88a2d11da73ac5662a4'to'{md5}9f95a522f5265b7e7945ff6
5369acdd2'
```

```
notice: Finished catalog run in 10.54 seconds
```

Nota: el conjunto de la configuración que se va a aplicar sobre una máquina en Puppet lo llamamos «catálogo», y «ejecución» al proceso de aplicarla.

Puedes encontrar un glosario de terminología de Puppet en:

https://puppet.com/docs/puppet/latest/glossary.html

Analicemos qué es lo que ha sucedido durante nuestra ejecución. Lo que el agente hace en primer lugar es guardarse en caché la configuración para su máquina. Por defecto, el agente de Puppet usará esta configuración desde la caché cuando no pueda conectarse al Master en futuras ejecuciones.

Lo siguiente que se puede apreciar en la salida de la ejecución del agente es el resultado de la aplicación de nuestros recursos. En primer lugar, se ha instalado el paquete sudo y a continuación se ha copiado el archivo `/etc/sudoers`. Una acción que realiza Puppet durante el proceso de copia de ficheros es la de realizar una copia de seguridad de la versión previa del fichero; a esta operación se la denomina *bucketing*, y nos permite recuperar el fichero antiguo si nos damos cuenta de que hemos sobrescrito el fichero de forma equivocada. El tipo `filebucket`también se puede utilizar sobre un recurso en la máquina remota para indicarle a Puppet que haga una copia de seguridad de un fichero.

Puedes consultar más información sobre este tipo de recurso en:

https://puppet.com/docs/puppet/latest/types/filebucket.html

Finalmente, la última línea de la salida de nuestra ejecución del catálogo indica la duración del proceso, que en este caso fue de 10.54 segundos.

Si ahora consultamos el Puppet Master, veremos cómo se ha registrado el resultado de la ejecución en esta parte:

```
notice: Starting Puppet server version 6.16
```

```
info: Autoloaded module sudo
```

```
info: Expiring the node cache of nodo1.ejemplo.edu
```

```
info: Not using expired node for nodo1.ejemplo.edu from cache;
expired at Wed May 15 11:25:13 +0100 2018
```

```
info: Caching node for nodo1.ejemplo.edu
```

```
notice: Compiled catalog for nodo1.ejemplo.edu in 0.03 seconds
```

En este fragmento se puede apreciar cómo Puppet ha cargado el módulo sudo, ha comprobado que los datos de caché han expirado, por lo que no los está usando y por tanto el resultado del proceso va a actualizar la caché, y finalmente ha compilado el catálogo para `nodo1.ejemplo.edu`. A continuación, este catálogo que se ha compilado se envía al agente para que este lo aplique sobre su máquina.

En caso de que el agente de Puppet se esté ejecutando en modo *daemon*, esperará el tiempo oportuno, que por defecto son 30 minutos, para volver a conectarse al Master y verificar si hay algún cambio de configuración para su nodo, o algún elemento nuevo de configuración que deba aplicar. El tiempo de intervalo periódico de esta consulta al Master se puede personalizar mediante la opción `runintervalen` el fichero de configuración del agente `/etc/puppet/puppet.conf` ubicado en el nodo donde se ejecuta.

```
[agent] runinterval=3600
```

Aquí se muestra ajustado el intervalo a 3600 segundos, o 60 minutos.

2.5. Instalación de Apache

Ahora que ya hemos visto cómo hacer una primera configuración con Puppet, vamos a desarrollar un ejemplo más completo de uso mediante la instalación y configuración básica de un servidor Apache.

Configuración de un entorno de pruebas con Vagrant

Vamos a preparar un entorno Vagrant con una máquina virtual para desarrollar el ejemplo, y poderlo provisionar de una manera sencilla, esta vez con Puppet. Ejecuta los siguientes comandos en una consola de terminal:

```
mkdir puppet-apache && cd puppet-apache
```

```
vagrant init ubuntu/bionic64
```

Necesitaremos configurar la red, con una dirección IP privada que establezcamos nosotros mismos, así como asignar mayor cantidad de memoria a la máquina virtual, para que tenga más recursos y la ejecución de Puppet y las diferentes operaciones de configuración sean más fluidas. Esto se consigue con el siguiente fragmento:

```
 config.vm.network "private_network", ip: "192.168.33.30"
```

```
 config.vm.provider "virtualbox" do |vb|
```

```
vb.memory = "1024"
```

```
 end
```

Ahora tenemos que establecer el mecanismo de aprovisionamiento para utilizar Puppet. Vamos a asumir que la máquina que estas utilizando no tiene Puppet instalado, por lo que lo instalaremos en la propia máquina virtual que Vagrant va a crear (a diferencia de lo que ocurre con Ansible, Vagrant no realiza automáticamente

la instalación de Puppet si lo utilizamos como aprovisionador). Con el objeto de automatizar esta instalación, vamos a incluir una sección de aprovisionamiento *shell* que permite ejecutar comandos *shell*, o un *script*, en la máquina virtual.

```
config.vm.provision "shell", inline: <<-SHELL

wget https://apt.puppetlabs.com/puppet6-release-bionic.deb

sudo dpkg -i puppet6-release-bionic.deb

sudo apt-get update

sudo apt-get install -y puppet-agent

SHELL
```

Este fragmento ejecuta los comandos necesarios para realizar la instalación de Puppet, configurando el repositorio de paquetes oficial de la versión 6 y utilizando el gestor de paquetes de Ubuntu `apt` para instalar el agente de Puppet.

Hemos instalado el agente, pero no tenemos Puppet Master al que conectarnos. Por ello, utilizaremos `puppet apply` como método de aprovisionamiento Puppet, lo que nos permitirá ejecutar Puppet en la máquina independientemente, sin tener un Puppet Master.

Por último, en el fichero `Vagrantfile` incluimos la sección de aprovisionamiento propia de Puppet, para ejecutar un *manifest* que desencadene la instalación:

```
config.vm.provision "puppet" do |puppet|

puppet.module_path = "modules"

puppet.manifests_path = "manifests" # Default

puppet.manifest_file = "default.pp" # Default

end
```

El primer parámetro del provisionador Puppet indica la ruta local donde están almacenados nuestros módulos, para poder hacer uso desde el fichero de manifiesto que se ejecute. Los valores establecidos para los dos siguientes parámetros son los valores por defecto, pero hemos querido incluirlos para mostrar las opciones más comunes de configuración a la hora de establecer un aprovisionamiento con Puppet.

Por tanto, el fichero de configuración Vagrant de nuestra máquina virtual (`Vagrantfile`) quedará como sigue:

```
Vagrant.configure(2) do |config|

config.vm.box = "ubuntu/bionic64"

config.vm.network "private_network", ip: "192.168.33.10"

config.vm.provider "virtualbox" do |vb|

vb.memory = "1024"

end
```

```
config.vm.provision "shell", inline: <<-SHELL

wget https://apt.puppetlabs.com/puppet6-release-bionic.deb

sudo dpkg -i puppet6-release-bionic.deb

sudo apt-get update

sudo apt-get install -y puppet-agent

SHELL

config.vm.provision "puppet" do |puppet|

puppet.module_path = "modules" # Default

puppet.manifests_path = "manifests" # Default

puppet.manifest_file = "default.pp" # Default

end

end
```

Una vez que hemos creado el fichero de configuración Vagrant, ya podemos preparar los ficheros Puppet para el aprovisionamiento. Lo describiremos a lo largo de la siguiente sección.

Creación del archivo init.pp

Tal como hemos especificado en el fichero de configuración de Vagrant, necesitamos crear las rutas y ficheros que Vagrant va a buscar para aprovisionar mediante Puppet. Ejecuta los siguientes comandos desde el directorio de tu `Vagranfile`:

```
mkdir manifests && mkdir modules
```

```
touch manifests/default.pp
```

```
Ahora vamos a editar el manifiesto con nuestro editor favorito en
incluir lo siguiente:
```

```
$document_root = '/vagrant'
```

```
include apache
```

Hemos definido una variable denominada `document_root` con el valor '`/vagrant`' y a continuación hemos incluido el módulo apache, que será el que definamos a continuación.

Ya tenemos creado el directorio de los módulos, por lo que ahora tenemos que crear el subdirectorio correspondiente a nuestro nuevo módulo apache, y ubicar su manifiesto en la ruta apropiada del módulo:

```
mkdir -p modules/apache/manifests
```

```
cd modules/apache
```

```
touch manifests/init.pp
```

Aparte del directorio del módulo, hemos creado el directorio manifests es incluido ahí el fichero `init.pp`, que será el punto de entrada al módulo. Nos hemos situado en el directorio del módulo apache, para que sea más cómodo a partir de ahora crear y manejar los ficheros del módulo.

El siguiente fragmento muestra el contenido de partida que debemos añadir al fichero `init.pp` de nuestro módulo:

```
class apache {
```

```
exec { 'apt-update':

command => '/usr/bin/apt-get update'

}

Exec["apt-update"] -> Package <| |>

package { 'apache2':

ensure => installed,

}

}
```

Hemos definido la clase apache con dos recursos, y una relación de dependencia. El primero de los recursos es de tipo `exec` y nos permite ejecutar un comando directamente en la máquina remota. En este caso, lo usamos para actualizar la caché del gestor de paquetes `apt`, antes de instalar el paquete `apache2`, que es el otro recurso que hemos incluido. Por último, la relación de dependencia indica que para todo recurso de tipo `package` se debe ejecutar antes el recurso `apt-update`, con lo que aseguramos que la caché del gestor de paquetes va a estar siempre actualizada antes de instalar o manejar cualquier paquete.

Con todo esto, vamos a hacer que se instale Apache en nuestra máquina virtual con la configuración por defecto cuando se aprovisione, por lo que si ejecutamos ahora vagrant up desde la ruta del fichero Vagrant podremos comprobar que, una vez creada la máquina, el primer aprovisionador instalará el agente Puppet, y a continuación el aprovisionador Puppet instalará Apache. Una vez finalizado el aprovisionamiento, podremos acceder desde un navegador a la dirección http://192.168.33.10 y comprobar cómo nos responde Apache con la página por

defecto.

Vamos ahora a sustituir esa página por defecto, para lo cual tendremos que cambiar la configuración del servidor Apache para indicarle qué página utilizar, además de proporcionar la página en cuestión. Añadimos lo siguiente a nuestra clase apache, justo antes de la llave de cierre de la clase:

```
file { '/etc/apache2/sites-enabled/000-default.conf':

ensure => absent,

require => Package['apache2'],

}

file { '/etc/apache2/sites-available/vagrant.conf':

content => template('apache/virtual-hosts.conf.erb'),

require => File['/etc/apache2/sites-enabled/000-default.conf'],

}

file { "/etc/apache2/sites-enabled/vagrant.conf":

ensure => link,

target => "/etc/apache2/sites-available/vagrant.conf",

require => File['/etc/apache2/sites-available/vagrant.conf'],

}

file { "${document_root}/index.html":
```

```
ensure => present,

source => 'puppet:///modules/apache/index.html',

require => File['/etc/apache2/sites-enabled/vagrant.conf'],

}
```

Como podemos ver, se trata de cuatro recursos de tipo fichero, aunque con distintas funciones. Analicemos lo que hace cada uno:

- El primero se encarga de borrar el fichero de configuración por defecto de Apache (`000-default.conf`) del directorio de sitios habilitados mediante el atributo `ensure => absent`, que se asegura de que el fichero está ausente, por lo que lo borrará si existe.

- El segundo crea el nuevo fichero de configuración (`vagrant.conf`). El contenido lo genera a partir de una plantilla, mediante el uso de la función template sobre el fichero origen: `template('virtual-hosts.conf.erb')`. El formato de los ficheros de plantilla es el formato erb de plantillas Ruby, aunque también podríamos haber usado el formato de plantillas epp, mediante la función `epp`. La ruta del fichero que se pasa a la función se especifica con modulo/plantilla, no teniendo que especificar el subdirectorio templates, pues es la ruta dentro del módulo en la que se buscará la plantilla.

- El tercero crea un enlace simbólico en el directorio `sites-enabled` al fichero de configuración que acabamos de generar en el segundo paso, que está ubicado en el directorio sites-available. Esta es una característica de Apache, que suele mantener un directorio de ficheros de configuración para los sitios disponibles (`sites-available`), pero solo se encuentran activos los que estén incluidos en el directorio sites-enabled, por lo que lo más adecuado es crear un enlace, en lugar de duplicar el fichero.

- El cuarto fichero se refiere a la página HTML (`index.html`) que vamos a mostrar como página por defecto, que se copiará desde el servidor de ficheros de Puppet a la ruta destino, que está definida por la variable que habíamos especificado como raíz de nuestro sitio web (`${document_root}`) en nuestro fichero *manifest* inicial default.pp. Como fichero origen, la ruta que se especifica, puppet:///modules/apache/index.html, indica al servidor de ficheros Puppet que el fichero `index.html` está en el módulo apache. El directorio files dentro del módulo se sobreentiende y no hay que especificarlo en la ruta.

Ficheros adicionales

- Hemos hecho referencia en los recursos de fichero anteriores a un par de ficheros que se deben suministrar, la plantilla para generar el fichero de configuración y la página HTML por defecto. Estos ficheros tendrán que estar incluidos en el módulo, en sus rutas correspondientes, para que Puppet los pueda encontrar y utilizar en cada caso. El fichero de plantilla se ubicará dentro del subdirectorio templates del módulo, mientras que el fichero HTML estará en el subdirectorio files. Los siguientes comandos ejecutados desde el directorio del módulo apache generarán los directorios y los ficheros correspondientes.

```
mkdir files && mkdir templates
```

```
touch files/index.html && touch templates/virtual-hosts.conf.erb
```

El fichero de plantilla (`virtual-hosts.conf.erb`) tendrá el siguiente contenido:

```
<VirtualHost *:80>
```

```
 ServerAdmin webmaster@localhost
```

```
 DocumentRoot <%= @document_root %>
```

```
 <Directory <%= @document_root %>>
```

```
Require all granted
</Directory>
ErrorLog ${APACHE_LOG_DIR}/error.log
CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

Como podemos apreciar, aparece un par de veces la referencia a la variable `document_root` que habíamos definido, mediante la sintaxis de plantillas Ruby:

```
<%= @document_root %>
```

Esto le indica a Puppet que debe sustituir esa referencia por el valor de la variable definida. Cabe señalar que en este caso también podemos ver otro tipo de referencia de variables, tal como `${APACHE_LOG_DIR}`, pero esta sintaxis está referenciando una variable de entorno. Puppet no lo interpretará y lo copiará tal cual al fichero destino, y será Apache quien lo interprete cuando lea el fichero de configuración resultante.

El otro fichero, `index.html`, es un fichero normal y corriente que se copiará tal cual desde el origen a la ruta destino. El contenido puede ser tal como:

```
<html>
<head>
<title>Ejemplo UNIR</title>
</head>
<body>
```

```
<h1>Hola hola, alumnos de UNIR!</h1>
```

```
</body>
```

```
</html>
```

Gestión de servicios

Si ahora ejecutamos `vagrant provision`, veremos que aparece en la salida la ejecución de los nuevos recursos, pero si accedemos a la IP de la máquina desde un navegador no veremos cambios, y se sigue mostrando la página por defecto de Apache. Esto es debido a que debemos reiniciar el servicio de Apache para que vuelva a leer la configuración y tome los cambios. Vamos a incluir lo siguiente dentro de nuestra clase apache:

```
service { 'apache2':
```

```
ensure => running,
```

```
enable => true,
```

```
hasstatus => true,
```

```
restart => "/usr/sbin/apachectl configtest && /usr/sbin/service
apache2 reload",
```

```
}
```

Con el recurso servicio podemos manejar la recarga del servicio de Apache, mediante el comando que se especifica en el atributo `restart`. También es necesario incluir en los recursos que alteran la configuración la notificación a este recurso, para que se ejecute, por lo que vamos a modificar un par de los recursos de fichero para añadirles el atributo `notify`:

```
file { "/etc/apache2/sites-enabled/vagrant.conf":

ensure => link,

target => "/etc/apache2/sites-available/vagrant.conf",

require => File['/etc/apache2/sites-available/vagrant.conf'],

notify => Service['apache2'],

}

file { "${document_root}/index.html":

ensure => present,

source => 'puppet:///modules/apache/index.html',

require => File['/etc/apache2/sites-enabled/vagrant.conf'],

notify => Service['apache2'],

}
```

Ahora, cuando se cree o actualice el fichero `vagrant.conf` de configuración de Apache y/o se modifique el fichero `index.html` (cualquiera de los dos, o ambos), se notificará al recurso de tipo servicio que recarga Apache. Aunque se produzcan varias notificaciones, el servicio solo se reiniciará una vez. Dado que esos ficheros ya existen, si ejecutamos vagrant provision nuevamente no van a cambiar, por lo que no se notificará al servicio. Es un buen momento para realizar todo el proceso desde cero y validar que todo funciona correctamente, así que procedemos a ejecutar `vagrant destroy` y confirmar la acción, para posteriormente ejecutar `vagrant up` y crear una nueva máquina virtual. A continuación, se muestra el contenido completo

de la clase apache, una vez realizadas todas las modificaciones:

```
class apache {

 exec { 'apt-update':

 command => '/usr/bin/apt-get update'

 }

 Exec["apt-update"] -> Package <| |>

 package { 'apache2':

 ensure => installed,

 }

 file { '/etc/apache2/sites-enabled/000-default.conf':

 ensure => absent,

 require => Package['apache2'],

 }

 file { '/etc/apache2/sites-available/vagrant.conf':

 content => template('apache/virtual-hosts.conf.erb'),

 require => File['/etc/apache2/sites-enabled/000-default.conf'],

 }

 file { "/etc/apache2/sites-enabled/vagrant.conf":
```

```
ensure => link,
target => "/etc/apache2/sites-available/vagrant.conf",
require => File['/etc/apache2/sites-available/vagrant.conf'],
notify => Service['apache2'],
}
file { "${document_root}/index.html":
ensure => present,
source => 'puppet:///modules/apache/index.html',
require => File['/etc/apache2/sites-enabled/vagrant.conf'],
notify => Service['apache2'],
}
service { 'apache2':
ensure => running,
enable => true,
hasstatus => true,
restart => "/usr/sbin/apachectl configtest && /usr/sbin/service
apache2 reload",
}
```

```
}
```

```
Uso de unless y variables de facts
```

Para demostrar un par de funcionalidades adicionales de Puppet, vamos a incluir dos recursos más a nuestro fichero inicial `default.pp`, que quedaría de la siguiente manera:

```
$document_root = '/vagrant'

include apache

exec { 'Skip Message':

 command => "echo ‘Este mensaje solo se muestra si no se ha
copiado el fichero index.html'",

 unless => "test -f ${document_root}/index.html",

 path => "/bin:/sbin:/usr/bin:/usr/sbin",

}

notify { 'Showing machine Facts':

 message => "Machine with ${::memory['system']['total']} of memory
and $::processorcount processor/s.

 Please check access to http://$::ipaddress_enp0s8}",

}
```

El primero de estos dos recursos tiene el atributo `unless` para condicionar su ejecución. Si nos fijamos en la salida de la ejecución, este comando que muestra un mensaje no se ha ejecutado, ya que se cumple la condición especificada mediante

`unless`, por lo que se omite. El segundo recurso añadido muestra un mensaje, en el cual se han incluido algunas variables con valores de *facts*: memoria de la máquina virtual, número de procesadores y dirección IP (utilizando el interfaz `enp0s8` que define VirtualBox para la IP que hemos definido desde Vagrant).

Si volvemos a ejecutar `vagrant provision`, comprobaremos que la configuración de nuestra máquina no varía (ya se había realizado previamente), y de los dos nuevos recursos solo veremos salida del segundo, mostrando el mensaje con el valor de los *facts* del sistema.

2.6. Referencias bibliográficas

Puppet. (2020). *Open-source Puppet documentation.* https://puppet.com/docs/puppet/latest/puppet_index.html

Rhett, J. (2015). *Learning Puppet 4.* O'Reilly.

Uphill, T. (2014). *Mastering Puppet*. Packt Publishing.

Referencia sobre sentencias y expresiones condicionales en Puppet

Puppet. (2020). https://puppet.com/docs/puppet/latest/lang_conditional.html

Enlace directo a la documentación de referencia sobre expresiones y sentencias condicionales en Puppet: `if`, `unless`, `case` y `selector`.

Referencia sobre cadenas de texto en Puppet

Puppet. (2020). https://puppet.com/docs/puppet/latest/lang_data_string.html

En este enlace se encuentra la referencia oficial sobre las cadenas de texto en Puppet. Puedes consultar su sintaxis, manejo, tipos diferentes de cadenas (declaradas con simples comillas o con dobles), secuencias de escape, etc. Todo lo que quieras saber sobre manejo de secuencias de caracteres en Puppet.

Referencia sobre expresiones regulares en Puppet

Puppet. (2020). https://puppet.com/docs/puppet/latest/lang_data_regexp.html

Enlace a la documentación oficial sobre expresiones regulares en Puppet, su sintaxis y utilización.

Puppet Forge

Puppet. (2020). https://forge.puppet.com/

Repositorio de módulos de Puppet, donde se puede encontrar el catálogo de módulos compartidos por Puppet, los *partners* y la comunidad de usuarios. Puedes descargar y reutilizar cualquiera de estos módulos, o subir tus contribuciones para que otros usuarios las puedan reutilizar.

Glosario de términos de Puppet

Puppet. (2020). https://puppet.com/docs/puppet/latest/glossary.html

En este enlace puedes encontrar los términos y las definiciones que se utilizan en Puppet, así como enlaces a su explicación más detallada en la propia documentación.

Test

1. ¿Qué es un módulo en Puppet?

A. Un elemento individual de configuración.

B. Una colección de recursos y variables.

C. Una colección de manifiestos que contienen recursos, clases, definiciones, archivos y plantillas.

D. Una colección de clases y sus definiciones.

2. ¿Cómo se define una variable en Puppet?

A. `$nombre: valor`

B. `$nombre = valor`

C. `nombre: $valor`

D. `nombre = $valor`

3. ¿Cómo se pueden definir un grupo de nodos con un nombre de *host* parecido en un recurso `node`?

A. A través de comodines en la definición del nombre.

B. A través de una expresión regular en la definición del nombre.

C. Especificando un nombre de variable.

D. Las opciones A y B son correctas.

4. ¿Dónde estará definida la clase principal de un módulo?

A. En el archivo `manifests/init.pp`

B. En el archivo `manifests/main.pp`

C. En el archivo `main/init.pp`

D. En el archivo `main/main.pp`

5. ¿Cómo podemos declarar que necesitamos que un paquete esté instalado?

   A. Mediante el recurso `yumo apt` con el atributo `ensure => present`

   B. Mediante el recurso `yum` o `apt` con el atributo `installed => true`

   C. Mediante el recurso `package` con el atributo `installed => true`

   D. Mediante el recurso `package` con el atributo `ensure => present`

6. ¿Qué significa el atributo `require => Package[“sudo”]` en un recurso?

   A. Que el paquete sudo debe instalarse después.

   B. Que si el paquete sudo no se ha instalado, no se hace nada.

   C. Que el recurso `Package[“sudo”]` deberá evaluarse primero.

   D. Que el recurso `Package[“sudo”]` deberá evaluarse a continuación.

7. ¿Cómo podemos hacer que Puppet evalúe una plantilla?

   A. Mediante el uso del tipo de recurso `templatecon` el atributo `content`

   B. Mediante la función `template` o epp en el atributo `contentde` un recurso de tipo `file`

   C. Mediante el atributo `contenten` un recurso de tipo `file`

   D. Mediante la función `templateo eppen` el atributo `contentde` un recurso de tipo `template`

8. ¿Cómo nos podemos asegurar de que no exista un fichero con Puppet?

   A. Mediante el uso del tipo de recurso `delete`

   B. Mediante el atributo `ensure => delete` en un recurso de tipo `file`

   C. Mediante el atributo `ensure => absent` en un recurso de tipo `delete`

   D. Mediante el atributo `ensure => absent` en un recurso de tipo `file`

9. ¿Cómo se indica en Puppet que un recurso debe disparar el reinicio de un servicio nginx?

A. Mediante el atributo en el recurso: `requires => Service['nginx']`

B. Mediante el atributo en el servicio: `requires => Resource['nginx']`

C. Mediante el atributo en el servicio: `notify => Resource['nginx']`

D. Mediante el atributo en el recurso: `notify => Service['nginx']`

10. ¿Para qué sirve el atributo `unless` en un recurso de Puppet?

A. Para que el recurso se ejecute el último.

B. Para que se evalúe el recurso si se cumple la condición dada.

C. Para que no se evalúe el recurso si se cumple la condición dada.

D. Para que no se evalúe el recurso si no se cumple la condición dada.