

Maestría en Desarrollo y Operaciones de  
Software

---

# Herramientas de Automatización de Despliegues

Herramientas de Automatización de Despliegues

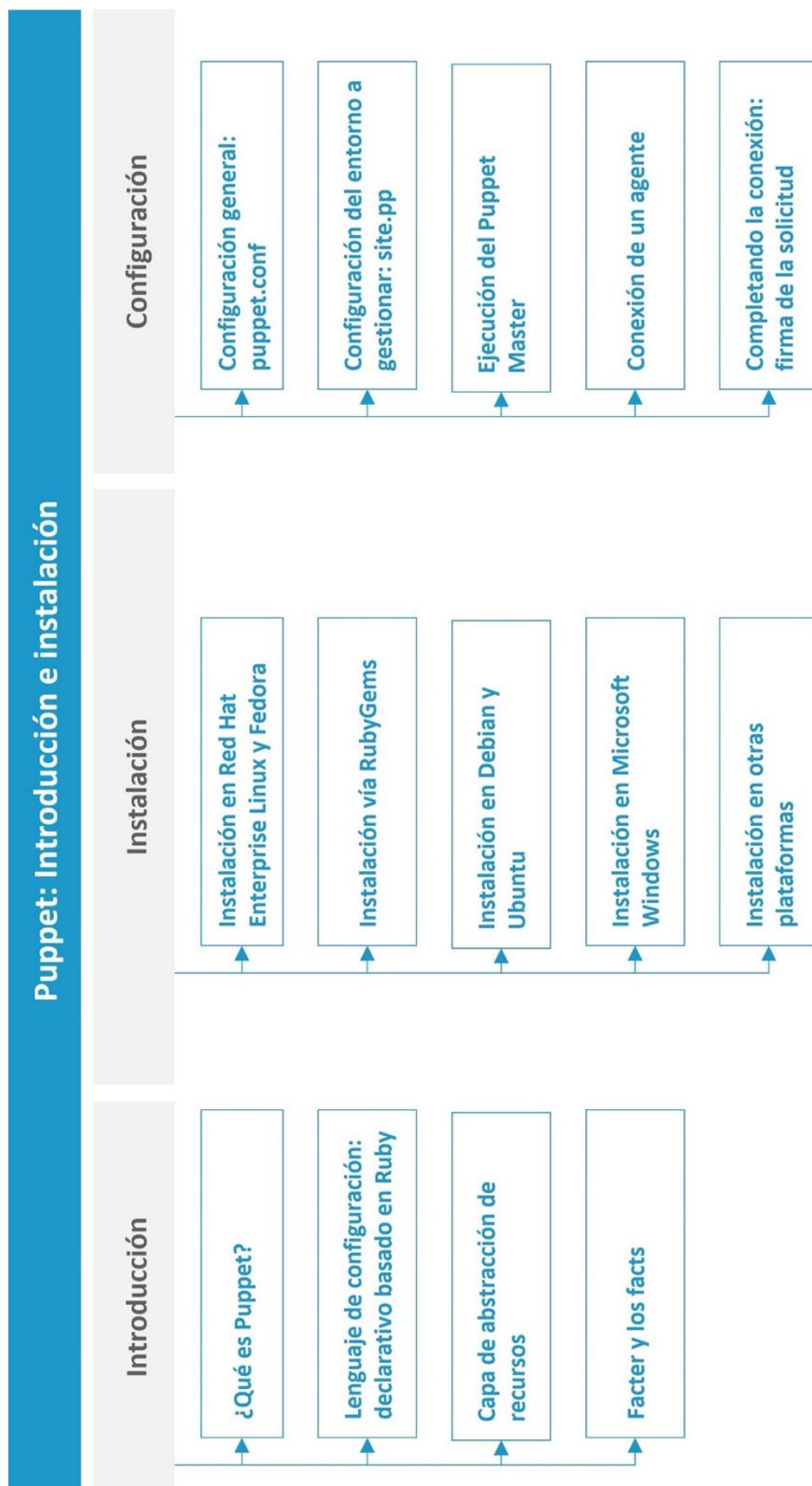
---

# Puppet. Introducción e instalación

# Índice

<b>Esquema</b>	<b>3</b>
<b>Ideas clave</b>	<b>4</b>
1.1. Introducción y objetivos	4
1.2. ¿Qué es Puppet?	5
1.3. Lenguaje de configuración y capa de abstracción de recursos	7
1.4. Facter y los <i>facts</i>	12
1.5. Instalación de Puppet	13
1.6. Configuración de Puppet	17
1.7. Referencias bibliográficas	26

# Esquema



## 1.1. Introducción y objetivos

Puppet es un *framework* de código abierto que incluye a su vez un conjunto de herramientas para la gestión de la configuración de los sistemas informáticos. Puppet proporciona una entrega y funcionamiento estándar de *software* sin importar dónde esté corriendo.

Con el enfoque de Puppet, el usuario define cómo desea que funcionen y se organicen sus aplicaciones e infraestructura, utilizando un lenguaje común y de fácil lectura. Se puede compartir, probar y aplicar los cambios necesarios a través del *datacenter*, con absoluta visibilidad en cada uno de los pasos, así como también obtener informes detallados para tomar decisiones acertadas y documentadas. En este tema vamos a ver cómo utilizar Puppet para administrar la configuración de servidores.

Los objetivos que se pretenden conseguir en este tema son los siguientes:

- ▶ Conocer qué es y cómo funciona Puppet.
- ▶ Conocer el lenguaje de configuración y modelo de abstracción.
- ▶ Interactuar con la herramienta de inventario Facter.
- ▶ Instalar Puppet.
- ▶ Configurar Puppet y el entorno.

## 1.2. ¿Qué es Puppet?

Puppet es una herramienta de gestión de la configuración escrita en lenguaje Ruby con licencia de código abierto GPLv2, aunque también dispone de una versión comercial denominada Enterprise. Se ejecuta habitualmente con un modelo cliente-servidor, aunque también se puede ejecutar en modo *stand-alone* (independiente).

Fue desarrollado por el ingeniero Luke Kanies en la compañía Puppet Labs (anteriormente llamada Reductive Labs). Kanies ha estado trabajando con sistemas Unix y con la administración de sistemas desde el año 1997; y ha desarrollado Puppet por la insatisfacción que le provocaban las herramientas de gestión de la configuración existentes en aquel entonces. En el año 2005 Kanies fundó la empresa Puppet Labs, para el desarrollo de herramientas de automatización de código abierto. Al cabo de poco tiempo, esta empresa lanzó Puppet, su producto principal.

Puppet es capaz de gestionar la configuración de servidores basados en las plataformas UNIX (incluyendo OSX) y Linux, así como servidores basados en Microsoft Windows.

Asimismo, es utilizado habitualmente para la gestión a lo largo de todo el ciclo de vida de las máquinas: comenzando desde la creación y primera instalación, continuando por las actualizaciones y el mantenimiento, y finalizando, al acabar su vida útil, migrando los servicios que proporcionaba a otros sistemas. El diseño de Puppet está pensado para estar en continua interacción con las máquinas que gestiona, al contrario que otras herramientas de aprovisionamiento que únicamente se encargan de la etapa de construcción de las máquinas.

Puppet tiene un modelo de funcionamiento sencillo, fácil de entender y de aplicar, que se compone de tres componentes básicos:

- ▶ Despliegue.

- ▶ Lenguaje de configuración y capa de abstracción de recursos.
- ▶ Capa transaccional.

## Despliegue

El despliegue de Puppet sigue habitualmente un **modelo cliente-servidor**. El servidor recibe el nombre de **Puppet Master**, mientras que el cliente de Puppet que se ejecuta en las máquinas (*hosts*) que se van a gestionar se llama **agente** y el propio *host* se define como un **nodo**.

El proceso de Puppet Master se ejecuta como un *daemon* en el servidor, donde se almacena la configuración necesaria de todo el entorno gestionado.

Los agentes se identifican con el servidor Puppet Master al conectarse y utilizan el estándar SSL para establecer un canal de comunicación cifrado, a través del cual se obtiene la configuración que se va a aplicar.

Cabe destacar que, si el agente no tiene disponible una configuración de Puppet, o ya tiene la configuración requerida, Puppet no hará ningún cambio. Solo realizará cambios en la máquina si, para alcanzar la configuración requerida, es necesario hacerlos, proceso que se conoce como **ejecución de la configuración**.

Cada uno de los agentes Puppet puede estar ejecutándose como un proceso *daemon* (demonio), mediante mecanismos tales como cron, o incluso el agente puede ejecutarse manualmente cuando así se requiera. Lo más habitual es configurar el agente para que se ejecute como un *daemon*, conectándose periódicamente con el servidor para determinar si su configuración está actualizada o, en caso contrario, para descargar y aplicar cualquier cambio requerido en la configuración.

Sin embargo, muchas personas creen que ejecutar el agente Puppet a través de un planificador como cron o manualmente es más adecuado para sus necesidades. El agente de Puppet consultará periódicamente el Puppet Master por si hubiera cambios en su configuración o algo nuevo que aplicar. La periodicidad por defecto de

esta consulta es de 30 minutos, pero este valor puede cambiarse según sean los requisitos del entorno.

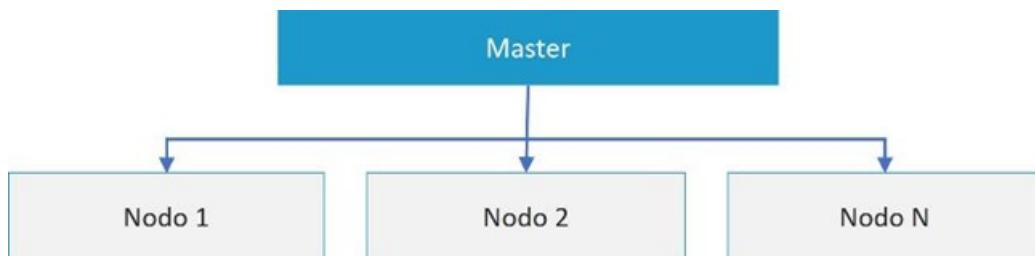


Figura 1. Master y nodos.

También existen otros modelos de despliegue. Se puede ejecutar Puppet en modo independiente, sin contar con una conexión a un Master. La configuración debe encontrarse instalada localmente en la máquina y se ejecuta el binario de Puppet que comprobará y aplicará los cambios necesarios para alcanzar esa configuración.

### 1.3. Lenguaje de configuración y capa de abstracción de recursos

Puppet cuenta con un lenguaje declarativo que sirve para definir los distintos elementos de la configuración, que se denominan **recursos**. Este aspecto declarativo es una característica importante de Puppet con respecto a otras herramientas de configuración. Mediante el lenguaje se declara el estado deseado de la configuración, tal como indicar que un determinado paquete debería estar instalado, o un determinado servicio no debería estar ejecutándose.

En cambio, otras herramientas utilizadas para la gestión de la configuración, tales como Shell o Perl, son de naturaleza imperativa o procedimental: indican las operaciones concretas a realizar, en lugar de declarar el estado final deseado. La mayoría de los scripts personalizados que se implementan para automatizar la configuración de un sistema son un ejemplo del modelo imperativo.

Esto significa que los usuarios de Puppet simplemente declaran el estado deseado de sus *hosts*: qué paquetes deben ser instalados, qué servicios deberían correr, etc. En otras palabras, Puppet se encargará de lograr el estado deseado y los administradores de sistemas abstraerán la configuración de los *hosts* a base de definir los recursos.

## Lenguaje de configuración

Vamos a ver un ejemplo práctico de lo que significa realmente que Puppet utiliza un lenguaje declarativo. Para ello, vamos a considerar lo que implicaría la instalación de la aplicación «vim» en diferentes entornos que cuentan con diferentes sistemas operativos, tales como Red Hat Enterprise Linux (RHEL), Ubuntu y Solaris. Los pasos a continuación indican lo que haría falta implementar en un script para realizar la instalación:

- ▶ Conectar a las máquinas necesarias, con sus usuarios y contraseñas o claves correspondientes.
- ▶ Comprobar si la aplicación «vim» está ya instalada.
- ▶ En caso de no estarlo, usar el comando adecuado en cada sistema operativo para instalar «vim», esto es, utilizar yum en Red Hat o apt-get en caso de Ubuntu.
- ▶ Informar del resultado al usuario para que sepa si la operación se ha completado con éxito.

Por su parte, Puppet se aproxima a este proceso de manera muy diferente. Lo primero que haremos será definir la configuración de recursos para el paquete «vim». Un recurso está compuesto por un **tipo**, que indica la clase de recurso que se está gestionando, como, por ejemplo, paquetes, servicios o ficheros; un **título**, que especifica el nombre del recurso, y una serie de **atributos**, que son los valores que definen el estado del recurso, como, por ejemplo, si el fichero debe estar presente o se debe borrar.

```
package {"vim":  
    ensure => present,  
}
```

En nuestro ejemplo, el recurso define que el paquete denominado «vim» debería estar instalado. Se construye así:

```
type {title:  
    attribute => value,  
}
```

Como se puede ver en la primera secuencia de código, el tipo de recurso es de tipo paquete (package). Puppet incorpora una serie de tipos de recursos de manera predeterminada incluyendo tipos para administrar archivos, servicios, paquetes y trabajos de cron, entre otros.

A continuación, se especifica el título del recurso, que en nuestro caso se trata del nombre del paquete que queremos instalar: «vim». Para utilizar el tipo y el título del recurso como referencia, Puppet nos permite unir el tipo de recurso con la primera letra en mayúscula seguido del nombre entre conchetes, como, por ejemplo: Package["vim"] .

Finalmente, hemos incluido un único atributo ensure, con el valor: present. Los atributos que especifiquemos le dicen a Puppet el estado deseado del recurso de configuración. Cada tipo dispone de una serie de atributos para configurarse. En nuestro caso, el atributo ensure indica el estado deseado del paquete: instalado, desinstalado, etc. El valor present especifica que queremos que el paquete esté instalado, por lo que se procederá a instalarlo si no lo está ya. Para indicar que no queremos que el paquete esté instalado, tendríamos que cambiar el valor de este atributo a absent, lo que lo desinstalaría en caso de estar instalado previamente.

## Capa de abstracción de recursos

Cuando se hayan definido todos los recursos, Puppet será el que se encargue de gestionar los detalles de la configuración del recurso una vez que se conecten los agentes. Se encargará del «cómo», ya que conoce cómo cada tipo de recurso es gestionado en las distintas plataformas y sistemas operativos. Los tipos de recursos constan de distintos proveedores en función de cada plataforma. En el caso de «vim», el proveedor conoce el cómo de la gestión de paquetes en cada plataforma, utilizando la herramienta de gestión de paquetes correspondiente en cada caso. Para el tipo de recurso package, Puppet tiene más de veinte proveedores distintos, que utilizan distintos gestores de paquetes, tales como: yum, aptitude, pkgadd, ports y emerge.

Cuando un agente se conecta, Puppet utiliza una herramienta denominada Facter (que veremos en el apartado siguiente) para devolver información sobre ese agente, incluyendo el sistema operativo que está ejecutando. Puppet se encarga de elegir el proveedor del tipo package que se corresponde con ese sistema operativo y será el proveedor el que verifique si el paquete «vim» se encuentra ya instalado. En Red Hat ejecutaría yum, por ejemplo, mientras que en Ubuntu se ejecutaría apt, y en Solaris, el comando pkg. Si el paquete no está ya instalado, Puppet se encargará de instalarlo a través del proveedor, pero si ya se encuentra instalado, no realizará ninguna acción. Puppet reportará al Puppet Master el resultado de la aplicación del recurso o, en caso de haberse producido un error, la información de este.

## Capa transaccional

La capa transaccional es el motor de Puppet. Se conoce como transacción en Puppet al proceso de configuración de cada máquina, esto es:

- ▶ Interpretación y compilación de la configuración.
- ▶ Transmisión de la configuración compilada a cada agente.

- ▶ Aplicación de la configuración en la máquina del agente.
- ▶ Informe del resultado de la ejecución al Puppet Master.

Lo primero que hace Puppet es el análisis de la configuración para calcular cómo debería aplicarse en el agente, creando para ello un grafo que incluye los distintos recursos y las relaciones entre ellos, así como con cada agente. Esto permite a Puppet establecer en qué orden aplicar cada recurso al *host*, basándose en las relaciones que se crean.

Acto seguido, con los recursos obtenidos previamente, Puppet compila un catálogo de estos para cada agente. Este catálogo es lo que se envía a cada máquina para que el agente de Puppet lo aplique sobre su máquina. Con resultado de esta ejecución, se genera un informe, que es lo que se envía posteriormente de vuelta al Puppet Master.

La capa transaccional permite crear configuraciones y aplicarlas repetidamente en el *host*. A esto lo llama Puppet **idempotencia**, que significa que múltiples aplicaciones del mismo conjunto de operaciones darán lugar a los mismos resultados. La configuración de Puppet se puede aplicar varias veces con seguridad, ya que vamos a obtener el mismo resultado en la máquina, lo que asegura que la configuración va a ser consistente.

Sin embargo, Puppet no es totalmente transaccional: las transacciones no se registran (más allá de los registros informativos) y, por lo tanto, no se pueden deshacer como se hace con algunas bases de datos. Lo que sí podemos hacer es configurar estas transacciones en modo NOOP (del inglés *no operation mode*), lo que nos posibilita el probar la ejecución de la configuración sin realmente realizar ningún cambio.

## 1.4. Facter y los *facts*

**Facter** es la herramienta que incluye Puppet para obtener la información del inventario del sistema. Esta herramienta devuelve los denominados *facts* (hechos), que es información relativa a cada máquina, como su nombre de *host*, dirección o direcciones IP, sistema operativo y su versión; y otra gran cantidad de datos de configuración. Estos datos se recopilan cuando se ejecuta el agente, y son posteriormente enviados al Puppet Master, creándose automáticamente variables para representarlos, lo que los hace disponibles para su uso en Puppet.

Para ver los *facts* que están disponibles en cada máquina, podemos ejecutar directamente en ella el comando `facter` desde la línea de comandos. Cada uno de los *facts* se muestra en forma de par «`clave => valor`» (`key => value pair`), como el ejemplo siguiente:

```
operatingsystem => Ubuntu  
ipaddress => 10.0.0.10
```

Por consiguiente, se pueden utilizar las variables correspondientes para configurar individualmente cada máquina, como por ejemplo la dirección IP, que podemos utilizar para configurar una red en la máquina. Al combinar estas variables de *facts* con la configuración de Puppet, podemos personalizar la configuración en cada una de las máquinas, permitiendo escribir recursos genéricos, como la configuración de red, y personalizarlos con datos de los agentes.

Facter también sirve de ayuda a Puppet para entender cómo debe manejar los recursos de un agente individual, como hemos visto en el apartado anterior. Por ejemplo, cuando Facter detecta que el sistema operativo de la máquina es Ubuntu, se lo dice a Puppet, que entonces sabe que debe usar `apt` para instalar los paquetes en la máquina del agente. Facter permite extensiones, con lo que se puede añadir información personalizada y específica sobre las máquinas.

## 1.5. Instalación de Puppet

Puppet está disponible para instalar en multitud de plataformas, entre las cuales destacamos:

- ▶ Red Hat Enterprise Linux, CentOS, Fedora y Oracle Enterprise Linux.
- ▶ Debian y Ubuntu.
- ▶ Mandrake y Mandriva.
- ▶ Solaris y OpenSolaris.
- ▶ MacOS X y MacOS X Server.
- ▶ BSD.
- ▶ AIX.
- ▶ HP-UX.
- ▶ *Hosts de Microsoft Windows* (únicamente el agente y con recursos de archivo con soporte limitado).

En estas plataformas, Puppet gestiona una variedad de elementos de configuración, incluyendo:

- ▶ Archivos: mediante el recurso `file`
- ▶ Servicios: `service`
- ▶ Paquetes: `package`
- ▶ Usuarios: `user`
- ▶ Grupos: `group`
- ▶ Trabajos Cron (Cron jobs): `cron`
- ▶ Comandos shell: `exec`
- ▶ SSH Keys: `sshkey`, `ssh_authorized_key`

Tanto las instalaciones del agente Puppet como las del servidor Puppet Master son muy semejantes, pero la mayor parte de los sistemas operativos y sus correspondientes gestores de paquetes los distribuyen en paquetes separados. En

algunos sistemas operativos y de distribución será necesario instalar Ruby y sus bibliotecas e incluso algunos paquetes adicionales, aunque la mayor parte de los gestores de paquetes incorporan los paquetes necesarios para instalar Puppet o Facter como prerequisito, tales como Ruby, por lo que se instalarán automáticamente junto con el propio Puppet o Facter.

## Instalación en Red Hat Enterprise Linux y Fedora

En Red Hat Enterprise Linux (RHEL) y derivados basados en Red Hat, como CentOS, es necesario instalar algunos requisitos previos: el lenguaje de programación Ruby, bibliotecas de Ruby y la biblioteca Shadow de Ruby, para permitir a Puppet gestionar usuarios y grupos, lo cual se puede hacer muy fácilmente a través del propio gestor de paquetes de Red Hat: yum.

```
# yum install ruby ruby-libs ruby-shadow
```

Acto seguido, para poder instalar la versión más actualizada de Puppet, tendrás que añadir el repositorio a la máquina, para luego instalar desde ese repositorio los paquetes propios de Puppet. Puedes añadir el repositorio EPEL incluyendo el siguiente comando RPM (.rpm Administrador de paquetes).

```
# rpm -Uvh https://yum.puppet.com/puppet6/puppet6-release-el-7.noarch.rpm
```

En el Puppet Master es necesario instalar el paquete `puppetserver` del repositorio anteriormente configurado:

```
# yum install puppetserver
```

Para instalar un agente en una máquina RHEL, solo se necesitan instalar los prerequisitos y el paquete del agente:

```
# yum install puppet-agent
```

## Instalación vía RubyGems

Tal como la mayor parte de las aplicaciones hechas con Ruby, también es posible instalar las herramientas Puppet y Facter mediante RubyGems. De cara a poder hacerlo de esta manera, necesitas primero instalar el propio Ruby y el paquete RubyGems adecuado a tu sistema operativo, que, para los sistemas basados en Fedora (RHEL, CentOS, Fedora), SUSE (SUSE/SLES) y Debian (Debian y Ubuntu), el paquete se llama `rubygems`. Una vez instalado este paquete, el comando para la gestión de gemas (`gem`) estará accesible para su uso, por lo que puedes utilizarlo para instalar Puppet y Facter:

```
# gem install puppet facter
```

## Instalación en Debian y Ubuntu

En Debian y Ubuntu también tenemos que configurar el repositorio de origen para la instalación de Puppet:

```
# wget https://apt.puppetlabs.com/puppet6-release-bionic.deb  
# dpkg -i puppet6-release-bionic.deb
```

A continuación, puedes instalar los paquetes necesarios para Puppet. En el Puppet Master instalamos la parte servidora:

```
# apt-get install puppetserver
```

En las máquinas donde necesitemos el agente:

```
# apt-get install puppet-agent
```

**Nota:** Tanto la instalación de los paquetes de `puppet-agent` como los de `puppetmaster` instalarán también los prerrequisitos necesarios, como Ruby, si no se encuentra ya instalado.

## Instalación en Microsoft Windows

Puedes instalar el agente de Puppet en máquinas Windows, para poder gestionarlo desde tu Puppet Master.

Se puede realizar una instalación del paquete MSI desde la interfaz gráfica, pero aquí vamos a incluir los comandos necesarios para instalarlo mediante la interfaz de comandos de Windows, lo que te permite personalizar `puppet.conf`, los atributos CSR y otras propiedades del agente. Una vez descargado el paquete MSI, ejecutamos:

```
msiexec /qn /norestart /i <PACKAGE_NAME>.msi
```

Puedes incluir `/l*v <LOG_FILE>.txt` para que se genere el log de la instalación en el fichero indicado.

## Instalación en otras plataformas

Hasta aquí hemos visto cómo instalar Puppet en algunas de las plataformas más populares. Pero, además de las que hemos visto, se puede instalar en muchas otras, tales como:

- ▶ MacOS X vía <http://downloads.puppetlabs.com/mac/>
- ▶ Solaris vía Blastwave.
- ▶ SLES/OpenSuSE vía <http://software.opensuse.org/>
- ▶ Mandrake y Mandriva vía el repositorio Mandriva contrib.
- ▶ FreeBSD vía ports tree.

Los paquetes de código fuente de Puppet también contienen algunos elementos en el directorio `conf`, como puede ser el archivo de especificaciones RPM y scripts de construcción de OS X, que pueden permitir al usuario crear paquetes propios que sean compatibles con el sistema operativo. Ahora que hemos instalado Puppet en alguna de estas plataformas, ya podemos empezar a configurarlo.

En el siguiente vídeo se explica con detalle la instalación de Puppet:



Accede al vídeo

---

## 1.6. Configuración de Puppet

Vamos a describir la configuración del Puppet Master, que será nuestro servidor de configuración, por lo que empezaremos haciendo un repaso a los ficheros de configuración, la configuración de red y el acceso mediante *firewall*, y veremos por último cómo iniciar el Puppet Master. Utilizaremos Puppet en el modo habitual de cliente-servidor, por lo que el Puppet Master contendrá toda la información de la configuración, mientras que los agentes de Puppet se conectarán a través de SSL y descargarán la configuración que les corresponda.

El directorio `/etc/puppet` es donde se guarda la configuración del Puppet Master, en la mayor parte de las plataformas.

El fichero principal de configuración de Puppet se puede encontrar en la siguiente ruta: `/etc/puppet/puppet.conf`. Este fichero se crea habitualmente durante la instalación de Puppet, pero, si no es el caso, el siguiente comando nos permite crear el fichero:

```
# puppetmasterd --genconfig > puppet.conf
```

**Nota:** Aquí asumimos que el directorio /etc/ es el que está usando el sistema operativo para almacenar los archivos de configuración, como ocurre en la mayoría de las distribuciones Unix/Linux. Si estás en otra plataforma que no usa esta ruta, como Microsoft Windows, utiliza la ubicación que corresponda para el fichero puppet.conf.

El fichero puppet.conf tiene una estructura muy similar a los ficheros de configuración de formato INI, ya que está dividido en diferentes secciones. Cada una de las secciones se centra en configurar un aspecto concreto de Puppet. Por ejemplo, tenemos la sección del agente [agent] para establecer la configuración del agente y la sección de Puppet Master [master] para configura el servidor Master. También se encuentra una sección llamada [main] para configuración general. En esta sección se configuran opciones generales para todos los componentes de Puppet.

Por ahora, no añadiremos en el fichero de puppet.conf nada más que una entrada, certname, para especificar el nombre del Puppet Master. Para ello, tenemos que añadir la entrada certname a la sección [master], que habrá que crearla en caso de que no exista todavía:

```
[master]
certname=puppet.ejemplo.edu
```

Debes reemplazar puppet.ejemplo.edu con el nombre correspondiente al dominio completo de tu máquina.

Al añadir esta opción certname y especificar el nombre de dominio completo, estamos facilitando la solución posterior de posibles problemas relacionados con los certificados. Asimismo, es recomendable que se cree una entrada CNAME DNS para el servidor de Puppet, en nuestro caso, puppet.ejemplo.edu y lo incluyas en tu configuración DNS, así como a tu archivo /etc/hosts, que quedaría similar a:

```
127.0.0.1 localhost
```

192.168.0.1 puppet.ejemplo.edu puppet

Ahora que hemos realizado la configuración de DNS adecuada para Puppet, vamos a seguir con el archivo site.pp que debemos añadir, en el que se incluyen los elementos de configuración básicos que vamos a administrar.

## El archivo site.pp

El fichero site.pp es el que contiene las máquinas que se van a gestionar y la configuración que debe aplicar a estas. Este archivo suele almacenarse en el subdirectorio manifests dentro del directorio /etc/puppet/ .

**Nota:** los ficheros que Puppet maneja para la definición de la configuración de los recursos se denominan **ficheros manifests (manifiestos)**. Un fichero manifest de Puppet tiene la extensión .pp.

Puppet habitualmente crea este directorio y el archivo site.pp cuando se instala, pero, si no existen aún, créalos manualmente, ya que, si no encuentra este fichero, Puppet no arrancará:

```
mkdir /etc/puppet/manifests  
touch /etc/puppet/manifests/site.pp
```

Otra consideración que hay que tener en cuenta es que se puede sobrescribir la ubicación y el nombre del fichero manifest mediante el archivo site.pp usando la opción de configuración manifestdir y manifest, respectivamente. Ambas opciones se pueden establecer en el fichero puppet.conf, bajo la sección [master].

El servidor Puppet Master se ejecuta por defecto escuchando el puerto TCP 8140, por lo que el *firewall* que esté configurado en la máquina del Puppet Master debe tener este puerto abierto, así como cualquier otro *firewall* o dispositivo de red que intervenga entre los clientes y el Master, ya que los clientes deben ser capaces de

acceder y conectar a través de ese puerto. Si por configuración se cambiase el puerto de escucha por defecto del Master, habría que hacer lo propio con dicho puerto. El siguiente ejemplo muestra cómo establecer la regla de *firewall* necesaria en el cortafuegos Netfilter para permitir conexiones a través del puerto por defecto:

```
-A INPUT -p tcp -m state --state NEW --dport 8140 -j ACCEPT
```

Esta línea establece la regla que permite el acceso a través del puerto TCP 8140 desde cualquier origen, aunque sería conveniente en la medida de lo posible solo permitir el tráfico entrante desde las redes donde residan las máquinas con los agentes que deben conectar a nuestro Puppet Master, tal como:

```
-A INPUT -p tcp -m state --state NEW -s 192.168.0.0/24 --dport 8140 -j ACCEPT
```

Con esta regla restringimos los orígenes que pueden acceder a través del puerto 8140 a las máquinas cuya IP está dentro de la subred 192.168.0.0/24.

## Inicio de Puppet Master

El Puppet Master puede iniciarse en la mayoría de las distribuciones de Linux mediante un *script init* de inicio de servicio. En Red Hat, ejecutaríamos el script de inicio con el comando *service* de esta forma:

```
# service puppetmaster start
```

En Debian o Ubuntu, lo ejecutamos con el comando *invoke-rc.d*:

```
# invoke-rc.d puppetmaster start
```

Si inicias el *Daemon*, se iniciará el entorno Puppet, se creará una autoridad de certificados locales (*local Certificate Authority*), certificados y claves para el master, y

se abrirá el socket de red adecuado para recibir las conexiones del cliente. Se pueden ver los certificados e información SSL de Puppet en el directorio /etc/puppet/ssl.

```
# ls -l /etc/puppet/ssl
drwxrwx--- 5 puppet puppet 4096 2009-11-16 22:36 ca
drwxr-xr-x 2 puppet root 4096 2009-11-16 22:36 certificate_requests
drwxr-xr-x 2 puppet root 4096 2009-11-16 22:36 certs
-rw-r--r-- 1 puppet root 361 2009-11-16 22:36 crl.pem
drwxr-x--- 2 puppet root 4096 2009-11-16 22:36 private
drwxr-x--- 2 puppet root 4096 2009-11-16 22:36 private_keys
drwxr-xr-x 2 puppet root 4096 2009-11-16 22:36 public_keys
```

El directorio en el Master contiene la autoridad de certificados (*Certificate Authority*), las solicitudes de certificado de los clientes, un certificado para el Master y certificados para todos sus clientes.

**Nota:** se puede sobrescribir la ruta donde se encuentran los archivos SSL mediante la opción ssldir.

También puedes ejecutar Puppet Master desde la línea de comandos para ayudar a probar y depurar errores (*debug*). Es muy recomendable que lo hagas si pruebas Puppet por primera vez. Para ello, inicia el *daemon* de Puppet Master:

```
# puppet master --verbose --no-daemonize
```

La opción *--verbose* muestra la salida de log detallada y la opción *--no-daemonize* mantiene el demonio en el primer plano y redirige el log a la salida estándar. También es posible utilizar el parámetro *--debug* para generar una salida más detallada de depuración del proceso *daemon*.

## Un único binario

Toda la funcionalidad de Puppet está disponible a partir de un único fichero binario, como ocurre en otras herramientas como GIT, en lugar de los binarios individuales para cada función utilizados en las versiones iniciales de Puppet. Por ello, es posible arrancar el Puppet Master de esta forma:

```
# puppet master
```

Y para iniciar la funcionalidad del agente:

```
# puppet agent
```

Puedes ver la lista completa de las funciones disponibles en el comando de Puppet ejecutándolo con el parámetro de ayuda:

```
$ puppet --help
```

## Conexión del primer agente

Ahora que tenemos al servidor Puppet Master configurado y funcionando, es el momento de conectar el primer agente. En la máquina donde vamos a instalar el agente, es necesario instalar los prerequisitos y el paquete adecuado, como ya vimos previamente, mediante el gestor de paquetes correspondiente. A modo de ejemplo, en esta sección instalaremos un agente en la máquina nodo1.ejemplo.edu y la conectaremos con nuestro Master.

Para conectar el primer agente, vamos a ejecutarlo desde la línea de comandos, en vez de ejecutarlo como servicio, para ver más fácilmente lo que va pasando cuando conectamos a través de la salida estándar por consola. El *daemon* del agente de Puppet es ejecutado utilizando el comando `puppet agent` y se puede ver la conexión iniciada con el Master en el siguiente listado (también se puede ejecutar el cliente Puppet en el propio Puppet Master, pero vamos a comenzar con la forma más tradicional cliente-servidor):

```
node1# puppet agent --server=puppet.ejemplo.edu --no-daemonize --verbose
info: Creating a new certificate request for nodo1.ejemplo.edu
info:      Creating      a      new      SSL      key      at
/var/lib/puppet/ssl/private_keys/nodo1.ejemplo.edu.pem
warning: peer certificate won't be verified in this SSL session notice: Did
not receive certificate
```

Nota: Si no se especifica el servidor, Puppet buscará por defecto un *host* llamado *puppet*. Por lo general es recomendable definir un CNAME para el Puppet Master, tal como: `puppet.ejemplo.edu`

Podemos definir esto mismo en el fichero `/etc/puppet/puppet.conf` de configuración del agente, en la sección principal:

```
[main]
server=puppet.ejemplo.edu
```

El agente necesita poderse conectar con el Master, por lo que debe poder resolver su nombre de *host* y tener acceso al puerto de conexión, por eso es necesario, aparte de haber incluido la regla para permitir la conexión al puerto en el *firewall*, tener definido un CNAME para el Puppet Master y especificarlo en el fichero `/etc/hosts` del cliente.

El parámetro `--no-daemonize` nos permite ejecutar el agente Puppet en primer plano, sin crear el proceso *daemon*, y muestra la salida del comando por consola. Si no especificamos este parámetro, por defecto el agente Puppet se ejecutará en modo *daemon*.

Tal como ya hemos mencionado, para autenticar las conexiones entre el Master y los agentes, Puppet hace uso de certificados SSL. En las últimas versiones de Puppet se soporta tanto una arquitectura de CA (autoridad certificadora) simple, con un certificado raíz autofirmado que también se utiliza para firmar, como una

arquitectura de CA intermedia, con un certificado raíz autofirmado que emite un certificado CA intermedio que se utiliza para firmar las peticiones entrantes de certificado. La arquitectura con CA intermedia es la recomendada, ya que es más segura y facilita la regeneración de certificados.

Para generar una CA intermedia para el servidor Puppet se debe ejecutar el siguiente comando antes de iniciar el servidor por primera vez:

```
puppetserver ca setup
```

Cuando un agente Puppet se conecta por primera vez al servidor, envía la solicitud de certificado al Master y espera a que el Master lo firme y devuelva el certificado. Cada dos horas comprobará si recibe el certificado firmado, hasta que se reciba o se cancele su ejecución (usando Ctrl-C, por ejemplo).

**Nota: el tiempo que espera el agente para recibir el certificado se puede configurar mediante la opción –waitforcert, mediante un valor en segundos, o 0 para no esperar y parar de ejecutarse de inmediato.**

## Completando la conexión

El paso necesario que nos queda para completar la conexión y autenticar al nuevo agente es firmar el certificado que el agente envió al Master. Esto se hace mediante `puppet cert` en el Master:

```
puppet# puppetserver ca list  
nodo1.ejemplo.edu
```

La acción `list` muestra todos los certificados que están a la espera de ser firmados, tras lo que procederemos a firmarlo mediante la acción `sign`.

```
puppet# puppetserver ca sign --certname nodo1.ejemplo.edu
```

Signed nodo1.ejemplo.edu

Puppet también permite activar el modo autosign, en el que, en lugar de firmar cada certificado individualmente, todos los certificados provenientes de direcciones IP o rangos de direcciones especificadas se firmarán automáticamente. Esto tiene, obviamente, algunas implicaciones y riesgos de seguridad, por lo que debes asegurarte de que lo utilizas adecuadamente.

En el agente debería mostrarse la siguiente salida un par de minutos después de haber firmado el certificado (o podríamos parar y arrancar de nuevo el agente de Puppet para no tener que esperar):

```
notice: Got signed certificate
notice: Starting Puppet client version 6.4
```

Ya tenemos a nuestro agente autenticado con el Master, aunque ahora recibimos otro mensaje de error:

```
err: Could not retrieve catalog: Could not find default node or by name with
'nodo1.ejemplo.edu , nodo1' on node nodo1.ejemplo.edu
```

El agente está conectado y se ha autenticado correctamente con el Master, pero el mensaje de error nos está indicando que no existe todavía ninguna configuración disponible para la máquina nodo1.ejemplo.edu. El siguiente paso será definir en el Master la configuración que debe utilizar este agente.

**Nota: Las conexiones SSL confían en la exactitud de los relojes de los *hosts*, por lo que, si esta es incorrecta entre el Master y el agente, la conexión puede fallar con un error que indica que los certificados no son de confianza. Para asegurarnos de que la hora de los relojes de las máquinas está sincronizada podemos utilizar NTP (Network Time Protocol).**

## 1.7. Referencias bibliográficas

Puppet. (2020). *Open-source Puppet documentation. Welcome to Puppet 7.8.0.*

[https://puppet.com/docs/puppet/latest/puppet\\_index.html](https://puppet.com/docs/puppet/latest/puppet_index.html)

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

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

Herramientas de Automatización de Despliegues

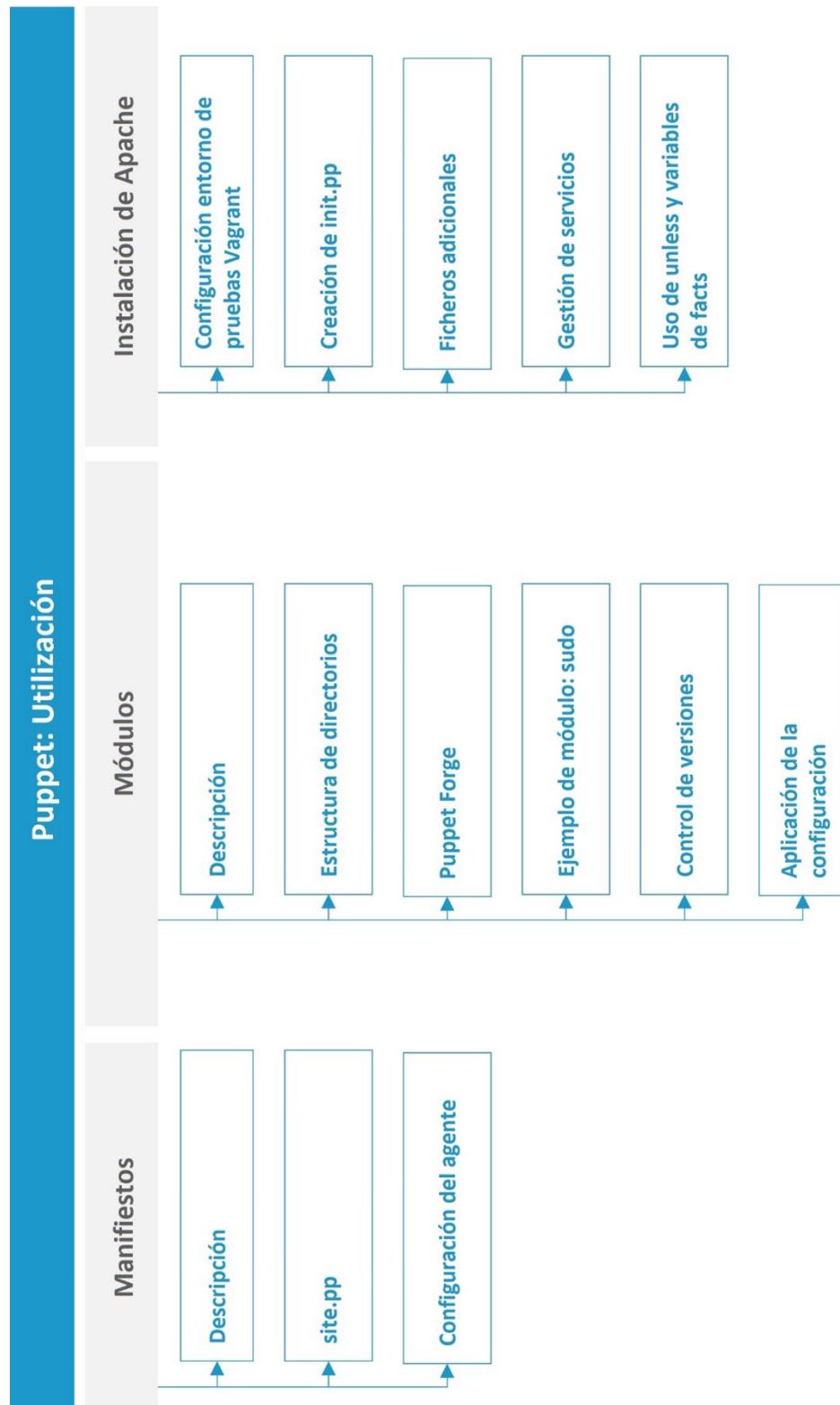
---

# Puppet. Utilización

# Índice

<b>Esquema</b>	<b>3</b>
<b>Ideas clave</b>	<b>4</b>
2.1. Introducción y objetivos	4
2.2. Archivos de manifiesto ( <i>manifest</i> )	4
2.3. Módulos	9
2.4. Ejemplo de módulo: sudo	12
2.5. Instalación de Apache	20
2.6. Referencias bibliográficas	31

# 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.

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



Accede al vídeo

---

## 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](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\\_regex.html](https://puppet.com/docs/puppet/latest/lang_data_regex.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](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 <i>custom facts</i> 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	<i>Plug-ins</i> , tales como <i>facts</i> (en el subdirectorio <i>facter/</i> ) y tipos de recursos, funciones y proveedores ( <i>puppet/</i> ) personalizados
locales	Archivos relativos a la localización del módulo, para otros lenguajes que no sean el inglés
manifests	Los manifiestos del módulo: <i>init.pp</i> 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 <i>spec</i> 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](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 subdirectorío `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.

1. Se asegurará de que esté instalado el paquete `sudo`.

2. En el caso de una máquina Ubuntu, también se asegurará de que esté instalado el paquete sudo-ldap.
3. 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 `runinterval` en 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 estás 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` e 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:

1. 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.
2. 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.

3. 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.
4. 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

5. 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](https://puppet.com/docs/puppet/latest/puppet_index.html)

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

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

Herramientas de Automatización de Despliegues

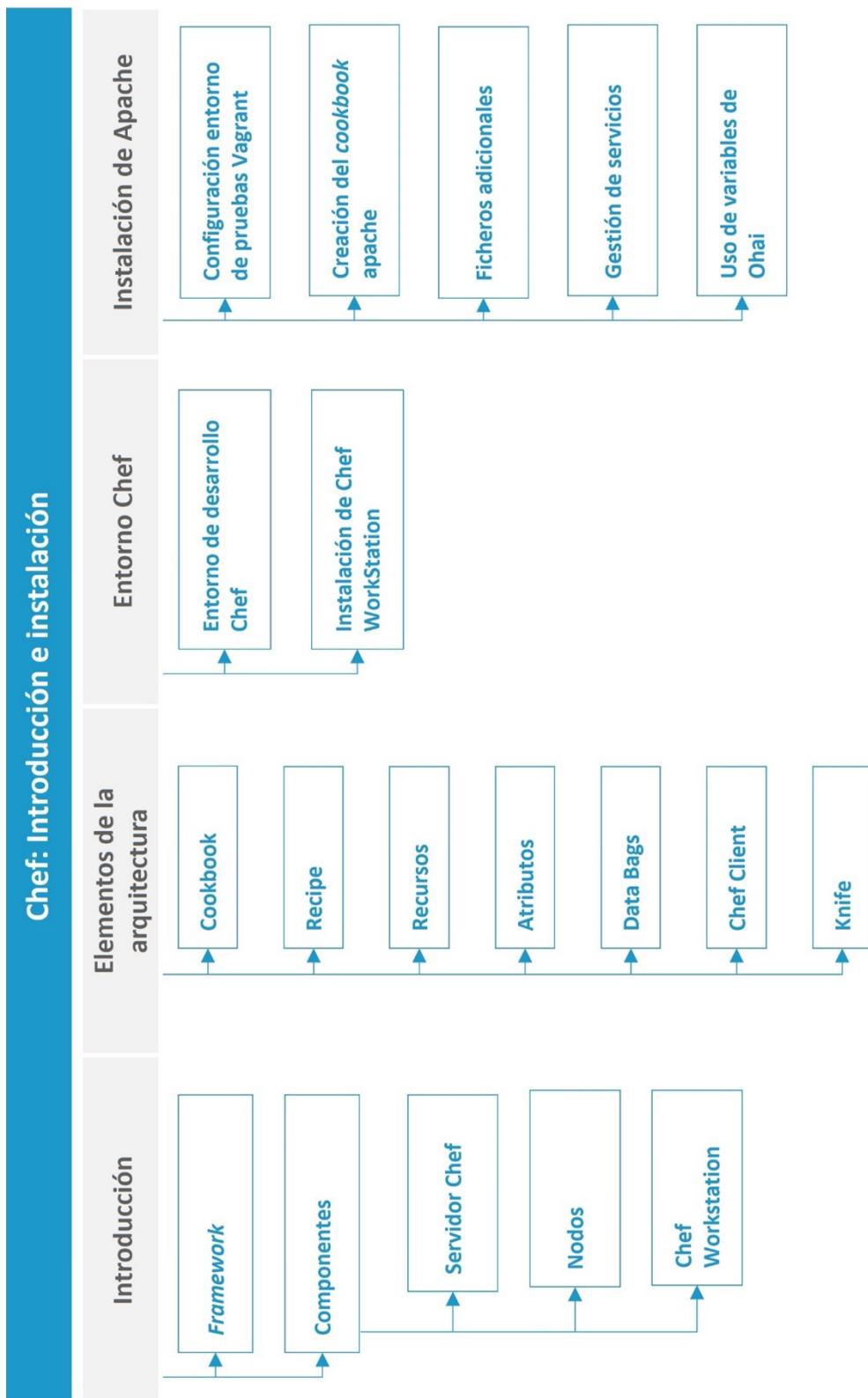
---

# Chef. Introducción e instalación

# Índice

<b>Esquema</b>	<b>3</b>
<b>Ideas clave</b>	<b>4</b>
3.1. Introducción y objetivos	4
3.2. El <i>framework</i> Chef	4
3.3. Elementos de la arquitectura de Chef	7
3.4. Entorno de desarrollo Chef	8
3.5. Instalación de Apache mediante Chef	10
3.6. Chef Supermarket	23
3.7. Referencias bibliográficas	23

# Esquema



## 3.1. Introducción y objetivos

La gestión de la **infraestructura como código** viene a referirse tanto a la escritura de código, mediante un lenguaje descriptivo cualquiera, para el control y aprovisionamiento de servidores y su proceso de configuración, como al despliegue del código de las aplicaciones que se ejecutan en dichos servidores. Se trata además de la **adopción de pruebas automatizadas**, así como otra serie de prácticas que ya se utilizan ampliamente desde hace tiempo en el desarrollo de aplicaciones, entre las que podemos mencionar el control de versiones, las pruebas de integración, el uso de patrones de diseño, etc.

A la hora de aplicar estas mismas prácticas y normas, hemos de asegurarnos de proporcionar y gestionar la configuración de servidores mediante código, para, de esta forma, construir procesos seguros, repetibles y automatizados.

Los objetivos de este tema son los siguientes:

- ▶ Introducir la herramienta Chef.
- ▶ Explicar e instalar su entorno de desarrollo (CDK).
- ▶ Usar Chef para instalar Apache.

## 3.2. El *framework* Chef

En el proceso de maduración del desarrollo de software han surgido los denominados marcos de trabajo, más conocidos por su nombre en inglés: *frameworks*. Su objetivo principal es el de simplificar y reducir el tiempo de desarrollo, permitiendo a los

desarrolladores poder centrarse en el desarrollo específico que cumpla con los requisitos necesarios, sin tener que preocuparse de elementos y servicios de uso común que son proporcionados por el *framework*, y lograr así una entrega más rápida. Estos *frameworks* de software suelen ofrecernos además un conjunto común de convenciones, procedimientos y servicios que tienen como objetivo fomentar que el equipo desarrolle de forma homogénea, facilitando así el desarrollo sostenible del proyecto.

Chef es uno de los *frameworks* más extendidos para la gestión de la infraestructura como código, proporcionando una extensa biblioteca de primitivas y utilidades para la gestión de todos los recursos que se usan en los procesos de construcción y mantenimiento de la infraestructura de sistemas. Chef utiliza un lenguaje específico de dominio (DSL), que está basado en el lenguaje de programación Ruby, y proporciona una capa de abstracción sobre los recursos que permite, tanto a un administrador de sistemas como a un desarrollador, definir y aprovisionar fácilmente entornos escalables.

## Componentes Chef

El *framework* Chef está compuesto por tres componentes básicos que interactúan entre sí: el **servidor Chef**, las máquinas gestionadas denominadas **nodos** y la **estación de trabajo Chef** (*Chef workstation*).

### Servidor Chef

El servidor Chef constituye el **centro de los datos de configuración**, que contiene los *cookbooks*, que son los módulos de configuración de recursos, las políticas que se aplicarán a los diferentes nodos, y los metadatos, que describen cada uno de los nodos administrados por Chef.

## Nodos

Los nodos usan la herramienta **Chef Client** para conectarse con el Chef Server y solicitar los detalles de configuración que deben aplicarse sobre la máquina donde se ejecutan. A este proceso de aplicar los cambios sobre los nodos se le denomina **Chef run**.

Cabe destacar que cuando un nodo se crea y se registra en el Chef Server, el Chef Client se instala en el proceso de arranque, para iniciarse cuando la máquina se levanta.

En Chef, se llama **rol** a un conjunto de atributos y una lista de recetas para el nodo. Cada nodo puede tener uno o más roles, y estos pueden ser reutilizado por varios nodos, por lo que se puede definir un conjunto de nodos que tengan el mismo rol dentro de nuestro sistema. La **lista de ejecución** se refiere a la lista de recetas asociadas con un nodo a través del rol o las propias recetas de las que depende, y el orden de ejecución será el mismo orden en que está definida.

## Chef Workstation

La estación de trabajo Chef, también denominada «repositorio de Chef (Chef-repo)», contiene la estructura del proyecto gestionado por Chef y lo maneja el desarrollador o administrador desde su *workstation*. Todos los componentes de Chef que constituyen nuestro sistema están definidos aquí: *cookbooks*, entornos, roles y pruebas. Una buena práctica que es frecuente es la de mantener el Chef-repo en un sistema de control de versiones, para así gestionarlo como si fuera código fuente de una aplicación.

La estructura de directorios de un Chef-repo puede variar, ya que algunos usuarios prefieren un único Chef-repo para mantener todos los *cookbooks*, mientras que otros prefieren almacenar cada *cookbook* en un repositorio separado, pero, en cualquier

caso, el repositorio Chef (Chef-repo) debe poder gestionar toda la infraestructura y, para ello, debe contener la información de todas sus partes.

### 3.3. Elementos de la arquitectura de Chef

La lista que se enumera a continuación describe los principales elementos de la arquitectura de Chef:

- ▶ **Chef Server:** servidor centralizado que contiene y gestiona la configuración de todos los nodos. Puede estar ubicado en un servidor independiente o alojado como servicio en la nube en [Chef](#).
- ▶ **Nodo:** incluye la información relativa a las recetas y roles que deben aplicarse en la máquina durante la ejecución de Chef Client. Los atributos y la lista de ejecución del nodo son sus principales características.
- ▶ **Cookbook:** contiene todos los recursos e instrucciones que se necesitan para configurar nodos, y pueden ser reutilizados en varias listas de ejecución. Los *cookbooks* se componen habitualmente de varias recetas.
- ▶ **Recipes** (recetas): consiste en un conjunto de recursos que configuran un nodo, y es una de las partes fundamentales de Chef.
- ▶ **Recursos:** son una abstracción multiplataforma de partes configurables de un nodo, como pueden ser usuarios, paquetes, archivos o directorios.
- ▶ **Atributos:** representan la configuración del nodo, como por ejemplo el nombre del *host*, las versiones de los lenguajes de programación para instalar, servidor de base de datos, etc.
- ▶ **Data Bags:** contienen conjuntos de datos que están disponibles globalmente y pueden ser utilizados por los nodos y roles.
- ▶ **Chef Client** (Cliente Chef): realiza el trabajo necesario en nombre del nodo, donde ejecuta las recetas correspondientes para configurar e instalar el *software*.
- ▶ **Repositorio Chef:** componente donde se ubica el repositorio de los *cookbooks*, roles, archivos de configuración y otros artefactos que maneja el usuario.

- ▶ **Chef Solo:** herramienta de línea de comandos que permite ejecutar *cookbooks* Chef independientemente, sin conectarse a un servidor Chef. Es una versión de código abierto del Chef Client.
- ▶ **Knife** (cuchillo): herramienta de línea de comandos que utilizan los usuarios para cargar los ficheros de configuración en el servidor Chef.

## 3.4. Entorno de desarrollo Chef

A continuación, puedes ver el vídeo *Instalación de Chef*:



Accede al vídeo

---

El repositorio Chef es el componente con el que se suele trabajar habitualmente, realizando las siguientes tareas:

- ▶ Desarrollo de *cookbooks* y *recipes* (recetas).
- ▶ Carga de información en el servidor Chef.
- ▶ Sincronización del código fuente del repositorio Chef con el control de versiones.
- ▶ Administración de las opciones de configuración y de los entornos.
- ▶ Interacción y aprovisionamiento de los nodos nuevos y los ya existentes.

El cometido fundamental de una estación de trabajo en una instalación de Chef es el de sincronizar los datos del repositorio Chef con el servidor Chef, así como con todos los nodos con los que interactúa.

Dentro del conjunto estándar de herramientas que incorpora Chef se incluyen dos herramientas fundamentales de línea de comandos: knife (cuchillo), que es la herramienta de línea de comandos que interactúa con el servidor Chef y la sincronización de los nodos, y el propio comando chef, que trabaja con los

componentes del repositorio Chef y la gestión del entorno de Ruby embebido utilizado por Chef.

Además de las herramientas estándar que proporciona Chef, existe una serie de herramientas específicas adicionales que han sido ampliamente adoptadas por la comunidad, tales como:

- ▶ **Berkshelf**: gestor de dependencia de *cookbooks*.
- ▶ **Test Kitchen**: *framework* de pruebas de integración.
- ▶ **ChefSpec**: *framework* de pruebas unitarias.
- ▶ **Foodcritic**: herramienta para la realización de análisis de código estático en *cookbooks*.

## Instalación de Chef Workstation

Anteriormente, de cara a que la instalación de un entorno de desarrollo Chef fuera lo más sencilla posible, contábamos con el **kit de desarrollo de Chef** (ChefDK), elaborado por la propia comunidad Chef. Este proyecto tenía como objetivo recopilar las herramientas Chef más ampliamente adoptadas en un solo paquete de fácil instalación, y con soporte para múltiples plataformas.

Actualmente contamos con Chef Workstation, desarrollada esta vez por la propia organización Chef Software, que incluye todas las funcionalidades de ChefDK además de otras nuevas, y que es el producto que seguirá recibiendo actualizaciones y mejoras en el futuro. Esta nueva herramienta sigue siendo gratuita para uso no comercial, por lo que para nuestro caso es recomendable instalarla, aunque ya contásemos con ChefDK, para obtener las últimas versiones soportadas de las herramientas incorporadas, así como nuevas funcionalidades.

Para instalarlo, basta con descargar la versión correspondiente a nuestro sistema operativo desde la página de descargas e instalar el paquete siguiendo los pasos de instalación.

---

Puedes acceder a la descarga de Chef Workstation a través del enlace:

<https://downloads.chef.io/chef-workstation/>

---

Para revisar todos los componentes instalados por la instalación, podemos ejecutar el comando chef con el argumento -v para que liste las versiones de los componentes:

```
$ chef -v
```

```
Chef Workstation version: 20.6.62
Chef Infra Client version: 16.1.16
Chef InSpec version: 4.19.0
Chef CLI version: 3.0.4
Test Kitchen version: 2.5.1
Cookstyle version: 6.7.3
```

Para obtener instrucciones más detalladas sobre este proceso, puedes consultar la documentación oficial.

Una vez realizada la instalación satisfactoriamente, ya estamos preparados para comenzar a usar Chef para gestionar la infraestructura. Ahora tenemos todas las herramientas necesarias para desarrollar y probar recetas, cargarlas en el servidor y aprovisionar nodos.

### 3.5. Instalación de Apache mediante Chef

De cara a poder demostrar las características fundamentales de Chef para el manejo de la infraestructura como código, vamos a ir desarrollando un ejemplo que se encarga de instalar un servidor HTTP Apache en una máquina virtual aprovisionada con Vagrant, de manera semejante a lo que hemos hecho en temas anteriores.

Lo primero será preparar un entorno de pruebas mediante Vagrant, que nos servirá para generar una máquina virtual que será la que aprovisionemos con Chef, y sobre la que iremos incorporando diferentes recursos hasta completar una instalación de Apache con una configuración personalizada y una página de inicio propia.

## Configuración de un entorno de pruebas con Vagrant

Nuevamente vamos a utilizar Vagrant para preparar nuestro entorno de pruebas con una máquina virtual que utilizaremos para desarrollar el ejemplo, aprovisionándolo fácilmente con Chef a través de los comandos Vagrant. Ejecuta los siguientes comandos desde un terminal:

```
mkdir chef-apache && cd chef-apache  
vagrant init ubuntu/bionic64
```

Vamos a configurar la red con una dirección IP privada que establezcamos nosotros mismos, y a asignar mayor cantidad de memoria a la máquina virtual, para que la ejecución de Chef y las diferentes operaciones de configuración sean más fluidas. Para ello, incluimos el siguiente fragmento en el Vagrantfile:

```
config.vm.network "private_network", ip: "192.168.33.40"  
config.vm.provider "virtualbox" do |vb|  
    vb.memory = "1024"  
end
```

A continuación, incluiremos la configuración necesaria para aprovisionar nuestra máquina virtual con Chef. A diferencia de lo que pasaba con Puppet, Vagrant es capaz de instalar automáticamente Chef en la máquina virtual si detecta que no está instalado y hemos especificado `chef_solo` como el aprovisionador a utilizar. Este aprovisionador permite aprovisionar una máquina con Chef, sin la necesidad de contar con un servidor al que conectar el nodo, por lo que es el más apropiado para nuestro caso.

Incluimos el siguiente fragmento en el Vagrantfile:

```
config.vm.provision :chef_solo do |chef|
  chef.cookbooks_path = "chef-repo/cookbooks"
  chef.add_recipe "chef_apache"
  chef.arguments = "--chef-license accept"
end
```

Aquí estamos indicando que el aprovisionador a usar será `chef_solo`, indicándole la ruta donde almacenamos las recetas y la receta a ejecutar. El último parámetro sirve para especificar argumentos en la línea de comandos al ejecutar Chef, y en este caso nos permite añadir el argumento para aceptar la licencia de Chef, que es necesario realizar al ser una nueva instalación de Chef la que se realiza en la máquina virtual. Esta licencia nos permite usar Chef para aprendizaje y experimentación, y cualquier uso comercial debe contar con un acuerdo de licencia comercial con Chef.

En definitiva, el fichero de configuración Vagrant (Vagrantfile) para nuestra máquina virtual quedará así:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/bionic64"

  config.vm.network "private_network", ip: "192.168.33.40"
  config.vm.provider "virtualbox" do |vb|
    vb.memory = "1024"
  end

  config.vm.provision :chef_solo do |chef|
    chef.cookbooks_path = "chef-repo/cookbooks"
    chef.add_recipe "chef_apache"
    chef.arguments = "--chef-license accept"
  end
end
```

Una vez creado el fichero de configuración Vagrant, ya podemos preparar el *cookbook* Chef necesario para el aprovisionamiento, que describiremos a lo largo de la siguiente sección.

## Creación del *cookbook* apache

Lo primero que vamos a hacer es crear las rutas y ficheros que hemos especificado en el *Vagrantfile* para que cuando ejecute Vagrant se encuentre todos los recursos que necesita para aprovisionar.

```
mkdir -p chef-repo/cookbooks/ && cd chef-repo/cookbooks/
```

Una vez creado el directorio de *cookbooks*, para generar nuestro *cookbook* utilizaremos el comando *chef* correspondiente, que creará toda la estructura de directorios y ficheros que se requiere.

```
chef generate cookbook apache
```

Si es la primera vez que ejecutamos un comando *chef*, nos pedirá que aceptemos la licencia de los productos asociados, para poder comenzar a utilizar sus funcionalidades.

Una vez aceptadas las licencias, o si ya se habían aceptado anteriormente, el comando creará un directorio apache con la siguiente estructura:

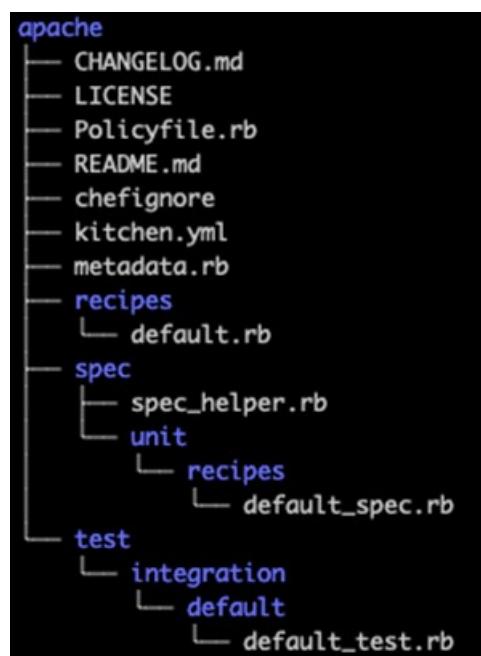


Figura 1. Estructura de directorios de un rol. Fuente: elaboración propia.

En esta estructura se pueden encontrar varios ficheros que se generan por defecto y que nos sirven de plantilla o versión inicial para incorporar funcionalidad a nuestro *cookbook*. El fichero principal donde se especifica la configuración a obtener es la receta por defecto, ubicado en: `recipes/default.rb`, al que vamos a incluir el primer fragmento de código:

```
apt_update 'Update the apt cache daily' do
  frequency 86400
  action :periodic
end

package 'apache2'
```

Aquí hemos definido un recurso de tipo `apt_update` que se encargará de refrescar la caché del gestor de paquetes, y como argumentos le indicamos que sea una acción periódica con una frecuencia de 86 400 segundos (1 día, que es el valor por defecto, pero se ha incluido aquí por claridad). Esto indica a Chef que, si al aplicar la

configuración, esta acción no se ha realizado hace menos de 1 día, se volverá a ejecutar. El siguiente recurso es de tipo package y, si no especificamos ningún argumento, le indica a Chef que el paquete apache2 debe estar instalado, que es la acción por defecto.

Esto hará que cuando se aprovisione nuestra máquina virtual se instalará Apache con la configuración por defecto, por lo que si ejecutamos ahora vagrant up desde la ruta del fichero Vagrantfile comprobaremos que, una vez creada la máquina, el aprovisionador Chef instalará Apache. Una vez que finalice el aprovisionamiento, podremos acceder desde un navegador a la dirección <http://192.168.33.40> y comprobar cómo se muestra la página por defecto de Apache.

A continuación, vamos a sustituir esa página por defecto, para lo cual tendremos que cambiar la configuración del servidor Apache e indicarle qué página utilizar, además de proporcionarle dicha página. Añadimos lo siguiente a nuestra receta:

```
file '/etc/apache2/sites-enabled/000-default.conf' do
  action :delete
end

template '/etc/apache2/sites-available/vagrant.conf' do
  source 'virtual-hosts.conf.erb'
end

link '/etc/apache2/sites-enabled/vagrant.conf' do
  to '/etc/apache2/sites-available/vagrant.conf'
end

cookbook_file "/vagrant/index.html" do
  source 'index.html'
  only_if do
    File.exist?('/etc/apache2/sites-enabled/vagrant.conf')
  end
end
```

El primero de los recursos añadidos es de tipo `file` y nos permite gestionar los ficheros del nodo. En este caso, el argumento `action` nos indica que deseamos borrar el fichero, por lo que, si el fichero especificado en la ruta se encuentra disponible, se eliminará.

El segundo de los recursos es de tipo `template` que nos permite procesar una plantilla del servidor y guardar el fichero generado en la ruta de recurso especificada. El argumento que se utiliza en este caso es `source`, que proporciona la ruta de la plantilla a utilizar. La plantilla tiene la extensión `.erb`, que indica el formato de plantillas de Ruby, y se buscará en el subdirectorio `templates` dentro del `cookbook`.

El tercero de los recursos es de tipo `link` y se utiliza para generar un enlace simbólico en la máquina destino, indicando en el recurso la ruta del enlace a crear, y en el argumento `to` el fichero a enlazar.

El cuarto recurso añadido arriba es de tipo `cookbook_file` y permite copiar un fichero desde el directorio del `cookbook` al nodo, especificando en el recurso la ruta destino y en el argumento `source` el fichero a utilizar, que se buscará partiendo del subdirectorio `files` dentro del `cookbook`. También podemos apreciar que se incluye una sentencia `only_if`, que sirve para condicionar la ejecución del recurso a que se cumpla la condición especificada en la sentencia, que en este caso consiste en preguntar por la existencia del fichero que debería haber creado el paso anterior, por lo que, si todo ha ido bien, la condición será cierta y se ejecutará el recurso. Este tipo de sentencia condicional se llama en Chef `guard` (`guard`), y también contamos con la inversa `not_if`, que no ejecutará el recurso si se cumple la condición.

En este último recurso hemos utilizado la ruta `/vagrant` como destino del fichero `index.html` que hemos copiado. Esta ruta también será necesario utilizarla en el fichero de configuración de Apache para indicarle dónde están los ficheros de la web. Sería bueno poder definir una variable que nos permitiera establecer este valor una única vez y poderlo referenciar desde ambos sitios.

Por tanto, necesitamos definir una variable, a la que Chef llama atributo, para poder especificar el valor que deseamos utilizar como raíz de nuestra página, para lo cual vamos a crear el fichero `attributes/default.rb` en nuestro `cookbook` con el siguiente contenido:

```
default['apache']['document_root'] = "/vagrant"
```

Esto define un atributo de tipo default (su valor se inicializa cada vez que el Chef Client ejecuta la configuración de su nodo) en la colección apache, denominado `document_root` y con el valor “/vagrant”. El recurso `cookbook_file` ahora sería:

```
cookbook_file "#{node['apache']['document_root']}/index.html" do
  source 'index.html'
  only_if do
    File.exist?('/etc/apache2/sites-enabled/vagrant.conf')
  end
end
```

Tal como se muestra, la referencia al atributo se hace con el símbolo de la almohadilla (#) y, entre llaves ({}), el nombre del atributo dentro del objeto nodo.

## Ficheros adicionales

En los recursos anteriores hemos referenciado un par de ficheros que deberían estar contenidos en el `cookbook`; la plantilla que genera el fichero de configuración de Apache y el fichero `index.html` que se utilizará como página por defecto. Estos ficheros deben estar ubicados en las rutas en las que Chef espera encontrarlos, para que la ejecución de la configuración no dé errores y pueda ejecutar todos los pasos. Para generar dichos ficheros, nos situamos en el directorio de nuestro `cookbook` apache y ejecutamos los siguientes comandos:

```
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 <%= node['apache']['document_root']%>

    <Directory <%= @node['apache']['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 mediante la sintaxis de expresiones Ruby al atributo `node[:apache][:document_root]` que habíamos definido en el fichero de atributos, que sigue el formato: `<%= expresión %>`

Esto le indica a Chef que debe sustituir esa referencia por el valor del atributo definido. 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. En este caso, Chef no lo interpretará y lo copiará tal cual al fichero destino, y quien lo interprete será Apache cuando lea el fichero de configuración resultante.

El otro fichero al que hacemos referencia es `index.html`, que se copiará tal cual desde el origen a la ruta destino. El contenido de ejemplo es el siguiente:

```
<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 de ficheros, pero si accedemos a la IP de la máquina virtual desde un navegador, no veremos cambios y se seguirá mostrando la página por defecto de Apache. Esto es debido a que el servicio de Apache no ha recargado la configuración y, por tanto, no se ha enterado de los cambios. Vamos a incluir un servicio recurso detrás de la instalación del paquete apache2 en nuestra receta:

```
service 'apache2' do
  supports :status => true
  action :nothing
end
```

En este recurso indicamos el nombre del servicio, en nuestro caso, 'apache2', y como argumentos estamos proporcionando supports para indicar que el estado del servicio se puede determinar mediante la opción status del sistema operativo, y action con el valor :nothing para especificar que no queremos hacer nada cuando Chef lea el recurso, ya que vamos a incluir notificaciones en los recursos que requieren que la configuración se recargue, y en ese momento es cuando el recurso ejecutará la acción que corresponda. Modificamos los siguientes recursos para añadir la notificación:

```
template '/etc/apache2/sites-available/vagrant.conf' do
  source 'virtual-hosts.conf.erb'
  notifies :restart, resources(:service => "apache2")
end

link '/etc/apache2/sites-enabled/vagrant.conf' do
  to '/etc/apache2/sites-available/vagrant.conf'
  notifies :restart, resources(:service => "apache2")
end
```

Hemos incluido la notificación tanto en el fichero de configuración, como en el enlace que lo apunta, ya que así cualquier cambio en alguno de los dos recursos provocará la recarga de la configuración de Apache. Por defecto, aunque se disparen varias notificaciones en una ejecución, solo se recargará el servicio una vez al final de la ejecución de la configuración (*Chef Client run*), aunque se puede especificar un temporizador distinto a la notificación para que se ejecute inmediatamente (:*immediate*), o incluso antes de la ejecución del recurso que lo incluye (:*before*). Podríamos haber añadido la notificación al recurso que cambia el fichero index.html, pero esto no es necesario, ya que los cambios en este fichero los detectará Apache, y siempre servirá la última versión disponible.

Si has ejecutado `vagrant provision` tras la inclusión de los ficheros de configuración, y dado que estos no han variado, si lo vuelves a ejecutar ahora no se disparará el reinicio del 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 receta principal de nuestro *cookbook* apache, una vez realizadas todas las modificaciones:

```
apt_update 'Update the apt cache daily' do
  frequency 86400
  action :periodic
end

package 'apache2'

service 'apache2' do
  supports :status => true
  action :nothing
end

file '/etc/apache2/sites-enabled/000-default.conf' do
  action :delete
end
```

```

template '/etc/apache2/sites-available/vagrant.conf' do
  source 'virtual-hosts.conf.erb'
  notifies :restart, resources(:service => "apache2")
end

link '/etc/apache2/sites-enabled/vagrant.conf' do
  to '/etc/apache2/sites-available/vagrant.conf'
  notifies :restart, resources(:service => "apache2")
end

cookbook_file "#{node['apache']['document_root']}/index.html" do
  source 'index.html'
  only_if do
    File.exist?('/etc/apache2/sites-enabled/vagrant.conf')
  end
end

```

## Uso de variables de Ohai

Al igual que otras herramientas de gestión de la configuración, Chef cuenta con un mecanismo para recopilar datos de los nodos y hacerlos disponibles a través de atributos, a los que habitualmente se conoce como *facts*. La herramienta de Chef que nos facilita estas variables es Ohai, y vamos ahora a incluir otra pequeña receta que demuestra cómo se utilizan algunos de los atributos que Ohai pone a nuestra disposición. Para ello, creamos en el mismo directorio de recetas el fichero facts.rb con el siguiente contenido:

```

log 'Showing machine Ohai attributes' do
  message "Machine with #{node['memory']['total']} of memory and
#{node['cpu']['total']} processor/s. \
\nPlease           check           access           to
http://#{node[:network][:interfaces][:enp0s8][:addresses].detect{|k,v|
v[:family] == 'inet'}.first}"
end

```

Este recurso log mostrará un mensaje que incluye la cantidad de memoria disponible en la máquina virtual (`node['memory']['total']`) y el número total de procesadores (`node['cpu']['total']`). Para obtener el atributo que almacena la dirección IP —que, como podemos ver, es un atributo cuya ubicación no es nada evidente dentro del objeto—, hemos accedido primero a la máquina virtual mediante `vagrant ssh` y ejecutado el comando `ohai` directamente, que mostrará por la salida estándar el contenido del objeto `nodo` que construye con todos sus atributos. Una vez analizada la salida, hemos construido la expresión Ruby que da acceso al atributo correspondiente cuya familia es de tipo `inet` dentro del objeto. Hemos utilizado esta fórmula porque la IP por defecto de la máquina virtual (disponible directamente en el atributo `'ipaddress'` es privada y no accesible desde el `host`).

Ahora nos faltaría referenciar esta receta para que se ejecute, para lo cual, al final de nuestra receta por defecto, que es la que se ejecuta el referenciar al `cookbook`, incluimos lo siguiente:

```
include_recipe '::facts'
```

Esta sentencia le indica a Chef que queremos incluir el contenido de la receta indicada en este punto de la ejecución, como si incorporásemos todo su contenido en el punto donde se incluye. Para referenciar a una receta se utiliza el nombre de `cookbook` y, si la receta es diferente a la de por defecto, el separador `::` y el nombre de la receta. Como en nuestro caso es el mismo `cookbook`, no hace falta especificar su nombre, pero sí debemos incluir el separador y, a continuación, el nombre de nuestra receta.

Si volvemos a ejecutar `vagrant provision` ahora, comprobaremos que la configuración de nuestra máquina no varía (ya se había realizado previamente), pero ahora la salida muestra la ejecución de la nueva receta, y su mensaje con el valor de los `facts` del sistema.

## 3.6. Chef Supermarket

Chef, como otras herramientas de gestión de la configuración similares, cuenta con un repositorio compartido de *cookbooks* donde cualquier usuario de la comunidad puede publicar sus *cookbooks* o consultar los que hay disponibles, alrededor de 4000, para hacer uso de ellos. Este repositorio es el Chef Supermarket.

Cada *cookbook* publicado consta de información relativa a su descripción, requisitos, atributos, versión, fecha de publicación, dependencias, etc., con lo que tendremos una información muy valiosa para determinar si el *cookbook* en cuestión cumple en todo o en parte la función que andamos buscando.

Un aspecto interesante que incluyen los *cookbooks* publicados en el Chef Supermarket es una medida de la calidad del *cookbook*, calculada a través de una serie de métricas automáticas que comprueban si contiene indicaciones para pruebas, diferentes colaboradores, incorpora guía para contribuciones, plataformas soportadas, pasa el análisis de código estático, etc. Aunque esta medida no puede sustituir una prueba real de las capacidades y uso del *cookbook*, puede proporcionar una idea del grado de calidad con el que ha sido desarrollado.

Cabe señalar que el Chef Supermarket no solo consta de *cookbooks* publicados, sino que además incorpora una sección de herramientas y *plugins*, donde podemos encontrar otra serie de utilidades y elementos adicionales que enriquecen el ecosistema de Chef.

## 3.7. Referencias bibliográficas

Chef Software. (2020). *Chef Documentation*. <https://docs.chef.io/>

Marschall, M. (2015). *Chef Infrastructure Automation Cookbook*, 2nd Edition. Packt Publishing.

Waud, E. (2016). *Mastering Chef Provisioning*. Packt Publishing.

Herramientas de Automatización de Despliegues

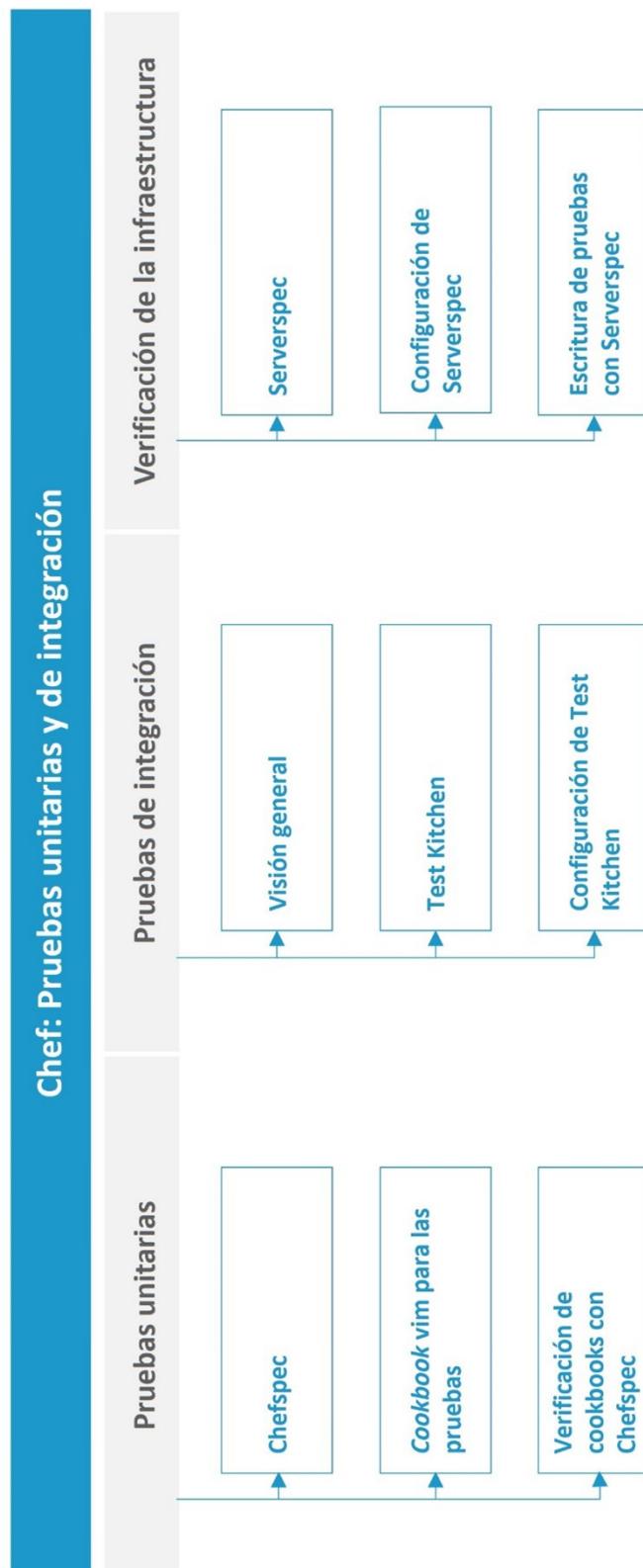
---

# Chef. Pruebas unitarias y de integración

# Índice

<b>Esquema</b>	<b>3</b>
<b>Ideas clave</b>	<b>4</b>
4.1. Introducción y objetivos	4
4.2. Pruebas unitarias de <i>cookbooks</i> con ChefSpec	5
4.3. Pruebas de integración de infraestructura con Test Kitchen	16
4.4. Verificación de la ejecución de Chef con Serverspec	21
4.5. Referencias bibliográficas	29

# Esquema



## 4.1. Introducción y objetivos

Por medio de la utilización de herramientas de gestión de la configuración tales como Chef para gestionar tu infraestructura puedes incorporar las mejores prácticas empleadas en el desarrollo de aplicaciones y aplicarlas así al desarrollo y la mejora de la infraestructura de servidores.

Estas mejores prácticas incluyen, aparte de utilizar un sistema de control de versiones para almacenar y gestionar tus ficheros de definición de la configuración que componen los *cookbooks*, definir una serie de pruebas para verificar que la configuración definida realmente hace lo que se espera, y que futuros cambios en la configuración no producen efectos no deseados en los recursos ya existentes (regresión). Estos conjuntos de pruebas se catalogan en distintos grupos, según su alcance, tales como pruebas unitarias, que se enfocan en verificar la funcionalidad de un elemento individualmente, o pruebas de integración, enfocadas en verificar cómo interactúa un conjunto de elementos relacionados entre sí.

Los objetivos de este tema son los siguientes:

- ▶ Realizar pruebas unitarias mediante ChefSpec.
- ▶ Realizar pruebas de integración con Test Kitchen.
- ▶ Verificar la ejecución con Serverspec.

Aquí puedes ver el vídeo *Mejores prácticas con Chef*:



Accede al vídeo

## 4.2. Pruebas unitarias de *cookbooks* con ChefSpec

El hecho de tratar la infraestructura como código nos abre la posibilidad de implementar pruebas unitarias automatizadas para verificar la funcionalidad de configuración requerida. En este apartado vamos a ver cómo se puede utilizar ChefSpec con Chef para este fin.

Las metodologías de desarrollo ágiles trajeron ideas innovadoras que mejoraron y enriquecieron el mundo del desarrollo de *software*, cambiando la manera de verlo y de hacer las cosas. Esta misma filosofía ágil también es aplicable a la forma en que vemos nuestra infraestructura.

Al gestionar la infraestructura de la misma manera que otras partes importantes del proyecto, nos surge la necesidad de que la configuración de dicha infraestructura sea sometida a pruebas con el fin de verificar y garantizar su fiabilidad. Gracias a herramientas como Chef y similares de gestión de la configuración, estas ideas se pueden implementar de una manera similar a como se implementan en la gestión del propio código fuente de las aplicaciones.

Una de las pautas fundamentales para la creación de software sostenible y de calidad es **poder hacer pruebas** de forma automatizada. En función de este *testing*, podemos realizar cambios y mejoras en el código en el futuro con la garantía de que, si algo de lo ya existente deja de funcionar, las pruebas fallarán y nos avisarán a este respecto.

En general, la manera fundamental y más sencilla para empezar con las pruebas es la **prueba o el test unitario**. Vamos a comenzar repasando los principios básicos de la prueba unitaria con Chef y para ello vamos a utilizar ChefSpec, que es un *framework* de pruebas para Chef construido sobre RSpec, herramienta de pruebas de Ruby.

## Prerrequisitos

Dado que estamos trabajando con el software de gestión de infraestructura Chef, debemos contar con todas las herramientas del ecosistema Chef instaladas para pruebas, mediante la instalación del paquete Chef Workstation o el [kit de desarrollo Chef](#). Esto nos proporciona, aparte de las herramientas esenciales para gestionar el código que ejecuta la infraestructura, las herramientas de pruebas como ChefSpec, que vamos a utilizar para las pruebas unitarias.

Si necesitas más detalles sobre la configuración del entorno de desarrollo Chef, consulta el tema «[Chef: introducción e instalación](#)».

## Escritura de un *cookbook* para las pruebas

Vamos a comenzar por escribir algunas recetas básicas, para poder utilizarlas con ChefSpec y poder así ir viendo más profundamente lo que podemos hacer con esta herramienta. Para ello, vamos a escribir un *cookbook* que va a instalar el editor Vim en Linux, así como algunos paquetes adicionales. En primer lugar, vamos a necesitar generar un *cookbook* básico, ubicado en nuestro directorio de *cookbooks* mediante:

```
chef generate cookbook vim_pruebas_chef
```

Nuestro libro de cocina recién generado debería tener este aspecto:

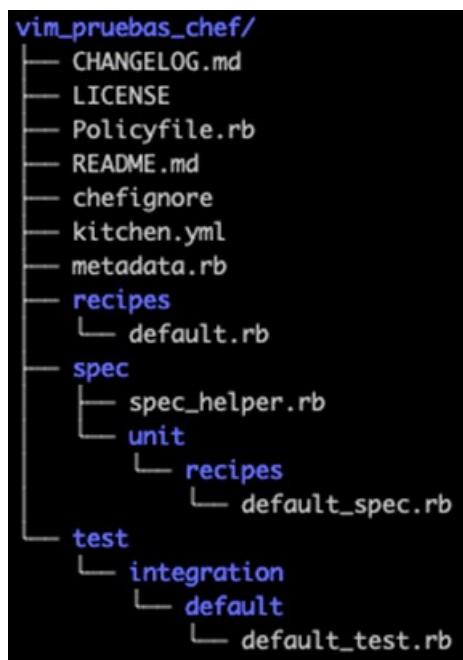


Figura 1. Estructura de directorios y ficheros de un *cookbook*. Fuente: elaboración propia.

A continuación, vamos a escribir algunas recetas que se aseguren de que Vim esté instalado, y usaremos las fuentes o los paquetes oficiales. Incluimos lo siguiente en el fichero *recipes/default.rb* dentro del *cookbook*:

```
begin
  include_recipe      "vim::#{node['vim']['install_method']}"      rescue
Chef::Exceptions::RecipeNotFound
  Chef::Log.warn"A #{node['vim']['install_method']} recipe does not exist
for the platform_family: #{node['platform_family']}"
end
```

En esta receta hemos incluido otra receta en función de un atributo que indica el método de instalación (`node['vim']['install_method']`), captura la excepción si no existe una receta para el método de instalación especificado y muestra un mensaje al respecto.

Ahora, la manera más adecuada de soportar diversas plataformas con diferentes nombres de paquetes desde Chef es mediante la utilización de la función de ayuda `value_for_platform` que, a partir de una estructura de datos, nos devuelve un valor correspondiente a la plataforma sobre la que ejecuta la receta. Así, en función de este valor, podremos determinar los paquetes que deberemos instalar en cada plataforma. Incluimos lo siguiente en el fichero `recipes/package.rb`:

```
vim_base_pkgs = value_for_platform({
    ["ubuntu", "debian", "arch"] => {"default" => ["vim"]},
    ["redhat", "centos", "fedora", "scientific"] => {"default"=> ["vim-minimal", "vim-enhanced"]},
    "default"=> ["vim"]})

vim_base_pkgs.each do |vim_base_pkg|
    package vim_base_pkg
end

node['vim']['extra_packages'].each do |vimpkg|
    package vimpkg
end
```

La instalación a partir del código fuente de `vim` también sería sencilla ya que, aparte de instalar las dependencias básicas necesarias, lo único que se debe hacer en la receta es descargar la versión deseada del código fuente, y ejecutar el comando `make`. Creamos la receta `recipes/source.rb` que realizará las acciones indicadas con el siguiente contenido:

```
cache_path = Chef::Config['file_cache_path']
source_version = node['vim']['source']['version']

apt_update "update_apt_cache"

node['vim']['source']['dependencies'].each do |dependency|
    package dependency do
        action :install
    end
end
```

```

end

remote_file "#{cache_path}/vim-#{source_version}.tar.bz2" do
  source "http://ftp.vim.org/pub/vim/unix/vim-#{source_version}.tar.bz2"
  checksum node['vim']['source']['checksum']
  notifies :run, "bash[install_vim]", :immediately
end

bash "install_vim" do
  cwd cache_path
  code <<-EOH
  mkdir vim-#{source_version}
  tar -jxf vim-#{source_version}.tar.bz2 -C vim-#{source_version} --strip-
  components 1
  (cd      vim-#{source_version}/          &&      ./configure
  #{node['vim']['source']['configuration']} && make && make install)
  EOH
  action :nothing
end

```

Estas serían las recetas necesarias, a las que debemos complementar con un conjunto de atributos predeterminados para cada caso que nos proporcionen los datos requeridos para que el *cookbook* funcione. A continuación, se muestra el contenido de los ficheros de atributos para nuestro caso.

```

#./attributes/default.rb

default['vim']['extra_packages'] = []
default['vim']['install_method'] = 'package'

#./attributes/source.rb

default['vim']['source']['version']      = '8.2'
default['vim']['source']['checksum']      =
'f087f821831b4fece16a0461d574ccd55a8279f64d635510a1e10225966ced3b'
default['vim']['source']['prefix']        = '/usr/local'
default['vim']['source']['configuration'] =    "--without-x  --enable-
pythoninterp --enable-rubyinterp --enable-tclinterp --enable-luainterpr --

```

```

enable-perlinterp      --enable-cscope          --with-features=huge      --
prefix=#{default['vim']['source']['prefix']}
default['vim']['source']['dependencies'] =
  if platform_family? 'rhel', 'fedora'
    %w( ctags
        gcc
        lua-devel
        make
        ncurses-devel
        perl-devel
        perl-ExtUtils-CBuilder
        perl-ExtUtils-Embed
        perl-ExtUtils-ParseXS
        python-devel
        ruby-devel
        tcl-devel
        bzip2
    )
  elsif platform_family?('suse')
    %w( ctags
        gcc
        lua-devel
        make
        ncurses-devel
        perl
        python-devel
        ruby-devel
        tcl-devel
        tar
    )
  else
    %w( exuberant-ctags
        gcc
        libncurses5-dev
        libperl-dev
        lua5.1
        make
        python-dev
    )
  end
end

```

```
    ruby-dev  
    tcl-dev  
    bzip2  
)  
end
```

Pero ¿cómo asegurarnos de que el código de estas recetas junto con sus atributos realmente funciona o si es un código válido?

Por supuesto, podríamos ejecutar el *cookbook* en un servidor existente o en una máquina virtual e inspeccionar los resultados manualmente. Sin embargo, a medida que se empieza a trabajar con más de un par de *cookbooks*, se hace un trabajo pesado y laborioso, haciendo que el proceso de verificación sea lento y doloroso. Para afrontar esta circunstancia, sería más adecuado escribir una serie de pruebas unitarias que puedan verificar que los métodos apropiados de Chef se ejecutan, en lugar de tener que verificar los resultados del Chef *run* manualmente.

En cualquier caso, cabe señalar que las pruebas unitarias no sustituyen a las pruebas de integración, sino que las complementan y son mucho mejor que no tener nada, además de que son más sencillas de elaborar. Estas pruebas unitarias pueden detectar errores y *bugs* en una fase de desarrollo temprana y, por tanto, pueden ahorrarnos un valioso tiempo sin generar además ningún coste de aprovisionamiento.

## Verificación de *cookbooks* con pruebas ChefSpec

La manera de comenzar a escribir pruebas unitarias para un *cookbook* es crear en la carpeta `spec` un archivo de especificaciones correspondiente a cada receta, por lo que vamos a empezar escribiendo las especificaciones para la receta predeterminada. Dado que esta receta solo consta de la inclusión de otra receta para

cada método de instalación, empezar nuestras pruebas con ella es lo más conveniente.

Lo primero que debemos hacer es incluir la gema de ChefSpec y establecer un run en memoria, lo que ejecutará muy rápido al no estar aprovisionando realmente ningún equipo. Esto es fácil de configurar, incluyendo lo siguiente en el fichero `spec/unit/recipes/default_spec.rb` o, si ya existe el archivo (las últimas versiones de Chef incorporan ya alguna prueba unitaria de ejemplo), sustituiríamos el contenido del bloque `describe` por el siguiente:

```
require 'chefspec'

describe 'vim_pruebas_chef::default' do
  platform 'ubuntu'
end
```

El siguiente paso será definir algunos casos de prueba de la funcionalidad que se describe en el interior de la receta y establecer las expectativas de estos casos de prueba.

Lo más básico que podemos probar es que nuestra configuración de ejecución Chef establezca un atributo predeterminado para especificar el método de instalación. Para ello, hay que agregar un nuevo bloque `it` a la receta de pruebas, dentro del bloque `describe`:

```
require 'chefspec'

describe 'vim_pruebas_chef::default' do
  platform 'ubuntu'

  context 'with default attributes' do
    it "should have default install_method 'package'" do
      expect(chef_run.node['vim']['install_method']).to eq('package')
    end
  end
end
```

```
end
```

Vamos ahora a ejecutar los test mediante el siguiente comando Chef, que debería mostrar un resultado similar al que se muestra.

```
$ chef exec rspec  
.  
  
Finished in 3.28 seconds (files took 2.35 seconds to load)  
1 example, 0 failures
```

Tal como podemos apreciar, nuestro caso ha pasado la prueba sin errores, verificando así que el valor por defecto del atributo está configurado. A continuación, vamos a comprobar si la receta que se incluye en la ejecución es la adecuada. Añadimos el siguiente caso de prueba dentro del bloque context del fragmento anterior:

```
it "should include the vim_pruebas_chef::package recipe when  
install_method='package'" do  
  expect(chef_run).to include_recipe('vim_pruebas_chef::package')  
end
```

Y creamos un nuevo bloque context dentro del bloque padre describe para representar el caso de un valor diferente para el atributo `install_method`, cuyo valor sobrescribimos al inicio del bloque:

```
context "with 'source' as install_method" do  
  override_attributes['vim']['install_method'] = 'source'  
  
  it "should include the vim_pruebas_chef::source recipe when  
install_method='source'" do  
    expect(chef_run).to include_recipe('vim_pruebas_chef::source')  
  end  
end
```

Además de las comprobaciones (*matchers*) que ya incluye RSpec por defecto, ChefSpec define *matchers* adicionales para todos los recursos básicos de Chef.

---

Para consultar los RSpec matchers estándar:

<https://www.relishapp.com/rspec/rspec-expectations/docs/built-in-matchers>

---

---

Para consultar los matchers propios de Chef:

<https://www.rubydoc.info/github/sethvargo/chefspec>

---

A continuación, se muestra el fichero `spec/unit/recipes/default_spec.rb` de pruebas unitarias completo que hemos definido en los pasos anteriores con los tres casos de prueba definidos, que deberían pasar con éxito:

```
require 'spec_helper'

describe 'vim_pruebas_chef::default' do
  platform 'ubuntu'

  context 'with default attributes' do
    it "should have default install_method 'package'" do
      expect(chef_run.node['vim']['install_method']).to eq('package')
    end

    it "should include the vim_pruebas_chef::package recipe when
install_method='package'" do
      expect(chef_run).to include_recipe('vim_pruebas_chef::package')
    end
  end

  context "with 'source' as install_method" do
    override_attributes['vim']['install_method'] = 'source'

    it "should include the vim_pruebas_chef::source recipe when
install_method='source'" do
      expect(chef_run).to include_recipe('vim_pruebas_chef::source')
    end
  end

```

```
    end  
end
```

Dado que nuestras pruebas están escritas en lenguaje Ruby al igual que los propios *cookbooks*, podemos también generar dinámicamente los casos de prueba para múltiples plataformas utilizando código Ruby. Vamos entonces ahora a probar nuestra receta package.rb contra diferentes versiones de Ubuntu, Debian y RedHat Linux. Creamos el fichero de pruebas spec/unit/recipes/package\_spec.rb con:

```
require 'spec_helper'  
  
describe 'vim_pruebas_chef::package' do  
  package_checks = {  
    'ubuntu' => {  
      '16.04' => ['vim'],  
      '18.04' => ['vim'],  
      '20.04' => ['vim'],  
    },  
    'debian' => {  
      '8.11' => ['vim'],  
      '9.11' => ['vim'],  
      '10' => ['vim']  
    },  
    'redhat'=> {  
      '7.7'=> ['vim-minimal','vim-enhanced'] ,  
      '8'=> ['vim-minimal','vim-enhanced']  
    }  
  }  
  
  package_checks.each do |platform, versions|  
    versions.each do |version, packages|  
      packages.each do |package_name|  
        it "should install #{package_name} on #{platform} #{version}" do  
          chef_run = ChefSpec::SoloRunner.new(platform: platform, version:  
version, file_cache_path: '/var/chef/cache') do |node|  
            node.normal['vim']['install_method'] = 'package'  
          end.converge(described_recipe)  
          expect(chef_run).to install_package(package_name)
```

```
    end
  end
end
end
end
```

En este ejemplo se itera por la lista de plataformas y versiones, y se generan los casos de prueba dinámicamente en función de ellos, cada uno define las expectativas de los paquetes necesarios que se instalarán.

A continuación, se enumeran las herramientas que hemos utilizado hasta aquí. En el siguiente apartado vamos a avanzar en nuestro plan de pruebas utilizando herramientas adicionales para definir otro tipo de pruebas:

- ▶ [Chef Workstation / ChefDK](#)
- ▶ [RSpec](#)
- ▶ [ChefSpec](#)

## 4.3. Pruebas de integración de infraestructura con Test Kitchen

Para las pruebas de integración de infraestructura, vamos a ver cómo probar automáticamente los *cookbooks* de Chef en una máquina virtual que es una copia del servidor de producción.

Uno de los mayores impedimentos a la hora de gestionar la infraestructura como código es la forma en que la probamos, debido a que la práctica de escribir pruebas automatizadas para la infraestructura es algo que en el pasado no existía y, por consiguiente, el proceso se realizaba de manera manual. Este proceso resultaba ser

repetitivo y propenso a errores, al tener que iniciar sesión en una máquina virtual y ejecutar varios comandos manualmente para verificar el sistema.

El principal inconveniente de este proceso manual es que se requiere contar con una lista concreta de todo lo que debe ser probado y, si a esto añadimos un equipo de personas que está constantemente modificando y evolucionando el sistema, esto se vuelve muy complicado. Por otra parte, aunque se cuente con una batería de pruebas completa y bien definida, si estas son difíciles de ejecutar y no están automatizadas, el equipo terminará habitualmente haciendo caso omiso de ellas y saltándose su ejecución.

Gracias al desarrollo de las herramientas de gestión de la configuración, tales como Ansible, Chef o Puppet, el proceso de las pruebas ha evolucionado considerablemente. El hecho de tener herramientas que facilitan la automatización de las pruebas hace que estas se vuelvan más fáciles de ejecutar, más fiables y, lo que es más importante, mucho más rápidas y sin apenas requerir intervención manual.

En este apartado vamos a cubrir el desarrollo de pruebas de integración para los servidores administrados por Chef, para lo que utilizaremos la herramienta Test Kitchen como ejecutor de máquinas virtuales, respaldado por Vagrant y VirtualBox. La prueba propiamente dicha la escribiremos en Ruby utilizando el *framework* de pruebas Serverspec.

## Requisitos previos

Todas las herramientas que vamos a manejar están incluidas en la instalación de Chef WorkStation o ChefDK (kit de desarrollo Chef), por lo que ya deberías tenerlas instaladas. Si no es así, consulta el tema «Chef: introducción e instalación» para revisar el apartado de instalación.

Las demás herramientas de ayuda que vamos a utilizar, tales como Vagrant y VirtualBox, también deberían estar instaladas si has seguido los ejemplos de los

temas anteriores, por lo que ya deberías estar en disposición de poder desarrollar las pruebas de integración mediante Test Kitchen.

## Visión general de las pruebas de integración

Para el propósito de este apartado vamos a utilizar el *cookbook* básico de instalación de Vim desde el código fuente o desde los paquetes oficiales de distribución, que ya hemos utilizado en el apartado anterior. De cara a las pruebas de integración, lo que más nos interesa son los resultados de la propia ejecución, mientras que la implementación concreta es menos relevante.

Vamos a probar los dos escenarios diferentes posibles, el que incluye la receta para la instalación de Vim desde paquetes y el que lo instalará desde las fuentes.

### Test Kitchen

**Test Kitchen** es una herramienta de automatización de pruebas distribuida con Chef WorkStation y ChefDK que es capaz de gestionar máquinas virtuales internamente llamadas nodos, aplicar la configuración Chef y realizar test. Se puede utilizar de una manera totalmente automatizada, donde todos los pasos se ejecutan secuencialmente, incluyendo la destrucción del nodo al final de la ejecución, aunque también se puede optar por la ejecución manual de tareas.

De cara a ejecutar las pruebas de integración, es necesario utilizar la misma configuración Chef que se va a utilizar en un entorno real, con la misma lista de ejecución, recetas, funciones y atributos. Opcionalmente, puede ser requerido el proporcionar una serie de atributos adicionales personalizados, que se utilizarán solo en el entorno de prueba, como pueden ser datos de prueba, por ejemplo.

Un nodo se puede representar en Test Kitchen con cualquier tipo de virtualización, a través de *plugins* denominados **drivers**. En la mayoría de los casos se suele utilizar

Vagrant, pero hay diferentes alternativas tales como Docker o cualquiera de los muchos proveedores de nube soportados, como Amazon Web Services o DigitalOcean.

---

La lista completa de los drivers soportados por Test Kitchen se puede encontrar en:

<https://kitchen.ci/docs/drivers/>

---

Todos los ajustes de configuración de Test Kitchen se definen dentro del fichero `kitchen.yml` en el directorio raíz del *cookbook*.

A pesar de que en nuestro caso vamos a utilizar Test Kitchen con un solo *cookbook*, se puede utilizar para probar el funcionamiento de la lista completa que esté asociada con el tipo de máquina que estés utilizando.

### Configuración de Test Kitchen

Vamos a probar nuestro `cookbook vim_pruebas_chef` y sus recetas mediante `chef_solo` en las plataformas Ubuntu 18.04 y CentOS 7.8 con el *driver* de Vagrant. El fichero de configuración `kitchen.yml` debe contener lo siguiente:

```
---
driver:
  name: vagrant

provisioner:
  name: chef_zero

platforms:
  - name: ubuntu-18.04
  - name: centos-7.8

suites:
  - name: source
    run_list:
```

```

- recipe[vim_pruebas_chef::default]
  attributes:
    vim:
      install_method: "source"
- name: package
  run_list:
    - recipe[vim_pruebas_chef::default]
  attributes:
    vim:
      install_method: "package"

```

Tal como podemos ver, en el fichero YAML estamos especificando el *driver*, el aprovisionador, las plataformas y las suites de pruebas, que en nuestro caso se corresponde a dos elementos para probar nuestro *cookbook* con los dos posibles valores de `install_method`. Para verificar la configuración y ver un resumen de todas las instancias disponibles proporcionadas por Test Kitchen, basta con ejecutar:

```
$ kitchen list
```

Instance	Driver	Provisioner	Verifier	Transport	Last Action
Last Error					
source-ubuntu-1804	Vagrant	ChefZero	Busser	Ssh	<Not Created>
<None>					
source-centos-78	Vagrant	ChefZero	Busser	Ssh	<Not Created>
<None>					
package-ubuntu-1804	Vagrant	ChefZero	Busser	Ssh	<Not Created>
<None>					
package-centos-78	Vagrant	ChefZero	Busser	Ssh	<Not Created>
<None>					

Como se puede ver, Test Kitchen nos está proporcionando cuatro casos: cada suite contra cada plataforma definida en el archivo de configuración. Estas instancias no se han creado aún, así que vamos a escribir algunos casos de prueba para poder probarlos contra estas instancias en el siguiente apartado.

## 4.4. Verificación de la ejecución de Chef con Serverspec

Ahora que tenemos preparada la infraestructura completa para realizar nuestras pruebas de integración, nos falta establecer nuestras expectativas mediante los casos de prueba y encontrar una manera de verificarlos, para lo cual necesitaremos otra herramienta, dado que **Test Kitchen** es únicamente una herramienta de automatización, y no nos proporciona una manera de escribir los casos de prueba.

**Serverspec** es un *framework* de pruebas basado en RSpec, que es a su vez un *framework* de pruebas muy extendido entre la comunidad Ruby. Permite escribir pruebas de estilo RSpec para comprobar la configuración de la infraestructura, sin importar cómo ha sido aprovisionada dicha infraestructura. Las pruebas se definen de una manera descriptiva, así que con herramientas como esta se reduce al mínimo la posibilidad del error humano.

### Configuración de Serverspec

Para poder comenzar a escribir pruebas con Serverspec, es necesario contar con un subdirectorio llamado `integration` dentro del directorio de pruebas (`test`) de nuestro *cookbook*, donde se van a alojar todas las pruebas de integración. Bajo `integration` debería crearse un subdirectorio por cada suite, y dentro, a su vez, uno por cada *framework* de pruebas que vamos a utilizar, lo que significa que puedes utilizar tantos *frameworks* de pruebas como deseas en la misma suite simultáneamente.

```
test
└ integration
    └─#{SUITE 1}
        └─#{TEST FRAMEWORK}
    └─#{SUITE 2}
        └─#{TEST FRAMEWORK}
```

Dado que en nuestro caso vamos a utilizar Serverspec como el *framework* para las pruebas, a continuación se muestran los comandos necesarios para crear la estructura de directorios correspondiente:

```
mkdir -p test/integration/source/serverspec
mkdir -p test/integration/package/serverspec
```

Lo primero que vamos a necesitar es un *script* de apoyo hecho en Ruby que se encargue de cargar Serverspec y de establecer las opciones de configuración generales que vamos a necesitar, tales como la ruta utilizada para buscar los binarios durante la ejecución de las pruebas. Creamos el fichero `spec_helper.rb` dentro de la ruta `test/integration/package/serverspec/` con el siguiente contenido:

```
require 'serverspec'
require 'pathname'

set :backend, :exec
set :path, '/bin:/usr/local/bin:$PATH'
```

Una vez que contamos con este fichero de configuración, podemos incluirlo a través de la instrucción `require` al principio de cada fichero de pruebas que escribamos, con el objetivo de no tener que especificar esta configuración en cada nuevo fichero de pruebas. Las pruebas se escriben en Ruby estándar, y por esto utilizamos la instrucción `require` que proporciona este lenguaje:

```
require 'spec_helper'
```

## Escritura de pruebas con Serverspec

La prueba más básica que podemos realizar sobre nuestra receta es si nuestro aprovisionador ha instalado el paquete correspondiente que estamos esperando. Si la comprobación la hiciéramos de forma manual, lo más seguro es que solo ejecutásemos el comando `vim` para verificar si el programa existe y se ejecuta con éxito. Con Serverspec, tenemos una manera mucho más adecuada de comprobar el correcto funcionamiento de cualquier recurso específico, como puede ser un comando.

Además del uso de los comandos, con Serverspec también se puede verificar el estado propiamente dicho de los paquetes, lo cual es adecuado utilizar en nuestro caso concreto, al haber instalado en nuestro ejemplo los paquetes proporcionados por la distribución correspondiente. Las ventajas de utilizar Serverspec van mucho más allá, ya que el propio *framework* nos ofrece una gran cantidad de recursos y validaciones adicionales que permiten verificar una gran cantidad de elementos, tales como servicios o puertos.

---

Puedes encontrar la lista completa de los tipos de recursos que proporciona Serverspec con ejemplos en el sitio de la documentación Serverspec:

[https://serverspec.org/resource\\_types.html](https://serverspec.org/resource_types.html)

---

Por tanto, vamos a escribir nuestra primera prueba describiendo el estado del paquete `vim` e indicando que esperamos que se instale, añadiendo lo siguiente al fichero `test/integration/package/serverspec/vim_spec.rb`:

```
require 'spec_helper'

describe package('vim') do
  it {should be_installed}
end
```

De la misma manera podríamos también verificar cualquier otro recurso, como el *output* de un comando, o simplemente su estado de salida. En nuestro ejemplo, una comprobación que puede resultar útil es la versión que se ha instalado de la aplicación Vim. Del mismo modo que haríamos de forma manual, se puede comprobar la salida del comando `vim --version` y verificar si devuelve la versión deseada o incluye una cadena o expresión regular concreta, para lo cual añadimos lo siguiente a nuestro fichero `vim_spec.rb`:

```
describe command('vim --version') do
  its (:stdout) {should match /VIM - Vi IMproved/}
end
```

Dado que ya tenemos un par de casos de prueba definidos, vamos a ejecutarlos para comprobar su correcto funcionamiento, lo cual haremos mediante la ejecución del comando `kitchen test`, indicando como argumento el nombre de la instancia que se quiere probar. Ya habíamos visto cómo mostrar una lista de todas las instancias disponibles, mediante el comando `kitchen list`, por lo que vamos ahora a comprobar la instalación del paquete contra Ubuntu 18.04:

```
$ kitchen test package-ubuntu-1804
```

```
----> Starting Test Kitchen (v2.5.1)
----> Cleaning up any prior instances of <package-ubuntu-1804>
----> Destroying <package-ubuntu-1804>...
      Finished destroying <package-ubuntu-1804> (0m0.00s).
----> Testing <package-ubuntu-1804>
----> Creating <package-ubuntu-1804>...
      Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'bento/ubuntu-18.04'...
```

```
[...]
```

```
----> Running serverspec test suite
----> Installing Serverspec..
```

```
[...]
```

```

Package "vim"
  is expected to be installed

Command "vim --version"
  stdout
    is expected to match /VIM - Vi IMproved/

Finished in 0.15106 seconds (files took 0.27762 seconds to load)
2 examples, 0 failures

Downloading files from <package-ubuntu-1804>
Finished verifying <package-ubuntu-1804> (0m7.75s).
----> Destroying <package-ubuntu-1804>...
==> default: Forcing shutdown of VM...
==> default: Destroying VM and associated drives...
Vagrant instance <package-ubuntu-1804> destroyed.
Finished destroying <package-ubuntu-1804> (0m5.76s).
Finished testing <package-ubuntu-1804> (1m7.30s).
----> Test Kitchen is finished. (1m7.92s)

```

Como podemos ver en la salida de la ejecución, Test Kitchen ha creado una nueva máquina virtual con la versión y sistema operativo especificados, la ha aprovisionado con una lista específica de ejecución, ha ejecutado las pruebas contra ella y finalmente la ha destruido después de la ejecución, instalando en cada paso las herramientas requeridas, si fuera necesario. En nuestro caso, ha ejecutado los dos ejemplos que teníamos definidos sin errores.

Como ya hemos mencionado anteriormente, este es un proceso totalmente automatizado, pero para la fase de desarrollo sería demasiado lento tener que ejecutar este proceso con cada cambio de código, por lo que podemos crear (o utilizar el término *converge* en la terminología Test Kitchen) una instancia de forma manual y solo aplicar pruebas después de que se introduzca un cambio importante, por lo que no hay que esperar para hacer *converge* continuamente cada vez. Cuando las pruebas se pasan con éxito, vamos a destruir las instancias que no necesitamos.

Ahora vamos a probar a ejecutar manualmente el proceso de *converge* que se encarga de crear y aprovisionar la nueva instancia, y añadimos a continuación algunos casos de prueba adicionales.

```
$ kitchen converge package-ubuntu-1804
-----> Starting Test Kitchen (v2.5.1)
-----> Creating <package-ubuntu-1804>...
      Bringing machine 'default' up with 'virtualbox' provider...
      ==> default: Importing base box 'bento/ubuntu-18.04'...
[...]
      Converging 1 resources
      Recipe: vim_pruebas_chef::package
      * apt_package[vim] action install (up to date)

      Running handlers:
      Running handlers complete
      Chef Infra Client finished, 0/1 resources updated in 02 seconds
      Downloading files from <package-ubuntu-1804>
      Finished converging <package-ubuntu-1804> (0m15.65s).
-----> Test Kitchen is finished. (1m3.13s)
```

Ya tenemos nuestra instancia aprovisionada, por lo que vamos ahora a incluir más casos de prueba para poder así incluir el soporte para múltiples plataformas que ya hemos definido en la fase de instalación. Tal como están definidas, las pruebas que tenemos hasta ahora no son aplicables a otras plataformas donde el nombre del paquete no coincide con el nombre que tiene en Ubuntu, como ocurre por ejemplo en CentOS, que el paquete se llama `vim-minimal`. Dado que las pruebas se escriben en Ruby, se pueden utilizar las construcciones gramaticales de este lenguaje para establecer las condiciones en función de la familia de sistema operativo como, por ejemplo, para escribir casos de prueba separados para Ubuntu y CentOS.

Modificamos el fichero `vim_spec.rb` con lo siguiente:

```
require 'spec_helper'
```

```

if os[:family] == 'ubuntu'
  describe package('vim') do
    it {should be_installed}
  end
end

if os[:family] == 'redhat'
  describe package('vim-minimal') do
    it {should be_installed}
  end

  describe package('vim-enhanced') do
    it {should be_installed}
  end
end

describe command('vim --version') do
  its (:stdout) {should match /VIM - Vi IMproved/}
end

```

Debido a que ya tenemos la instancia aprovisionada desde el paso anterior, se pueden aplicar nuevos cambios fácilmente para probarlos y comprobar el resultado de las pruebas.

```
kitchen verify package-ubuntu-1804
```

Las pruebas deberían pasar con éxito y, dado que ya no vamos a incluir ninguna otra mejora, vamos a proceder ahora a destruir el nodo:

```
kitchen destroy package-ubuntu-1804
```

Ya están hechas las pruebas para la instalación del editor desde paquetes, y se puede utilizar una técnica semejante para definir las pruebas de las recetas que compilan la aplicación desde el código fuente. Las pruebas que debemos definir tienen que

comprobar si se ha instalado la versión adecuada y cuál es el usuario que la ha compilado.

Con `vim` esto se puede obtener con la salida del comando `vim --version`, que proporciona ambos datos, por lo que creamos el fichero `vim_spec.rb` en el directorio de la suite de las fuentes `test/integration/source/serverspec/` con lo siguiente:

```
require 'spec_helper'

describe command('vim --version') do
  its(:stdout) {should match /VIM - Vi IMproved/}
  its(:stdout) {should match /Compiled by vagrant@source-/}
end
```

Para comprobar el resultado de las pruebas, las ejecutamos nuevamente de la forma estándar, indicando el tipo de nodo a crear:

```
kitchen test package-ubuntu-1804
```

Llegados a este punto, deberíamos poder ejecutar todas las pruebas contra todas las plataformas definidas inicialmente, mediante el comando `kitchen test`, sin proporcionar ningún parámetro, y Test Kitchen se encargará de generar, probar y destruir todos los nodos definidos en el fichero de configuración `kitchen.yml`.

Si en algún momento necesitas acceder a una instancia aprovisionada por Test Kitchen para, por ejemplo, probar un único comando, o explorar cómo verificar algo manualmente, puedes iniciar sesión en la instancia de convergencia, una vez que esté creada, y ejecutar comandos directamente. Para ello, puedes utilizar los siguientes comandos:

```
kitchen converge package-ubuntu-1804 # Crea la instancia de convergencia
kitchen login package-ubuntu-1804
```

A modo de conclusión de todo lo que hemos visto hasta aquí, se puede decir que Test Kitchen nos posibilita de una manera sencilla la automatización del proceso de definir y probar la infraestructura de servidores heterogéneos. Con esta herramienta, se pueden ejecutar pruebas automatizadas contra la infraestructura de integración cuando ocurra algún cambio en la plataforma de infraestructura, modificación de código realizada por el equipo de desarrollo o incluso en el proceso de integración continua (CI), y se aplicarán así al código de la infraestructura todos los beneficios de la automatización en la integración continua.

## 4.5. Referencias bibliográficas

Chef Software. (2020). *Chef Documentation*. <https://docs.chef.io/>

Marschall, M. (2015). *Chef Infrastructure Automation Cookbook*, Second Edition. Packt Publishing.

Waud, E. (2016). *Mastering Chef Provisioning*. Packt Publishing.

Herramientas de Automatización de Despliegues

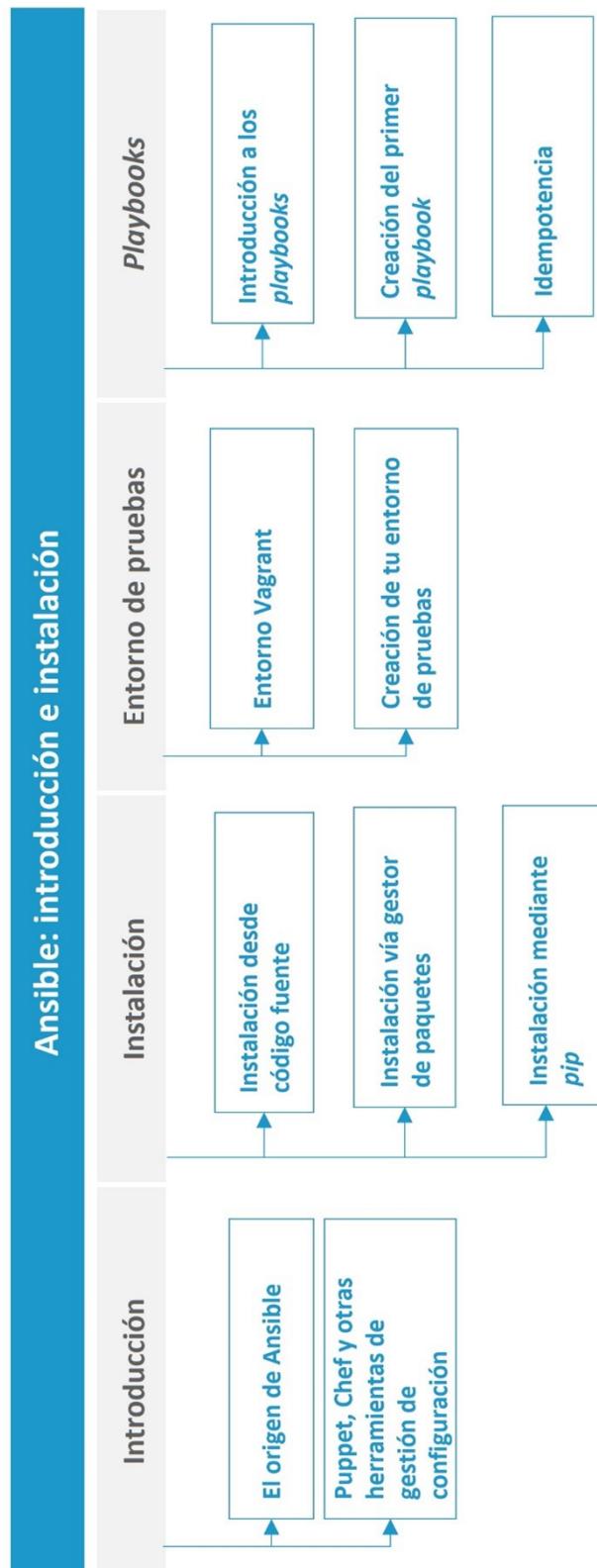
---

# Ansible. Introducción e instalación

# Índice

<b>Esquema</b>	<b>3</b>
<b>Ideas clave</b>	<b>4</b>
5.1. Introducción y objetivos	4
5.2. El origen de Ansible	4
5.3. Instalación de Ansible	8
5.4. Creación de un entorno de pruebas	10
5.5. Introducción a los <i>playbooks</i>	16
5.6. Referencias bibliográficas	26

# Esquema



## 5.1. Introducción y objetivos

Hoy en día, cualquier negocio es un negocio digital. La tecnología es el motor de la innovación, y la entrega de las aplicaciones de una forma más rápida que la competencia ayuda a marcar la diferencia. Históricamente, se necesitaba una gran cantidad de esfuerzo manual y coordinación que complicaba el proceso. Pero, hoy en día, existen herramientas de gestión de la configuración tales como Ansible que son implementadas por miles de empresas a fin de acabar con la complejidad y acelerar sus entornos DevOps.

Los objetivos que se pretenden conseguir en este tema son los siguientes:

- ▶ Conocer la herramienta Ansible y sus principales características.
- ▶ Instalar un entorno Ansible sencillo.
- ▶ Introducir los conceptos básicos de *playbooks*.

## 5.2. El origen de Ansible

La primera versión de Ansible apareció en 2012 como un pequeño proyecto de apoyo de la mano de Michael DeHaan y ha tenido una ascensión meteórica en popularidad, con más de 40 000 estrellas y 5000 contribuidores únicos en GitHub.

Ansible es uno de los proyectos de código abierto con más popularidad que puedes encontrar en GitHub. No solo ha alcanzado un gran éxito, sino que gigantes como la NASA, Spotify o Apple lo han adoptado como herramienta de gestión de la configuración.

Ansible utiliza ficheros en formato **YAML**, que es un lenguaje de representación de datos utilizado habitualmente para configuración, como la fuente principal de información en tiempo de ejecución. Los ficheros YAML son muy simples de escribir y, si nunca has trabajado con ellos, una vez que hagas un par de ellos, ya estarás suficientemente familiarizado. Los ficheros YAML son muy fáciles de leer y usar, al contrario que lo que ocurre con otros formatos o lenguajes como JSON o XML, que no son tan amigables. Sin embargo, incluye algunas particularidades, siendo las principales las de ser sensible a los caracteres de tabulación (no le gustan), a los espacios en blanco y a la sangría de cada línea. Si alguna vez has escrito código fuente en lenguaje de programación Python, todo esto no será ningún problema.

Python es el lenguaje de programación en el que está escrito Ansible. El ejecutable principal y todos los módulos son compatibles con Python 2.7 o Python 3.5, lo que significa que funcionan con cualquier versión de Python2 por encima de la 2.7 o Python 3 por encima de la 3.5. DeHaan eligió Python para Ansible porque no requiere dependencias adicionales en las máquinas que se van a gestionar. En aquella época, la mayoría de las herramientas de gestión de la configuración existentes necesitaban de la instalación de Ruby como prerequisito.

No solo no hay ninguna dependencia o requisito adicional de instalación de otros lenguajes para las máquinas que se van a gestionar, sino que no hay ningún otro requisito adicional. Ansible aprovecha el protocolo SSH para ejecutar sus comandos de forma remota en las máquinas que gestiona, por lo que no requiere la instalación de ningún otro protocolo o sistema de comunicación. Esto supone una gran ventaja, debido a:

- ▶ Las máquinas gestionadas van a ejecutar exclusivamente la aplicación o aplicaciones que le correspondan, sin ningún otro proceso ejecutándose en segundo plano que compita por la CPU y memoria de la máquina.
- ▶ Al hacer uso de SSH, toda su funcionalidad está disponible para aprovecharla. Puedes, por ejemplo, utilizar un *host* como máquina de salto para alcanzar otro

*host*. Además, no existe la necesidad de incluir un mecanismo de autenticación propio; puedes utilizar el que te proporciona SSH.

## Herramientas de gestión de la configuración

La primera herramienta que se considera pionera de la gestión de la configuración fue **CFEngine**. Sin embargo, los que mayor popularidad alcanzaron fueron Puppet y Chef, y por ello todavía hoy en día se comparan habitualmente con Ansible.

Según Heap (2016), Puppet y Chef son herramientas más similares entre sí de lo que lo son con Ansible, aunque todas ellas realizan funciones similares. Tanto Puppet como Chef se basan en el uso de un servidor centralizado para gestionar todo lo relativo al estado de la configuración requerido de los *hosts* y sus metadatos relacionados. Ansible, por el contrario, no necesita utilizar un servidor centralizado al que se conecten los agentes desplegados en las máquinas a gestionar, ya que no necesita el uso de agentes; es, por tanto, *agentless*. Esta es una característica muy importante, ya que al utilizar herramientas como Puppet y Chef, cada agente que ejecuta en una máquina gestionada se conectará periódicamente con el servidor centralizado para comprobar si existen cambios de la configuración, y los aplicarán automáticamente. Ansible, en cambio, delega completamente en el usuario final la labor de propagar los cambios de configuración cuando se requiera.

Ansible es más parecido a **SaltStack** (Salt), otra herramienta que también está escrita en Python y utiliza ficheros YAML para la configuración. Tanto Ansible como Salt están diseñadas fundamentalmente como motores de ejecución, donde la definición de la configuración del sistema no es más que una lista de comandos que ejecutar, que se abstraen mediante módulos reutilizables, los cuales se encargan de proporcionar una interfaz idempotente a los servidores.

Una característica común de todas estas herramientas, tanto Ansible y Salt como Chef y Puppet, es que son declarativas, en lugar de imperativas. La gran mayoría de los elementos de configuración que proporcionan permiten que el usuario defina el

estado de configuración deseado de sus *hosts* y sea la propia herramienta la encargada de alcanzarlo, mediante la aplicación de las acciones que considere necesarias. Para lograr esto es importante el concepto de idempotencia, que permite aplicar tantas veces como queramos una definición de configuración sobre un *host* y su estado resultante será el mismo en todas las ejecuciones. La idempotencia es una característica importante en este tipo de herramientas, y la explicaremos con detalle más adelante.

Como ya hemos mencionado más arriba, Ansible no requiere de ningún agente en las máquinas que gestiona. Esto es lo que se denomina **modelo *agentless***. En este modelo, es necesario enviar los cambios de la configuración a las máquinas cuando así se requiera, a demanda (cuando haya cambios en la configuración, o en las propias máquinas, generalmente). Este modelo es diferente del de Puppet y Chef (con agente), donde se usa un servidor centralizado que almacena la copia maestra de la configuración y al que las máquinas consultan periódicamente para asegurarse de que tienen el estado de su configuración actualizado.

Este modelo tiene ventajas e inconvenientes; la ventaja principal es que, una vez que haces cambios, puedes enviarlos inmediatamente a las máquinas, sin esperar a que un proceso demonio (el agente) compruebe si hay cambios. La desventaja es que tú eres el responsable de distribuir esos cambios a tus máquinas, mientras que con Puppet y Chef basta simplemente con guardar (*commit*) tus cambios en el servidor centralizado, teniendo la certeza de que serán distribuidos pronto. Cabe destacar que Ansible puede configurarse para utilizar este modelo de funcionamiento *pull*, pero no es lo más habitual.

Otra característica de Ansible que ya habíamos mencionado es el uso del protocolo SSH para conectarse remotamente a los *hosts* que gestiona. Esto nos proporciona una gran confianza en el mecanismo de transporte, aunque, por otro lado, puede acabar resultando lento. Por el contrario, Salt utiliza ZeroMQ, que es muy rápido cuando se trata de iniciar una conexión y enviar comandos al destinatario.

## 5.3. Instalación de Ansible

La instalación de Ansible puede ser una tarea sencilla, si utilizamos las versiones que suelen estar disponibles en el repositorio del gestor de paquetes del sistema operativo correspondiente, tales como apt-get en Debian y derivados o yum en Fedora. Sin embargo, la versión disponible en estos repositorios no suele ser la más reciente, por lo que, si deseamos estar a la última, tendremos que optar por otro método de instalación.

Dado que Ansible tiene un desarrollo muy acelerado, puedes querer utilizar la versión más actualizada posible, ya sea construyéndola tú mismo o instalándola mediante un gestor de paquetes.

Para garantizar el uso de la versión más actualizada, puedes optar por descargar directamente de GitHub el código fuente de la herramienta desde el repositorio ubicado en <https://github.com/ansible/ansible> e instalarlo tú mismo.

Puedes necesitar instalar algunas dependencias, si tu sistema no las tiene ya, tales como Git o la versión adecuada de Python y, una vez que las tengas, puedes descargar Ansible directamente desde el repositorio GitHub y construirlo:

```
git clone git://github.com/ansible/ansible.git -recursive  
cd ansible  
make  
sudo make install
```

Con esto construirás e instalarás la última versión disponible de desarrollo de Ansible desde el propio código fuente.

Si estás ejecutando desde una máquina con un sistema operativo basado en Debian o RedHat, puedes preferir utilizar dkpg o yum para instalarlo a través del gestor de

paquetes de tu sistema, dado que proporciona una versión más estable y probada, así como una manera limpia de desinstalar Ansible.

Si estás ejecutando en un sistema que no tiene Ansible disponible a través de su gestor de paquetes, puedes instalarlo a través del Makefile. Para ello, debes ejecutar `sudo make install` como se ha mostrado anteriormente, con lo que lograrás instalar Ansible en tu sistema.

Si tienes una máquina basada en OS X, puedes instalar Ansible a través del gestor de paquetes Homebrew (<https://brew.sh/>). Bastará con ejecutar `brew install ansible` y se instalará la última versión que se haya registrado en el gestor.

Otra manera de instalarlo sería utilizando PPA, para lo que necesitarás primero registrar el repositorio y actualizar la caché del gestor de paquetes. Necesitarás instalar previamente el paquete `python-software-properties` si no tienes ya instalado en tu sistema `apt-add-repository`:

```
sudo apt-get install python-software-properties # if required  
sudo apt-add-repository ppa:ansible/ansible  
sudo apt-get update  
sudo apt-get install ansible
```

También puedes optar por la instalación utilizando pip, el gestor de paquetes de Python, que debes tener instalado previamente en tu sistema. Este método puede ser la alternativa si tu sistema operativo no soporta el uso de PPA:

```
sudo pip install ansible
```

En el siguiente vídeo puedes ver la explicación detallada de la instalación de Ansible:



Accede al vídeo

---

## 5.4. Creación de un entorno de pruebas

Una vez instalado Ansible, puedes comenzar a automatizar tu infraestructura, para lo que necesitarás un entorno de pruebas y un *playbook* de Ansible. Una manera sencilla y rápida de poder desplegar un entorno de pruebas de desarrollo temporal es mediante las herramientas Vagrant y VirtualBox. Con estas herramientas generaremos el entorno donde instalar el conjunto (*stack*) PHP y MySQL mediante nuestro primer *playbook*.

### Creación de un entorno de pruebas Vagrant

Antes de desarrollar el primer *playbook*, vamos a necesitar un entorno donde probarlo. Se puede desarrollar *playbooks* y ejecutarlos directamente en tu máquina local, pero es más recomendable poder probarlos en un entorno aparte, donde no tenga importancia si se han producido errores o que la configuración no funcione tal como se esperaba. Una máquina virtual (VM) de VirtualBox (que proporciona la virtualización) y la herramienta Vagrant (que automatiza la gestión de esas máquinas) proporcionan los entornos de prueba.

La herramienta VirtualBox de Oracle es gratuita y proporciona un motor de virtualización. Nos permite crear una máquina virtual e instalar en ella cualquier sistema operativo de la misma manera que se haría en una máquina física. Se puede usar VirtualBox como *software* independiente, pero es recomendable combinarlo con Vagrant, que es fundamentalmente un lenguaje de *scripting* para máquinas virtuales.

Vagrant te permite gestionar y automatizar todo esto conjuntamente. Podrías tú mismo crear y lanzar una máquina con VirtualBox y luego ejecutar Ansible sobre ella manualmente, pero Vagrant te permite definir todo eso programáticamente y poder así ejecutarlo de manera automatizada. Esta configuración puede luego ser guardada en el repositorio junto con tu código y, cuando cambie, todos los miembros del

equipo tendrán la última versión la próxima vez que se actualicen contra el repositorio. Aunque lo vamos a usar con VMs, Vagrant también se podría utilizar para controlar *hardware* real.

## Crea tu entorno

Una vez que ya tienes las dependencias instaladas, puedes crear la máquina virtual en la que vas a instalar paquetes y configurarlos mediante Ansible. Crea un nuevo directorio con el nombre ansible-test, accede a él y ejecuta el siguiente comando:

```
vagrant init ubuntu/bionic64
```

Esto generará el fichero Vagrantfile en ese directorio. En el siguiente ejemplo creamos un directorio ansible-test donde vamos a crear la VM con Vagrant, y vemos el resultado de esta operación:

```
$ mkdir ansible-test && cd ansible-test  
$ vagrant init ubuntu/bionic64
```

```
A `Vagrantfile` has been placed in this directory. You are now  
ready to `vagrant up` your first virtual environment! Please read  
the comments in the Vagrantfile as well as documentation on  

```

Cuando el fichero Vagrantfile se ha creado, el comando `vagrant up` nos permite «levantar» la máquina virtual, que, si no está ya creada, se creará y arrancará, y en caso de estar parada, simplemente se arrancará. La primera vez que lo ejecutas, este comando realiza una serie de acciones. Primero comprobará si en tu máquina existe una caja (imagen de máquina virtual que maneja Vagrant) con el nombre `ubuntu/bionic64`. Si no existe, la descargará de Atlas, un repositorio centralizado de cajas que está mantenido por Hashicorp (la empresa responsable de Vagrant).

Si todavía no cuentas con la caja en tu *host*, el comando mostrará algo como:

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Box 'ubuntu/bionic64' could not be found. Attempting to find
and install...
default: Box Provider: virtualbox
default: Box Version:>= 0
==> default: Loading metadata for box 'ubuntu/bionic64'
default: URL: https://atlas.hashicorp.com/ubuntu/bionic64
==> default: Adding box 'ubuntu/bionic64' (20200519.1.0) for provider:
virtualbox
default:                                                 Downloading:
https://atlas.hashicorp.com/ubuntu/boxes/bionic64/versions/20200519.1.0/pr
viders/virtualbox.box
==> default: Successfully added box 'ubuntu/bionic64' (20200519.1.0) for
'vertualbox'!
```

Una vez que la caja ya está en tu sistema, bien porque se acabe de descargar o bien porque ya existiera, la ejecución sigue su proceso importando una copia de la caja en cuestión en el directorio de trabajo (`ansible-test`), pudiendo así reutilizar la misma caja desde distintos directorios. Seguidamente, a través de VirtualBox se crea una máquina virtual basada en esta imagen y se arranca. La salida de pantalla será similar a la siguiente:

```
==> default: Importing base box 'ubuntu/bionic64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/bionic64' is up to date...
==> default: Setting the name of the VM: ansible-test
default_1452487432943_ 66014
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
default: Adapter 1: nat
==> default: Forwarding ports...
default: 22 => 2222 (adapter 1)
==> default: Booting VM...
```

```
==> default: Waiting for machine to boot. This may take a few minutes...
default: SSH address: 127.0.0.1:2222
default: SSH username: vagrant
default: SSH auth method: private key
default: Warning: Connection timeout. Retrying...
default:
default: Vagrant insecure key detected. Vagrant will automatically replace
default: this with a newly generated keypair for better security.
default:
default: Inserting generated public key within guest...
default: Removing insecure key from the guest if it's present...
default: Key inserted! Disconnecting and reconnecting using new SSH key...
==> default: Machine booted and ready!
==> default: Checking for host entries
==> default: Mounting shared folders...
default: /vagrant => /Users/ansible/ansible-test
```

La máquina virtual estará ya creada y ejecutándose, lo que puedes comprobar mediante el siguiente comando status de Vagrant:

```
$ vagrant status
Current machine states:
default running (virtualbox)
```

También puedes entrar dentro de la máquina virtual mediante el comando `vagrant ssh`. Esto hará que accedas a la máquina virtual usando una clave SSH que fue generada por Vagrant cuando la VM se estaba creando. Una vez dentro de la máquina virtual, puedes comprobar que ejecutas la máquina correcta mediante el comando `cat /etc/issue`:

```
vagrant@ubuntu-bionic:~$ cat /etc/issue
Ubuntu 18.04.1 LTS \n \l
```

Llegados a este punto, has logrado crear e iniciar una máquina virtual usando Vagrant. Ahora que ya has hecho esto, podrás probar por ti mismo todas las indicaciones que se darán a lo largo de las siguientes secciones. Dado que no

necesitas esta máquina virtual de momento, vamos a eliminarla mediante el comando `vagrant destroy`:

```
$ vagrant destroy
default: Are you sure you want to destroy the 'default' VM? [y/N] y
==> default: Forcing shutdown of VM...
==> default: Destroying VM and associated drives...
==> default: Removing hosts
```

Cuando ejecutas el comando `vagrant destroy` y respondes 'y' al mensaje de confirmación, provocarás que la máquina virtual se elimine, perdiendo a su vez cualquier operación o cambio que hubieras realizado. Si vuelves a ejecutar el comando `vagrant up`, se volverá a copiar la imagen de partida nuevamente, y volverás a estar en la situación inicial.

Vamos ahora a modificar el `Vagranfile` que se generó, ya que por defecto las máquinas que se crean tienen asignados 489 MB de memoria RAM. Esta cantidad de memoria es escasa para nuestro cometido, por lo que vamos a aumentarla a 1024 MB de memoria para trabajar con cierta fluidez, así que edita el `Vagranfile` e incluye el siguiente fragmento antes de la última línea que contiene la palabra `end`:

```
config.vm.provider "virtualbox" do |vb|
  vb.memory = "1024"
end
```

Esto le indica a Vagrant que asigne 1024 MB de memoria al crear la máquina virtual. Para además indicarle a Vagrant que el aprovisionamiento de esta máquina virtual lo realizaremos mediante Ansible, es necesario incluir en el archivo de configuración de Vagrant el siguiente fragmento, también antes de la última línea con `end`:

```
config.vm.provision "ansible" do |ansible|
  ansible.playbook ="provisioning/playbook.yml"
end
```

Esto le dice a Vagrant que utilice Ansible para aprovisionar la máquina virtual, ejecutando el *playbook* que se llama `playbook.yml` dentro de un subdirectorio `provisioning` que se encuentra en el directorio actual (ruta relativa).

Si Ansible no está instalado en tu máquina (por ejemplo, porque utilices Windows como sistema operativo), necesitas utilizar el aprovisionamiento con `ansible_local`, tal como:

```
config.vm.provision "ansible_local" do |ansible|
  ansible.playbook = "provisioning/playbook.yml"
end
```

La diferencia entre estos dos fragmentos de configuración de Vagrant es que uno utiliza `ansible`, mientras que el otro, `ansible_local`. Al utilizar `ansible` estarás ejecutando Ansible desde tu máquina *host* para aplicar el *playbook* sobre la máquina virtual, mientras que con `ansible_local` lo ejecutará desde dentro de la máquina virtual. Vagrant se encargará de la instalación y la configuración de Ansible dentro de la máquina virtual automáticamente para poder realizar el aprovisionamiento.

Si ejecutamos ahora `vagrant up`, el error que se muestra nos estará indicando que el archivo `playbook.yml` al que se hace referencia no existe aún.

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
There are errors in the configuration of this machine. Please fix
the following errors and try again:
ansible provisioner:
* `playbook` does not exist on the host:
/Users/ansible/ansible-test/provisioning/playbook.yml
```

Crearemos a continuación el fichero que vamos a necesitar implementar para Ansible, evitando así el error que muestra Vagrant al provisionar la máquina virtual. Por tanto, debemos crear primero la carpeta `provisioning` y, dentro de esta, un archivo que se llame `playbook.yml`. Una vez que ambos están creados, ya podrás

ejecutar vagrant provision para intentar crear la VM, aunque esta vez mostrará un error distinto debido a que el fichero que acabamos de crear al que hacemos referencia no tiene un formato YAML válido:

```
ERROR! playbooks must be a list of plays
Ansible failed to complete successfully. Any error output should be
visible above. Please fix these errors and try again.
```

Hemos logrado avanzar, en cualquier caso, ya que este error indica que Ansible se está ejecutando y trata de leer el fichero indicado, pero no es un *playbook* con un formato adecuado.

Seguramente verás errores similares en otras ocasiones en el futuro, debido a que Ansible es bastante estricto con respecto a cómo están formateados sus ficheros *playbook*. El significado del error y el cómo arreglarlo lo explicaremos en el apartado siguiente.

## 5.5. Introducción a los *playbooks*

Como ya se ha mencionado anteriormente, Ansible utiliza YAML como formato de ficheros para definir el estado de la configuración. Estos ficheros son los denominados *playbooks*, en los que, a través de una terminología propia, se define el estado deseado mediante un listado de tareas, compuestas por un conjunto de comandos y argumentos, y cualquier otro dato de configuración que pudiera requerirse. Estas tareas se pueden ejecutar síncrona o asíncronamente, dependiendo de la naturaleza de la tarea y lo que el *playbook* requiera. Estas definiciones requieren una sintaxis muy reducida que lo hacen fácilmente legible y entendible, tanto para las máquinas como para las personas.

Los *playbooks* se parecen más a un modelo de sistemas que a un lenguaje de programación o *script*. Salvo unas contadas excepciones (que veremos más

adelante), se trata de definir el estado deseado de un sistema y dejar que Ansible se asegure de que las máquinas estén en ese estado definido.

Un ejemplo de esto sería el desarrollo de un *playbook* que especifique que el programa PHP se debe instalar en el *host* destino. Al ejecutar el *playbook*, Ansible lo instalará por ti si no está ya instalado, usando el gestor de paquetes que le indiques que debe utilizar. Ansible es capaz de detectar si ya está instalado y, en tal caso, no ejecutará nada. En la siguiente sección, vamos a desarrollar este *playbook*.

## Tu primer *playbook*

Ya hemos visto que un *playbook* de Ansible no es más que un archivo YAML con una sintaxis específica. No se trata de un nuevo lenguaje que debes aprender (como en el caso de Puppet) o de código ejecutable (como en el caso de Chef), sino que se trata de formato YAML estándar.

Los archivos YAML comienzan por definir una sección de metadatos (habitualmente conocida como asunto frontal – *front matter*). Dado que no es necesario añadir ningún metadato, no escribiremos nada en esta sección e iniciaremos nuestro *playbook* con tres guiones seguidos en una sola línea (que indican el final de la sección de metadatos, que está vacía en nuestro caso).

Debido a que los *playbooks* son archivos YAML estándar, deben seguir las mismas reglas que cualquier otro archivo YAML. La mayoría de las veces tropezarás con el hecho de que los ficheros YAML son sensibles a los espacios en blanco, por lo que los espacios y tabuladores en tus ficheros realmente significan algo.

Después de esta primera línea de tres guiones que cierra el asunto frontal, lo primero que debes hacer es decirle a Ansible contra qué máquinas debe ejecutarse este *playbook*, especificando el *host* o grupo de *hosts* en donde ejecutar. Por el momento le diremos a Ansible que ejecute en todos los *hosts* que estén disponibles mediante

la línea - hosts: all en nuestro archivo *playbook*. Ahora, el *playbook* (ubicado en provisioning/playbook.yml) contendrá lo siguiente:

```
---  
- hosts: all
```

Con esto, Ansible ya sabe «dónde» ejecutar, y ahora puedes decirle «qué» es lo que quieras que execute. La sección de tareas (tasks) es la que debemos utilizar para ello. Dentro de esta sección, le vamos a indicar a Ansible que haga un ping de las máquinas para asegurarnos de que puede conectarse a ellas:

```
---  
- hosts: all  
  tasks:  
    - ping:
```

A continuación, si la máquina ya estaba creada, puedes ejecutar Ansible mediante vagrant provision, o si no, crea y aprovisiona la máquina todo de una vez con vagrant up. Verás una salida similar a la siguiente:

```
$ vagrant provision  
==> default: Running provisioner: ansible...  
PLAY [all] ****  
  
TASK [setup] ****  
ok: [default]  
  
TASK: [ping] ****  
ok: [default]  
  
PLAY RECAP ****  
default: ok=2 changed=0 unreachable=0 failed=0
```

El bloque que dice TASK: [ping] te permite saber que tu acción se ha ejecutado correctamente. Esto nos indica que Ansible es capaz de conectarse a la máquina, por

lo que podrá gestionarla. Dado que esta ejecución es bastante sencilla, es muy fácil de seguir la salida de la ejecución y entender lo que está pasando; a medida que la complejidad del *playbook* se incrementa, puedes imaginar que la salida se va complicando cada vez más y, por lo tanto, es más difícil de entender. Es por esto muy recomendable utilizar la posibilidad que nos brinda Ansible de añadir un nombre a cada tarea para explicar su propósito. En el siguiente fragmento añadimos el atributo nombre a la tarea:

```
---
- hosts: all
  tasks:
    - name: Make sure that we can connect to the machine
      ping:
```

En esta ocasión, la salida de la ejecución de Ansible va a mostrar el nombre de tarea establecido, en lugar del mensaje genérico: TASK: [ping] que mostraba anteriormente:

```
TASK: [Make sure that we can connect to the machine] ****
ok: [default]
```

Hemos verificado que Ansible es capaz de conectarse al *host*. Podrías haber conectado también con ella usando vagrant ssh directamente, pero una vez ahí te encontrarías en una máquina vacía sin nada configurado todavía. Vamos a añadir una tarea adicional para decirle a Ansible que instale algunos paquetes en tu máquina virtual.

Vamos a instalar el *software* de código abierto PHP de desarrollo. Lo vamos a añadir mediante otra tarea en nuestro fichero *playbook.yml*, que ahora se parecerá al siguiente:

```
---
- hosts: all
  tasks:
```

```
- name: Make sure that we can connect to the machine
  ping:
- name: Install PHP
  apt: name=php state=present update_cache=yes
```

El módulo ping lo habíamos utilizado para verificar la conexión con el *host*. Esta vez, el módulo que estamos utilizando es apt. El módulo que se quiere utilizar se especifica antes de los dos puntos (:), mientras que los atributos se especifican detrás. El módulo ping no requiere de ningún atributo, mientras que el módulo apt tiene varios de ellos, algunos necesarios para que el módulo sepa lo que debe hacer. En nuestro caso, solo vamos a usar tres de ellos: nombre (name) para identificar el paquete, estado (state) para indicar el estado deseado y actualizar caché (update\_cache) para refrescar la caché de apt.

---

Puedes encontrar un listado completo de todos los argumentos que soporta cada módulo en la documentación oficial de Ansible. Para el módulo apt:

[https://docs.ansible.com/ansible/latest/modules/apt\\_module.html](https://docs.ansible.com/ansible/latest/modules/apt_module.html)

---

Al haber utilizado el módulo apt, le estamos indicando a Ansible que queremos que el paquete de nombre php esté instalado. El atributo state admite cualquier valor de una lista de posibles valores, incluyendo latest, present y absent.

El atributo update\_cache indica a Ansible que actualice la caché del gestor de paquetes primero. Si ejecutas nuevamente vagrant provision, Ansible debería intentar instalar el paquete php. Desgraciadamente, esto fallará, y mostrará el siguiente mensaje de error:

```
TASK: [Install PHP] *****
fatal:
[default]: FAILED! => {"changed": false, "cmd": "apt- get update", "msg": "E: Could not open lock file /var/lib/apt/lists/lock - open (13: Permission denied)
[...]
```

La ejecución ha devuelto un error debido a que Ansible está conectando con el usuario vagrant, que es el que se configura por defecto para acceso a la máquina virtual, y este usuario no cuenta con permisos suficientes para instalar paquetes. Aquí es donde la opción `become` (que controla con qué usuario se ejecutan los comandos) es útil. `become` se puede añadir en dos lugares distintos de tu *playbook*: o bien se añade en la tarea que requiere más permisos, o bien puede añadirse a nivel *playbook*, lo que provocará que todo comando del *playbook* se ejecute con permisos de administrador.

Para añadirlo a una acción individual:

```
- name: Install PHP
  apt: name=php5-cli state=present update_cache=yes
  become: true
```

En su lugar, puedes añadirlo a nivel general de todo el *playbook*, dado que varios de los comandos que vas a ejecutar requieren estos permisos. Después de añadirlo, tu *playbook* quedará de la siguiente manera:

```
---
- hosts: all
  become: true
  tasks:
    - name: Make sure that we can connect to the machine
      ping:
    - name: Install PHP
      apt: name=php state=present update_cache=yes
```

Al guardar los cambios y volver a ejecutar `vagrant provision`, la salida de Ansible debería indicar que PHP se ha instalado correctamente:

```
TASK [Install PHP] *****
changed: [default]
```

Ahora puedes añadir más pasos para instalar nginx y MySQL, agregando más tareas con el módulo `apt` e indicando que quieres que estén presentes `nginx` y `mysql-server`, como muestra el siguiente fragmento:

```
---
- hosts: all
  become: true
  tasks:
    - name: Make sure that we can connect to the machine
      ping:
    - name: Install PHP
      apt: name=php state=present update_cache=yes
    - name: Install nginx
      apt: name=nginx state=present
    - name: Install mySQL
      apt: name=mysql-server state=present
```

Lo mismo que ocurrió al ejecutar con el paquete `php`, cuando ejecutes otra vez `vagrant provision`, la salida obtenida será:

```
TASK: [Install nginx] ****
changed: [default]
```

```
TASK: [Install mySQL] ****
changed: [default]
```

Tras estas operaciones, vamos a acceder dentro de la máquina virtual mediante el comando `vagrant ssh` para comprobar que todo se ha instalado correctamente. Luego, con los comandos que se muestran a continuación, podrás comprobar que los programas en cuestión están instalados:

```
vagrant@vagrant-ubuntu-bionic:~$ which php
/usr/bin/php
```

```
vagrant@vagrant-ubuntu-bionic:~$ which nginx
/usr/sbin/nginx
```

```
vagrant@vagrant-ubuntu-bionic:~$ which mysqld  
/usr/sbin/mysqld
```

Llegados hasta este punto, ya tenemos automatizada la instalación de estas herramientas en la máquina virtual mediante Ansible, por lo que, si ahora eliminas la máquina virtual y la vuelves a crear de nuevo, el proceso de aprovisionamiento configurado volverá a instalar automáticamente estos paquetes. Por tanto, puedes ejecutar `vagrant destroy`, y seguidamente `vagrant up`, para comprobarlo.

Ya hemos terminado la labor de crear nuestro primer *playbook*, mediante el que instalamos los paquetes que hemos incluido en la lista de tareas. Ahora lo que queda por hacer es limpiar un poco, es decir, eliminar las tareas que no son requeridas y los duplicados.

Lo primero que vamos a hacer es borrar la tarea de ping. Dado que ya hemos comprobado que Ansible se puede conectar a la máquina, no es necesario realizar la misma tarea cada vez.

Otra mejora que podemos realizar es la de combinar todas las tareas del módulo `apt` en una sola y utilizar como parámetro una lista de elementos. Para entender cómo funciona esto basta decir que vamos a proporcionar una lista de elementos dentro del argumento `name` en vez de un único valor. Ansible entonces llamará a tu tarea con todos los elementos de la lista. Vamos a desglosar la tarea en múltiples líneas para que se vea más claro:

```
---  
- hosts: all  
  become: true  
  tasks:  
    - name: Install required packages  
      apt:  
        state: present  
        update_cache: yes  
        name:
```

- php
- nginx
- mysql-server

Si ejecutas otra vez `vagrant provision`, debería contraer toda la salida para esa tarea dentro de un bloque, lo que reduce significativamente la salida:

```
$ vagrant provision
==> default: Running provisioner: ansible...
default: Running ansible playbook...

PLAY [all] ****
TASK [setup] ****
ok: [default]

TASK [Install required packages] ****
ok: [default]

PLAY RECAP ****
default: ok=2 changed=0 unreachable=0 failed=0
```

Llegados hasta aquí, tienes una máquina con todas las dependencias necesarias instaladas, y no importa cuántas veces ejecutes el *playbook* ahora que el estado resultante de la configuración va a ser siempre el mismo.

## *Playbooks e idempotencia*

Idempotencia se refiere a la posibilidad de hacer algo muchas veces sin que el resultado varíe en cualquiera de las veces. Para Ansible, un *playbook* es idempotente si el resultado de ejecutarlo sucesivas veces no varía el estado de configuración obtenido tras la primera ejecución, con lo que ese estado de la configuración es estable e invariante, independientemente del número de veces que se ejecute el *playbook*.

Considerando el *playbook* del ejemplo anterior, la primera vez que lo ejecutas evalúa su contenido y aplica las transformaciones necesarias para asegurar que PHP, NGinX y MySQL se instalan. Puedes comprobar en la salida de Ansible que indica que el estado ha cambiado:

```
PLAY [all] ****
GATHERING FACTS ****
ok: [default]

TASK: [Install required packages] ****
changed: [default]

PLAY RECAP ****
default: ok=2 changed=1 unreachable=0 failed=0
```

Si ejecutas el *playbook* nuevamente, en lugar de indicar que ha habido cambios, dirá que está ok. Esto se debe a que el módulo `apt` comprobará antes que nada si el paquete que se desea instalar ya estuviera instalado en el sistema. Dado que en este caso los paquetes ya se encuentran instalados, no realizará ningún cambio. Por tanto, se puede decir que el *playbook* es idempotente:

```
PLAY [all] ****
GATHERING FACTS ****
ok: [default]

TASK: [Install required packages] ****
ok: [default]

PLAY RECAP ****
default: ok=2 changed=0 unreachable=0 failed=0
```

La mayoría de los módulos de Ansible que puedes utilizar en tus *playbooks* son idempotentes, pero también hay algunos módulos que, dependiendo de cómo los uses, pueden romper la idempotencia, como son los módulos `command` o `shell`, que

siempre se ejecutarán al procesar el *playbook*, dado que Ansible no puede determinar por sí solo si la acción que va a ejecutar el comando se ha realizado ya anteriormente o no. Por ejemplo, si borras un archivo a través de un comando, la siguiente vez que ejecutes ya no existirá ese archivo, por lo que podrá fallar la ejecución. Para hacer que tus comandos personalizados sean también idempotentes, puedes añadir condiciones a tareas para indicar cuándo ejecutarlas.

## 5.6. Referencias bibliográficas

Heap, M. (2016). *Ansible: from Beginner to Pro*. Apress.

Hochstein, L. y Moser R. (2014). *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. O'Reilly Media.

Herramientas de Automatización de Despliegues

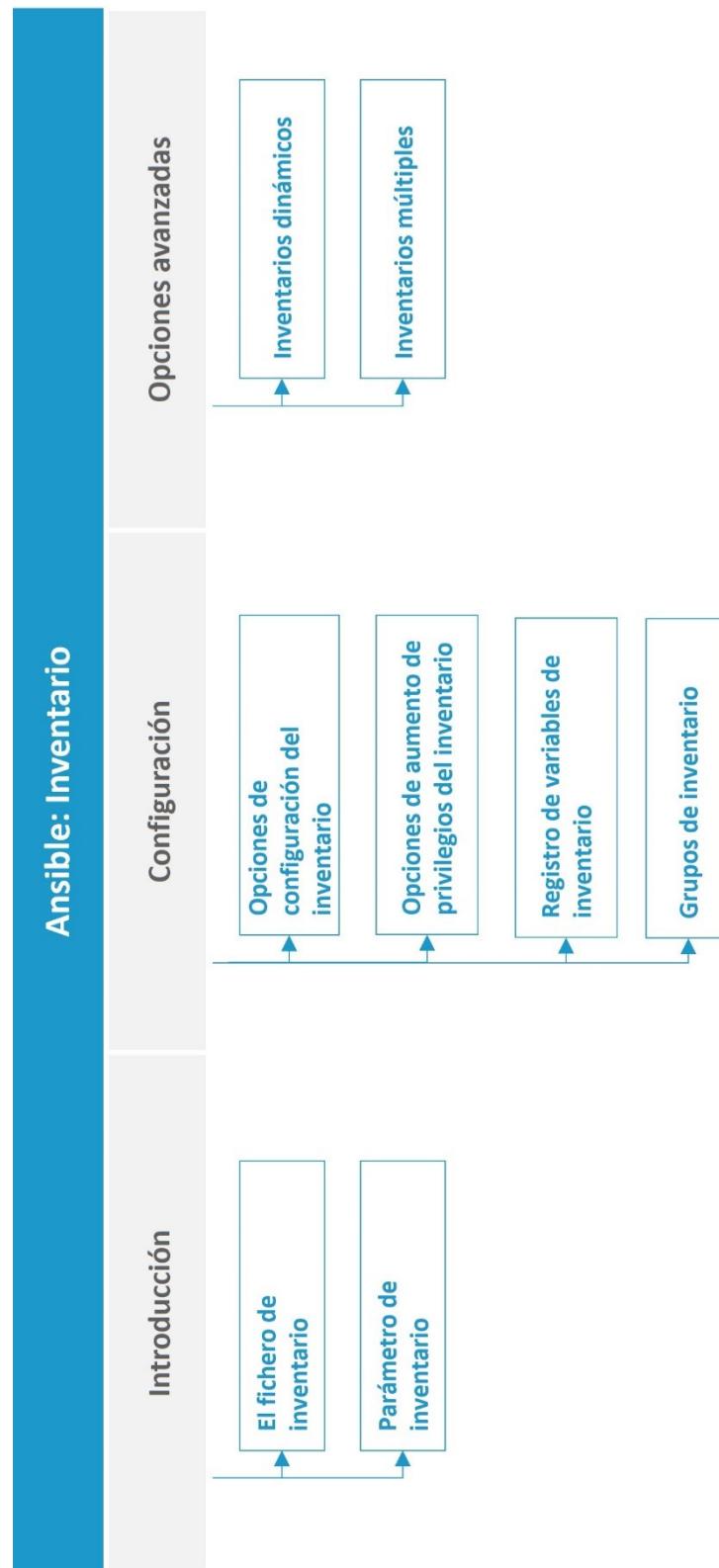
---

# Ansible. Inventario

# Índice

<b>Esquema</b>	<b>3</b>
<b>Ideas clave</b>	<b>4</b>
6.1. Introducción y objetivos	4
6.2. El fichero de inventario	5
6.3. Opciones de configuración del inventario	8
6.4. Registro de variables de inventario	12
6.5. Grupos de inventario	13
6.6. Un ejemplo de inventario	16
6.7. Opciones avanzadas de inventario	18
6.8. Referencias bibliográficas	21

# Esquema



## 6.1. Introducción y objetivos

En la gestión de la configuración, la herramienta que vayas a utilizar necesita saber qué máquinas debe gestionar, sobre las que aplicará la configuración. Esto se conoce como «inventario». Sin conocer el inventario, podrías llegar a tener un conjunto de *playbooks* que definen tu estado de configuración deseado del entorno, pero no sabrías sobre qué máquinas se debería aplicar.

En herramientas como Puppet y Chef, un servidor centralizado se encarga de almacenar esta información. Dado que con Ansible no contamos con un servidor centralizado, necesitamos otro modo de poder proporcionar esta información al código que se ejecuta, para asegurarnos de que se alcanza el estado deseado en las máquinas correspondientes. Aquí es donde entra en juego el fichero de inventario.

Los objetivos que se pretenden conseguir en este tema son los siguientes:

- ▶ Definir el inventario en Ansible.
- ▶ Conocer las opciones de configuración.
- ▶ Conocer las distintas variantes de definición del inventario.

A continuación, puedes ver el vídeo *Ansible vs. Puppet vs. Chef*:



Accede al vídeo

## 6.2. El fichero de inventario

Si no se le indica de otro modo, Ansible leerá el fichero de inventario de la ruta `etc/ansible/hosts`. Sin embargo, el definir tu inventario en este fichero no está recomendado. Lo más conveniente es mantener un fichero de inventario distinto por cada proyecto que tengas y pasarlo como parámetro al comando `ansible` o `ansible-playbook` mediante la opción `-i`. A continuación, se presenta un ejemplo de cómo puedes pasar como parámetro un fichero de inventario personalizado al comando `ansible` para que ejecute sobre el mismo el módulo `ping`:

```
ansible all -i /path/to/inventory -m ping
```

El fichero de inventario en Ansible puede implementarse en un fichero INI, o mediante un JSON. Los ejemplos más habituales en la web utilizan un fichero INI, mientras que el fichero JSON se utiliza típicamente para generar el inventario dinámicamente. Usar el formato INI hace que los ficheros de inventario sean normalmente bastante simples. Pueden ser tan sencillos como una simple lista de nombres de *hosts* en los que ejecutar. A continuación, se muestra un fichero de inventario sencillo:

```
host1.example.com  
host2.example.com  
host3.example.com  
192.168.9.29
```

En este ejemplo se define una lista de cuatro hosts sobre los que se quiere ejecutar. Ansible ejecutará sobre cada uno de ellos por turnos, siguiendo el orden establecido, desde `host1.example.com` hasta `192.168.9.29`. Esta es la forma más sencilla de fichero de inventario que puedes usar, en la que no hay información adicional para ningún *host* ni agrupaciones, sino sencillamente una lista de *hosts* sobre la que queremos ejecutar Ansible.

Si ejecutas el proceso SSHD en un puerto que no sea el estándar, también puedes especificarlo en tu fichero de inventario. No tienes más que añadir dos puntos seguidos del número de puerto detrás de tu nombre de *host*, como por ejemplo:

```
host1.example.com:50822
```

Si trabajas con un elevado número de servidores que siguen un patrón común para el nombre, puedes utilizar rangos en Ansible para identificarlos, dado que permite definir rangos en los nombres, dentro del fichero de inventario. En lugar de especificar cada *host* uno por uno, puedes usar esta funcionalidad de expansión de rangos para definirlos todos de una vez, tal como, por ejemplo:

```
host[1:3].example.com
```

Esta expresión equivale a:

```
host1.example.com  
host2.example.com  
host3.example.com
```

Para algunos casos de uso, una simple lista de *hosts* a los que conectar no es suficiente. Las máquinas con las que conectas también pueden tener diferentes usuarios de sistema habilitados y pueden no utilizar por defecto tu misma clave SSH. Afortunadamente, Ansible también permite especificar las opciones de configuración SSH que se quieran utilizar a la hora de establecer una conexión. En la siguiente sección veremos cómo hacerlo.

## Parámetro de inventario al ejecutar Ansible

Hasta este punto, hemos utilizado siempre `vagrant provision` para aprovisionar con Ansible. Esto es sencillamente un envoltorio que ejecutará el comando que corresponda dependiendo del aprovisionador que utilices. En este caso, se utilizará el comando `ansible-playbook` para aprovisionar mediante Ansible el *playbook* que

se indique a la instancia Vagrant con la que se trabaja. Para otras herramientas como Puppet o Chef, Vagrant se encargará de ejecutar el comando correspondiente para aprovisionar con ellas.

No obstante, no debes depender de Vagrant siempre a la hora de ejecutar Ansible porque estés trabajando en un entorno donde únicamente haya una máquina virtual ejecutando SSH. Para ejecutar en otros entornos, necesitas ser capaz de ejecutar Ansible en todas las máquinas que contengan sin requerir el uso de Vagrant.

Para poder probar la ejecución manual de Ansible, necesitas habilitar en tu `Vagrantfile` la gestión de red privada para poder así acceder por SSH a la máquina virtual. Esto se hace editando el fichero `Vagrantfile` y descomentando o añadiendo la siguiente línea:

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

Una vez guardados los cambios en el fichero, ejecuta `vagrant halt && vagrant up` para reiniciar tu máquina y habilitar esta red. Esto proporcionará a la máquina virtual creada una IP que podrás utilizar para conectar con ella. Una vez que tengas dirección IP, puedes manejárla como si no fuera una máquina gestionada por Vagrant, sino otra máquina accesible en la red en la que necesitas ejecutar Ansible. Esto podría ser una máquina virtual que se encuentra alojada en tu mismo equipo, o en algún lugar de la nube. Para Ansible, es sencillamente una máquina a la que puede conectarse.

El comando `ansible-playbook` permite proporcionar múltiples parámetros, pero solo son necesarios unos pocos para poder ejecutar Ansible. El siguiente ejemplo utiliza el parámetro “`-i`” con el fichero de inventario, para indicar los *hosts* con los que trabajar, y el nombre del *playbook* a ejecutar:

```
ansible-playbook -i <inventory_file> provisioning/playbook.yml
```

El fichero de inventario que se incluye a continuación es un ejemplo que puedes probar para ejecutar Ansible manualmente. Contiene la dirección IP de la máquina virtual a la que debe conectarse, el usuario de acceso y la ruta al archivo de clave privada a utilizar para la conexión mediante SSH. La clave privada es equivalente a una contraseña, y proporciona una firma criptográfica privada que se utiliza para verificar tu identidad, o la de la máquina *host* que se conecta, sin necesidad de introducir ningún valor de forma interactiva:

```
192.168.33.10 ansible_user=vagrant  
ansible_ssh_private_key_file=.vagrant/machines/default/virtualbox/private_key
```

Todo su contenido se encuentra en una única línea. Guardaremos el fichero con el nombre `inventory` y, a continuación, ejecutaremos el siguiente comando, que nos deberá devolver la misma salida que obtuviste al ejecutar `vagrant provision`, dado que equivale al fichero de inventario que genera y utiliza Vagrant al ejecutar Ansible sobre la máquina virtual.

```
ansible-playbook -i inventory provisioning/playbook.yml
```

## 6.3. Opciones de configuración del inventario

Al ejecutar Ansible directamente contra tu máquina virtual Vagrant, has necesitado configurar `ansible_user` y `ansible_ssh_private_key_file`, para así utilizar las credenciales correctas que había generado Vagrant. La siguiente tabla muestra las opciones de configuración más comunes.

El primer conjunto de opciones que se listan en la tabla están relacionadas con la conexión SSH que Ansible utiliza para ejecutar los comandos en los servidores remotos. Para la mayor parte de los servidores solo será necesario configurar estas dos primeras propiedades.

Opciones de configuración del inventario	
Opción de configuración	Explicación
<code>ansible_host</code>	<p>Permite utilizar un nombre diferente a su <code>hostname</code> real para un <code>host</code> en el fichero de inventario y en los <code>playbooks</code>. Esto es útil cuando quieras referirte a una máquina cuya dirección IP es dinámica y puede cambiar. Por ejemplo, en el fichero de inventario:</p> <pre>alpha ansible_host=192.168.33.10</pre> <p>Te permite referirte a la máquina «alpha» en cualquier parte, y Ansible se conectará a la dirección IP 192.168.33.10 cuando necesite acceder a ella.</p>
<code>ansible_user</code>	<p>El usuario SSH con el que acceder a la máquina remota:</p> <pre>ansible_user=Michael</pre> <p>Sería equivalente a:</p> <pre>ssh Michael@host1.example.com</pre>
<code>ansible_port</code>	<p>El puerto en el que tu servidor SSH está a la escucha:</p> <pre>ansible_port=50822</pre> <p>Sería equivalente a:</p> <pre>ssh host1.example.com -p 50822</pre>
<code>ansible_ssh_private_key_file</code>	<p>El fichero de clave SSH utilizado para acceder:</p> <pre>ansible_ssh_private_key_file=/path/to/id_rsa</pre> <p>Sería equivalente a:</p> <pre>ssh -i /path/to/id_rsa</pre>
<code>ansible_ssh_pass</code>	<p>Si el usuario con el que conectas a la máquina requiere una contraseña, se especifica en el fichero de inventario mediante esta propiedad.</p> <p><b>Nota:</b> esto es muy inseguro, y deberías utilizar «autenticación por clave SSH» o utilizar la opción <code>--ask-pass</code> en la línea de comandos para proporcionar la contraseña en tiempo de ejecución.</p>

Tabla 1. Opciones de configuración del inventario. Fuente: elaboración propia.

<code>ansible_ssh_common_args</code>	Parámetros adicionales que proporcionar a la llamada a cualquiera de los comandos SSH, SFTP o SCP. Por ejemplo: <code>ansible_ssh_common_args=' -o ForwardAgent=yes'</code> Es equivalente a: <code>ssh -o ForwardAgent=yes host1.example.com</code>
<code>ansible_ssh_extra_args</code>	Igual que <code>ansible_ssh_common_args</code> , pero los argumentos que especifiques solo se usarán cuando Ansible ejecute SSH.
<code>ansible_sftp_extra_args</code>	Igual que <code>ansible_ssh_common_args</code> , pero los argumentos que especifiques solo se usarán cuando Ansible ejecute SFTP.
<code>ansible_scp_extra_args</code>	Igual que <code>ansible_ssh_common_args</code> , pero los argumentos que especifiques solo se usarán cuando Ansible ejecute SCP.

Tabla 1. Opciones de configuración del inventario. Fuente: elaboración propia.

El ejemplo de fichero de inventario a continuación utiliza alguna de estas opciones:

```
alpha.example.com ansible_user=bob ansible_port=50022
bravo.example.com ansible_user=mary
ansible_ssh_private_key_file=/path/to/mary.key
frontend.example.com ansible_port=50022
yellow.example.com ansible_host=192.168.33.10
```

Con esto se establece un puerto alternativo para las máquinas alpha y frontend, usuarios específicos con los que accede a alpha y bravo, proporciona el fichero de clave privada para bravo y por último define que el nombre yellow.example.com es realmente la dirección IP 192.168.33.10. Esta información adicional es excesiva para un fichero de inventario tan pequeño, pero sirva como ejemplo ilustrativo.

Esto no acaba aquí y hay incluso más opciones disponibles. La siguiente tabla muestra opciones de aumento de privilegios que puedes utilizar en tus ficheros de inventario.

Opciones de aumento de privilegios del fichero de inventario	
Opción de configuración	Explicación
ansible_become_method	Indica el método que se utilizará para obtener privilegios de superusuario. Por defecto se usa sudo, pero puede ser cualquiera de los siguientes métodos: sudo, su, pbrun, pfexec o doas. Algunas como pbrun son herramientas comerciales de seguridad, que no se usarán salvo en casos muy concretos. sudo es la opción más adecuada para la mayoría de los casos.
ansible_become_user	Por defecto, become te elevará a nivel root. Si dispones de otro usuario con los permisos adecuados para realizar la tarea a ejecutar, puedes especificarlo con esta opción de configuración para usarlo en lugar de root, Esto es equivalente a ejecutar sudo pasándole un usuario como parámetro: sudo -u myuser.

Tabla 2. Opciones de aumento de privilegios del fichero de inventario. Fuente: elaboración propia.

Estas opciones relacionadas con el aumento de privilegios pueden definirse en el fichero de inventario, pero no se aplicarán salvo que establezcas become: true en tus *playbooks*.

Dado el siguiente fichero de inventario, alpha y bravo, ambos usarán el usuario automation cuando se establezca become: true en el *playbook*. frontend usará el usuario ansible y yellow usará root, que es el usuario por defecto:

```
alpha.example.com ansible_become_user=automation
bravo.example.com ansible_become_user=automation
frontend.example.com ansible_become_user=ansible
yellow.example.com
```

Cabe destacar que este usuario no es el que se utiliza para acceder a la máquina (se debe definir el argumento ansible\_user para eso). Este usuario es con el que se ejecutará al usar become: true en tu *playbook* o tarea. Será tu responsabilidad asegurarte de que el usuario al que cambies tenga los permisos suficientes para ejecutar la tarea que corresponda.

## 6.4. Registro de variables de inventario

Aparte de poder establecer las variables especiales de Ansible en el inventario, tal como `ansible_user` o `ansible_become_user`, puedes también definir cualquier variable que quieras usar a continuación en un *playbook* o plantilla. Sin embargo, definir variables en el fichero de inventario no es la solución más adecuada generalmente. Hay muchos otros lugares donde definir variables en un *playbook* y la mayoría serán más adecuados que este.

Si estás pensando añadir una variable a un fichero de inventario, analiza si este dato debería estar en un inventario realmente. ¿Se trata de un valor por defecto? ¿Es algo relacionado con un tipo específico de máquinas o con una aplicación específica? Más adelante veremos otros sitios más recomendables donde puedes definir estas variables.

No obstante, si acabas decidiendo que el fichero de inventario es el sitio adecuado donde definir tu variable, es muy sencillo definirla. Por ejemplo, si quieres una variable que se llame `vhost` accesible desde tu *playbook*, puedes definir un `host` como se muestra a continuación:

```
host1.example.com vhost=staging.example.com
```

Esto será útil en ciertas ocasiones, como por ejemplo cuando ejecutas tus sitios web de *staging* y producción desde la misma máquina y necesitas mediante el fichero de inventario diferenciar con cuál de los entornos estás trabajando. Si haces cambios de base de datos en tu *playbook*, pero no quieres impactar el despliegue en producción cuando pruebas en *staging*, puedes usar los siguientes ficheros de inventario para especificar la base de datos que quieras utilizar en cada caso:

```
$ cat staging-inventory
alpha.example.com database_name=staging_db
```

```
$ cat production-inventory  
alpha.example.com database_name=prod
```

Como podemos ver, la variable `database_name` estará disponible desde tu *playbook* con el valor correspondiente en cada entorno, de manera que puedas ejecutar lo que necesites en la base de datos oportuna. Lo único que debes hacer es asegurarte de proporcionar el fichero de inventario adecuado al ejecutar Ansible.

Por ejemplo, para aplicar el *playbook* en el entorno de producción, usaremos:

```
ansible-playbook -i production-inventory playbook.yml
```

## 6.5. Grupos de inventario

Hasta aquí hemos estado utilizando una lista simple de máquinas sobre las que ejecutar Ansible. No obstante, esto no se corresponde con la realidad, donde solemos tener平衡adores de carga, servidores web, servidores de aplicaciones, bases de datos, etc. Se hace necesario poder agrupar estos conjuntos de servidores por tipo y poder referenciarlos como a un único grupo. Ansible permite definir grupos de inventario para dar soporte a este caso de uso.

Para definir un grupo de servidores en el fichero de inventario se utilizan las cabeceras de sección INI, incluyendo su nombre entre corchetes, tal como especifica el formato de archivos INI:

```
[web]  
host1.example.com  
host2.example.com
```

```
[database]  
db.example.com
```

En este fragmento de inventario, definimos dos *hosts* agrupados como servidores web, y otro en el grupo database. En el formato INI los corchetes sirven como marcadores de sección, y lo que sería el nombre de la sección (lo que se incluya dentro de los corchetes) será el nombre del grupo.

Cuando ejecutamos Ansible podemos especificar en qué grupos de *hosts* se deben ejecutar nuestros comandos. Hasta ahora, hemos utilizado el grupo especial `all` para indicar que se ejecute en todos los *hosts* listados. Ahora podríamos especificar `web` o `database` para que Ansible ejecute solo en el grupo de servidores indicado. El siguiente comando ejecutará el módulo ping únicamente en el grupo `web`:

```
ansible web -i /path/to/inventory -m ping
```

También puedes establecerlo en un *playbook* cambiando simplemente el valor de `hosts`: al principio de tu *playbook*, tal como:

```
- hosts: web
tasks:
- ping:
```

De la misma manera que puedes establecer variables para máquinas específicas, también puedes establecer variables para los grupos. Para ello, utiliza una cabecera especial con el nombre de grupo y el sufijo `:vars` en tu fichero de inventario:

```
[web:vars]
apache_version=2.4
engage_flibbit=true
```

Las variables así definidas estarán disponibles en la ejecución Ansible para cualquier máquina del grupo `web`. ¡También puedes incluso crear grupos de grupos!

Imaginemos que tienes un grupo de máquinas en producción que son una mezcla de servidores CentOS 6 y CentOS 7. El fichero de inventario podría parecerse al siguiente:

```
[web_centos6]
host1.example.com
host2.example.com

[web_centos7]
shinynewthing.example.com

[database_centos]
database.example.com

[reporting_centos7]
reporting.example.com
```

Si necesitas ejecutar algo únicamente en las máquinas CentOS 6, típicamente tendrías que hacerlo sobre ambos grupos, webcentos6 y database\_centos6.

En lugar de ello, puedes crear un grupo de grupos utilizando el sufijo :children en tu nombre de grupo:

```
[centos6:children]
web_centos6
database_centos6

[centos7:children]
web_centos7
reporting_centos7
```

Con este grupo de grupos ya solo tendrás que apuntar a los *hosts* centos6 si solo necesitas ejecutar en los servidores CentOS 6. Es más, puedes también definir variables para este nuevo grupo, como harías con cualquier otro grupo:

```
[centos6:vars]
apache_version=2.2
```

```
[centos7:vars]
apache_version=2.4
```

Los grupos son una herramienta muy potente y pueden ser bastante útiles en el futuro cuando definamos variables fuera del fichero de inventario.

## 6.6. Un ejemplo de inventario

A continuación, se muestra un ejemplo de un despliegue ficticio que contiene un servidor web, una base de datos y un balanceador de carga. El entorno se ha ido creando a lo largo de un tiempo, por lo que existen varias versiones de sistemas operativos disponibles, con diferentes usuarios y métodos de acceso a los *hosts*:

```
[web_centos6]
fe1.example.com ansible_user=michael
ansible_ssh_private_key_file=michael.key
fe2.example.com ansible_user=michael
ansible_ssh_private_key_file=michael.key

[web_centos7]
web[1:3].example.com ansible_user=automation ansible_port=50022
ansible_ssh_private_key_file=/path/to/auto.key

[database_centos7]
db.example.com ansible_user=michael
ansible_ssh_private_key_file=/path/to/db.key

[loadbalancer_centos7]
lb.example.com ansible_user=automation ansible_port=50022
ansible_ssh_private_key_file=/path/to/lb.key

[web:children]
web_centos6
web_centos7
```

```
[database:children]
database_centos7

[loadbalancer:children]
loadbalancer_centos7
```

Las máquinas más antiguas funcionan con CentOS 6 y usan la cuenta personal de Michael como inicio de sesión de Ansible. Las máquinas más modernas tienen usuarios de automatización propios, con claves privadas definidas. El *host* de base de datos utiliza una cuenta personal y tiene una clave SSH diferente. Por último, el balanceador de carga dispone de un usuario de automatización, pero en vez de utilizar su clave de automatización, tiene su clave propia del balanceador de carga.

Finalmente, se definen grupos de grupos para hacer posible apuntar a todos los servidores web o a todos los servidores de base de datos como a un grupo, independientemente de la versión de CentOS. Aunque el grupo database solo tiene un grupo hijo por ahora, puedes querer agregar *hosts* con CentOS 7 en un futuro. Desde luego, tener un grupo preparado para usarse cuando sea necesario puede ahorrarnos mucho tiempo.

Observando este fichero de inventario, puede verse que hay siete *hosts* en este despliegue (fe1, fe2, web1, web2, web3, db y lb), qué usuario utilizará Ansible para acceder y qué clave usará para ello. Incluso se puede saber el puerto en el que está ejecutando el demonio SSH.

Un fichero de inventario bien escrito no debe ser simplemente algo que Ansible debe usar, sino que debe servir también como documentación propia del entorno, para cuando necesites referirte a ella, o mostrárselo a alguien.

## 6.7. Opciones avanzadas de inventario

### Inventarios dinámicos

Cuando únicamente tienes uno o dos servidores que administrar, es fácil mantener un fichero de inventario manualmente y no supone demasiado trabajo. Pero a medida que el número de servidores aumenta, este trabajo se complica cada vez más, aparte de que mantener una lista numerosa de servidores manualmente puede convertirse en una tarea propensa a errores.

Cuando se gestiona un número elevado de servidores, es poco probable que el control del inventario dependa de un archivo de texto estático; es probable que se opte en su lugar por una hoja de cálculo o una base de datos. ¿No sería fantástico poder usar estas mismas fuentes en Ansible como la fuente de datos de inventario?

Ansible dispone del concepto de **inventario dinámico**, que consiste en un fichero JSON que proporciona todos los datos necesarios sobre sus *hosts*. El formato de archivo JSON no es tan legible a simple vista como el formato INI, ya que está diseñado principalmente para ser leído por máquinas. Ansible hace su propio procesamiento de estos formatos y realiza algunas comprobaciones al acceder al fichero de inventario que se proporciona.

Cuando se ejecuta Ansible, este comprobará si el fichero pasado como parámetro del inventario es un archivo ejecutable. En caso de serlo, lo ejecutará y Ansible utilizará su analizador JSON para leer los datos resultantes. Si no es ejecutable, Ansible lo leerá suponiendo que está en formato de fichero INI y fallará en el análisis si es un archivo JSON estático.

El uso de un archivo ejecutable permite leer datos de inventario sobre sus *hosts* desde cualquier fuente de datos accesible local o remotamente, tales como un API remoto, una base de datos local, un grupo de ficheros que se analizan y comparan...

a fin de cuentas, lo que sea que se necesite para obtener la lista de servidores sobre la que ejecutar.

Ansible espera que el ejecutable devuelva un formato JSON específico que se utilizará como fuente del inventario de máquinas. Veamos un ejemplo a continuación:

```
{"my_script": ["dev2", "dev"], "_meta": {"hostvars": {"dev2": {"ansible_host": "dev2.example.com", "ansible_user": "ansible"}, "dev": {"ansible_host": "dev.example.com", "ansible_port": "50022", "ansible_user": "automation"}}}}
```

Como podemos observar, se trata de un conjunto de pares clave-valor, aunque los conjuntos pueden a su vez ser subconjuntos. La primera entrada tiene como clave el nombre del *script*, y como valor, la lista de nombres de *host* que se usará como inventario. A continuación, hay una sección de metadatos en la que para cada nombre de *host* hay algunas entradas que contienen el nombre de *host* a utilizar, el usuario SSH con el que acceder y el puerto SSH.

Si por ejemplo tuviéramos una base de datos que contiene todas las máquinas y quisieramos utilizar esa base de datos como fuente del inventario, el ejecutable podría implementar algo parecido al siguiente pseudocódigo:

```
machines = fetch_rows("SELECT hostname, user, key, port FROM active_machines")
hostnames = machines.map (m) => return m.hostname metadata = {
  'hostvars'=> {}
}
foreach (machines as m) {
  metadata.hostvars[m.hostname] = {
    ansible_user => m.user,
    ansible_port => m.port,
    ansible_ssh_private_key_file => m.key
  }
}
output_json({ 'my_script'=> hostnames, '_meta'=> metadata})
```

La función `fetch_rows` se encarga de leer la información de la base de datos, mientras que el resto del script formatea la información tal como Ansible la espera. Aunque este ejemplo solo está leyendo los datos desde un único lugar, podría tener que leer desde múltiples fuentes si fuera necesario para combinar todos los datos en el ejecutable con el que estás trabajando y devolverlos en el formato que espera Ansible.

Existen un gran número de soluciones de inventario dinámico ya construidas disponibles para Ansible.

Para `hosts` ubicados en Amazon AWS puedes optar por utilizar un inventario dinámico, debido a las capacidades de autoescalado de la nube. Aunque un inventario dinámico puede ser de naturaleza estática (por ejemplo, si lo mantienes en una base de datos de forma manual), es muy útil cuando estás utilizando entornos realmente dinámicos, como por ejemplo Amazon EC2, para crear máquinas bajo demanda. En este caso, debes asegurarte al crear el `host` de que tiene las etiquetas adecuadas para que, cuando ejecutes Ansible la próxima vez, se aprovisione junto con todas las demás máquinas de tu entorno.

Si no utilizas OpenStack en lugar de AWS, también existe ya implementado un script de inventario dinámico para OpenStack. Antes de implementar tu propio script, busca primero uno que te pueda servir y aprovechalo. Si no encuentras ninguno que sea adecuado a tu caso, entonces puedes escribir el tuyo propio, aunque siempre puedes aprovechar y reutilizar partes de algunos donde se resuelva alguna situación semejante a la tuya.

## Inventarios múltiples

Por último, ¿qué ocurre cuando se tiene una combinación de máquinas físicas y servidores en la nube en tu entorno? No es posible obtener la fuente desde una API basada en la nube exclusivamente, ya que no sabe acerca de sus máquinas físicas, y,

posiblemente, no se pueda realizar un seguimiento de todos los servidores en la nube manualmente. Afortunadamente, para estos casos Ansible también tiene una solución.

Si la ruta del fichero inventario que pasas como parámetro a Ansible es un directorio, Ansible leerá todos y cada uno de los archivos en ese directorio como un inventario y los fusionará. Esto te permite tener un fichero de inventario de gestión manual, así como otra parte en `ec2.py`, por ejemplo, que genera dinámicamente el inventario de Amazon EC2. Si el inventario es grande, incluso podríamos tener múltiples archivos INI desglosados por centro de datos o por rol (o cualquier división arbitraria que se pueda tener), así como varios ejecutables que se ejecutan para hablar con varios proveedores de nube. Cuando los datos de todos estos ficheros sean recogidos por Ansible, se tratarán como si fueran un único inventario extenso.

## 6.8. Referencias bibliográficas

Heap, M. (2016). *Ansible: from Beginner to Pro*. Apress.

Hochstein, L. y Moser R. (2014). *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. O'Reilly Media.

Herramientas de Automatización de Despliegues

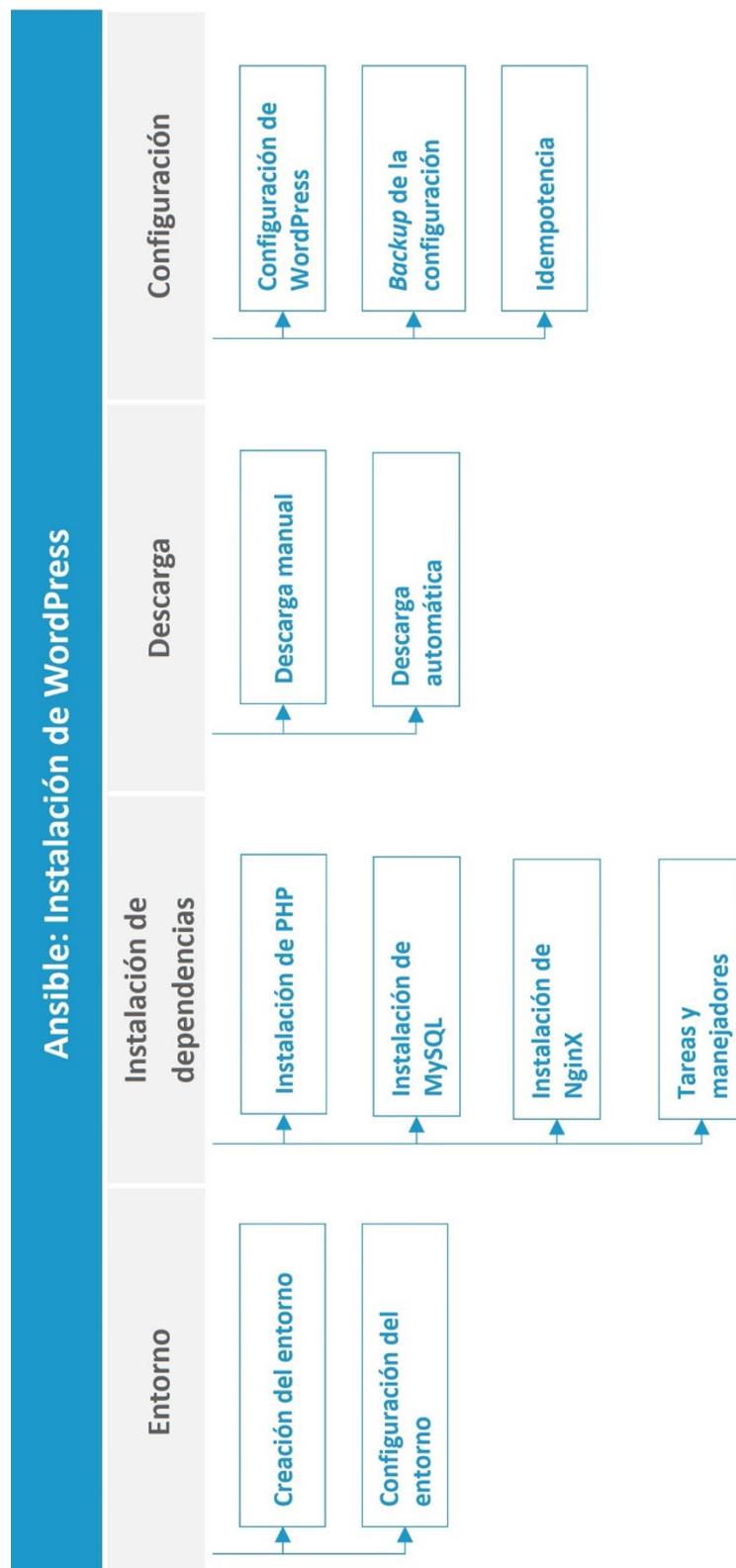
---

# Ansible. Instalación de WordPress

# Índice

<b>Esquema</b>	<b>3</b>
<b>Ideas clave</b>	<b>4</b>
7.1. Introducción y objetivos	4
7.2. Configuración del entorno	4
7.3. Instalación de dependencias	5
7.4. Tareas y manejadores	17
7.5. Descarga de WordPress	19
7.6. <i>Backup</i> de la configuración	26
7.7. Idempotencia	28
7.8. Referencias bibliográficas	29

# Esquema



## 7.1. Introducción y objetivos

Ahora que estás familiarizado con la creación de un entorno en el que desarrollar tus playbooks Ansible, vamos a preparar un playbook que descarga y configura WordPress, una popular aplicación de blogs de código abierto.

WordPress es una herramienta muy popular de código abierto desarrollada en lenguaje PHP que utiliza una base de datos MySQL para el almacenamiento de datos. Para instalar y desplegar WordPress, es necesario contar con PHP instalado (en su versión 5.2 o superior), un servidor web y un gestor de MySQL ya instalado.

Los objetivos que se pretenden conseguir en este tema son los siguientes:

- ▶ Instalar todas las dependencias necesarias de Ansible.
- ▶ Descargar todos los archivos de código fuente de WordPress.
- ▶ Instalar de manera automática una nueva instancia.

## 7.2. Configuración del entorno

Lo primero que vamos a necesitar es un entorno en el que construir y probar este *playbook*. Como ya hemos hecho anteriormente, vamos a usar Vagrant para ello. Crearemos una nueva máquina Vagrant para comenzar con una pizarra limpia. Ejecuta los comandos indicados a continuación desde un terminal:

```
mkdir ansible-wordpress && cd ansible-wordpress  
vagrant init ubuntu/bionic64
```

Como también hemos visto ya en el tema «Ansible: inventario», necesitaremos habilitar la red, aunque esta vez usaremos una dirección IP diferente, para permitirte ejecutar simultáneamente, si quisieras, el entorno (VM) que configuramos anteriormente y este nuevo entorno. Asimismo, aparte de la configuración de red, también necesitaremos ampliar la memoria RAM asignada a la máquina virtual, ya que MySQL Server no se iniciará con los 480 MB que Vagrant asigna por defecto. Tras realizar estos cambios, el archivo `Vagrantfile` contendrá lo que sigue:

```
Vagrant.configure(2) do |config|
    config.vm.box = "ubuntu/bionic64"
    config.vm.network "private_network", ip: "192.168.33.20"
    config.vm.provider "virtualbox" do |vb|
        vb.memory = "1024"
    end
    config.vm.provision "ansible_local" do |ansible|
        ansible.playbook ="provisioning/playbook.yml"
    end
end
```

Cuando se ejecute `vagrant up`, este archivo creará una máquina virtual con una dirección IP 192.168.33.20 y con 1 GB de memoria asignada, lo cual es suficiente para ejecutar WordPress.

## 7.3. Instalación de dependencias

Para ejecutar WordPress, necesitamos como prerequisito instalar tres piezas de *software*: PHP, nginx y MySQL. Como ya hicimos en el tema introductorio, vamos a comenzar creando un *playbook* sencillo que muestre que Ansible puede conectar con la máquina Vagrant:

```
mkdir provisioning
vi provisioning/playbook.yml
```

En el archivo provisioning/playbook.yml, especificamos en qué *host* o grupo de *hosts* se ejecutará el *playbook*, así como el conjunto de tareas que ejecutar. Comenzaremos con un *playbook* básico, que comprueba que puede conectarse con el entorno en el que se está probando, tal como el siguiente:

```
---
- hosts: all
  become: true
  tasks:
    - name: Make sure we can connect
      ping:
```

Una vez creado, ejecutamos vagrant up para crear la máquina virtual y aprovisionarla, y asegurarnos de que la ejecución del *playbook* muestra el resultado esperado indicando que puede conectar con la máquina recién creada.

## Instalación de PHP

WordPress necesita PHP con una versión 5.2 o superior para ejecutarse, pero es recomendable usar la última versión que se encuentre disponible siempre que sea posible. Si deseas instalar una versión más actualizada que la disponible en el repositorio oficial de Ubuntu, debes usar un PPA (del inglés *Personal Package Archive*). Esta es una forma de distribuir *software* que no está disponible en los repositorios oficiales, y con Ansible se puede hacer utilizando el módulo apt\_repository. En nuestro caso, vamos a instalar la versión presente en el repositorio oficial:

```
- name: Install PHP
  apt: name=php state=present update_cache=yes
```

Si ejecutas vagrant provision ahora, debería instalarse con éxito. Para asegurarnos de que todo funciona como esperamos, puedes ahora ejecutar vagrant ssh y así acceder a la máquina virtual.

Una vez dentro, puedes ejecutar `php --version` y comprobar así que muestra algo parecido a:

```
vagrant@ubuntu-bionic:~$ php --version
PHP 7.2.15-0ubuntu0.18.04.1 (cli) (built: Feb 8 2019 14:54:22) (NTS)
Copyright (c) 1997-2018 The PHP Group
Zend Engine v3.2.0, Copyright (c) 1998-2018 Zend Technologies
with Zend OPcache v7.2.15-0ubuntu0.18.04.1, Copyright (c) 1999-2018,
by Zend Technologies
```

Se ha instalado correctamente, por lo que ahora vamos a seguir instalando el resto de paquetes de PHP que necesitaremos. Vamos a usar una lista de nombres en el módulo `apt` para que el *playbook* sea más fácil de leer:

```
- name: Install PHP
  apt:
    state: present
    update_cache: yes
    name:
      - php
      - php-fpm
      - php-mysql
      - php-xml
```

La instalación de PHP también instalará Apache2, un servidor web que no vamos a usar en este ejemplo. No hay manera de evitarlo, pero se puede eliminar tan pronto como se instale, añadiendo la tarea siguiente al *playbook*:

```
- name: Remove apache2
  apt: name=apache2 state=absent
```

## Instalación de MySQL

Una vez instalado PHP y eliminado Apache, puedes continuar con la instalación de la siguiente dependencia: MySQL. Añade lo siguiente al *playbook*:

```
# MySQL
- name: Install MySQL
  apt:
    state: present
    update_cache: yes
    name:
      - mysql-server
      - python-mysqldb
```

Es conveniente que ejecutes Ansible regularmente mientras desarrollas un *playbook*, para ir validando su correcto funcionamiento, así que vamos a ejecutar `vagrant provision` en este momento para instalar todos los paquetes de PHP y MySQL. Puede tardar varios minutos en ejecutarse, pero debería finalizar con éxito.

Con esto es suficiente para dejar instalado el gestor de base de datos MySQL. Sin embargo, la instalación por defecto de MySQL genera una contraseña de `root` vacía y también deja accesibles a usuarios anónimos algunas de las bases de datos de prueba. Generalmente, en una instalación manual se ejecutaría el *script* `mysql_secure_installation` para ordenar todos estos archivos, pero dado que estamos ejecutando en un entorno automatizado con Ansible, tendremos que hacer el proceso de mantenimiento nosotros mismos.

Estas son las tareas que realizaremos con Ansible:

1. Cambiaremos la contraseña por defecto de `root`.
2. Eliminaremos los usuarios anónimos.
3. Eliminaremos las bases de datos de pruebas.

Para el cambio de la contraseña por defecto, debes generar una nueva. Para ello, puedes utilizar el comando `openssl` para generar una nueva contraseña de 15 caracteres. Utilizaremos el módulo `command` para hacerlo de la siguiente manera:

```
- name: Generate new root password
  command: openssl rand -hex 7
  register: mysql_new_root_pass
```

Aquí, hemos utilizado una característica de Ansible que no se había visto hasta ahora, que es el registro (`register`). Al usar `register` en una tarea, le decimos a Ansible que queremos guardar el valor de retorno de la ejecución de la tarea como una variable para poder utilizarlo posteriormente en un *playbook*.

La siguiente acción que vamos a realizar es la de quitar las bases de datos de prueba y los usuarios anónimos. Esto es fácil de hacer en Ansible gracias a los módulos `mysql_db` y `mysql_user`. Se debe hacer esto antes de cambiar la contraseña de `root` para que Ansible pueda realizar los cambios. Añade lo siguiente al *playbook*:

```
- name: Remove anonymous users
  mysql_user: name="" state=absent
- name: Remove test database
  mysql_db: name=test state=absent
```

La última acción que vamos a realizar para finalizar adecuadamente la instalación de MySQL es la de cambiar la contraseña de `root` y visualizarla por pantalla. Para ello, usaremos el valor previamente devuelto por `openssl` en un módulo específico de MySQL. La contraseña se debe establecer para cada uno de los `hosts` que deba poder acceder a la base de datos como usuario `root`. Vamos a utilizar para esto la variable especial `ansible_hostname`, que contiene el valor del nombre del `host` actual de la máquina y, acto seguido, estableceremos la contraseña para los tres valores distintos que se pueden utilizar para referirse a `localhost`:

```
- name: Update root password
```

```

mysql_user: name=root host={{item}}
  password={{mysql_new_root_pass.stdout}}
  loop:
    - "{{ansible_hostname}}"
    - 127.0.0.1
    - ::1
    - localhost
- name: Output new root password
  debug: msg="New root password is {{mysql_new_root_pass.stdout}}"

```

Sería posible en MySQL tener una contraseña diferente por cada usuario y lugar de conexión distinto. Hemos utilizado *loop* para establecer la contraseña de cada nombre de *host* que se refiere a la propia máquina iterando con cada uno de los valores de la lista de *loop* y sustituyéndolos en *{{item}}*, incluyendo *ansible\_hostname*, una variable que se rellena automáticamente con el nombre de *host* de la máquina actual. Para cambiar la contraseña, puedes utilizar el módulo *mysql\_user* pasándole un nombre de usuario, el *host* y el valor de la contraseña en el atributo *password*. En este caso, se pasa el valor *stdout* de la variable *mysql\_new\_root\_pass* (que contiene el texto que fue devuelto al terminal) que fue definida con el resultado de la llamada a *openssl* para generar la contraseña para el usuario *root*.

Llegados a este punto, la instalación es segura, pero no está terminada. Ansible ejecuta los comandos de base de datos sin proporcionar una contraseña, lo que era adecuado cuando no estaba establecida la contraseña de *root*, pero ahora fallará, dado que sí la tiene. Vas a necesitar definir un nuevo archivo de configuración para MySQL (*/root/.my.cnf*) que contenga la nueva contraseña de *root* para que este usuario pueda ejecutar comandos MySQL automáticamente.

Ansible proporciona varios mecanismos para crear o escribir archivos, como son los módulos *copy* y *template*. En este caso se trata de crear un archivo que va a contener varias líneas con variables, por lo que necesitarás utilizar el módulo *template* para llenar su contenido. Primero, hay que crear un directorio donde alojar la plantilla y crear ahí el archivo que vas a copiar. Vamos a ejecutar los siguientes comandos desde

el terminal (desde el mismo directorio que el del archivo `Vagrantfile`) para crear los directorios y archivos que necesitamos:

```
mkdir -p provisioning/templates/mysql  
touch provisioning/templates/mysql/my.cnf
```

Ahora edita el fichero `my.cnf` e incluye el siguiente contenido:

```
[client]  
user=root  
password={{mysql_new_root_pass.stdout}}
```

También es necesario indicarle a Ansible que procese esta plantilla y la copie en tu máquina virtual; esto se hace mediante el módulo `template`. Para ello, añade lo siguiente al *playbook*:

```
- name: Create my.cnf  
  template: src=templates/mysql/my.cnf dest=/root/.my.cnf
```

El fichero resultante que se va a copiar al *host* va a contener el nombre de usuario y la contraseña del usuario `root` de MySQL. Esto se necesita para posibilitar a Ansible que realice cambios de manera automatizada, sin intervención del usuario.

Cabe señalar que cada vez que se ejecute el *playbook*, se va a generar una nueva contraseña de `root` para MySQL. Aunque es conveniente cambiar las contraseñas de `root` con cierta frecuencia, es probable que no deseas hacerlo cada vez que ejecutas. Para inhibir este comportamiento, puedes indicarle a Ansible que no ejecute determinados comandos cuando exista un archivo específico. Ansible proporciona la opción `creates` para determinar si ya existe un archivo antes de ejecutar el módulo:

```
- name: Generate new root password  
  command: openssl rand -hex 7 creates=/root/.my.cnf  
  register: mysql_new_root_pass
```

Si el archivo /root/.my.cnf no existe al ejecutar el *playbook*, mysql\_new\_root\_pass.changed tendrá el valor true. En caso contrario, si el archivo existe, tendrá el valor false. Este valor se podrá usar en el resto del *playbook* para condicionar cualquier tarea en función de ese valor. El siguiente listado de tareas son un ejemplo de cómo mostrar la nueva contraseña de root si el archivo .my.cnf no existía y se acaba de crear, y muestra otro mensaje diferente en caso de ya existir:

```
- name: Generate new root password
  command: openssl rand -hex 7 creates=/root/.my.cnf
  register: mysql_new_root_pass
  # If /root/.my.cnf doesn't exist and the command is run
- debug: msg="New root password is {{mysql_new_root_pass.stdout}}"
  when: mysql_new_root_pass.changed
  # If /root/.my.cnf exists and the command is not run
- debug: msg="No change to root password"
  when: not mysql_new_root_pass.changed
```

Una vez que hemos realizado el cambio incluyendo creates=/root/.my.cnf, se debe agregar el argumento when a todas las tareas que queramos condicionar. Tras hacer estos cambios, la sección MySQL del *playbook* deberá ser como sigue:

```
# MySQL
- name: Install MySQL
  apt:
    state: present
    update_cache: yes
    name:
      - mysql-server
      - python-mysqldb
- name: Generate new root password
  command: openssl rand -hex 7 creates=/root/.my.cnf
  register: mysql_new_root_pass
- name: Remove anonymous users
  mysql_user: name="" state=absent
  when: mysql_new_root_pass.changed
- name: Remove test database
  mysql_db: name=test state=absent
```

```

when: mysql_new_root_pass.changed
- name: Output new root password
    debug: msg="New root password is {{mysql_new_root_pass.stdout}}"
when: mysql_new_root_pass.changed
- name: Update root password
  mysql_user: name=root host={{item}}
  password={{mysql_new_root_pass.stdout}}
loop:
  - "{{ansible_hostname}}"
- 127.0.0.1
- ::1
- localhost
when: mysql_new_root_pass.changed
- name: Create my.cnf
  template: src=templates/mysql/my.cnf dest=/root/.my.cnf
when: mysql_new_root_pass.changed

```

Ahora puedes ejecutar `vagrant provision` para generar la nueva contraseña de root y dejar limpia la instalación de MySQL. Si vuelves a ejecutar `vagrant provision` otra vez, podrás ver que todos estos pasos se omiten, al haber sido ya ejecutados:

```

TASK [Remove anonymous users] ****
skipping: [default]

```

Aquí termina la configuración de MySQL. Hemos comenzado descargando e instalando todos los paquetes requeridos y hemos reforzado la instalación mediante la inhabilitación de los usuarios anónimos, las bases de datos de pruebas y la creación de la contraseña para root. Con ello ya tenemos tanto PHP como MySQL instalados, y ahora necesitaremos instalar un servidor web que gestione las peticiones entrantes.

## Instalación de NginX

Antes de poder comenzar con la instalación de WordPress, es necesario instalar y configurar NginX. NginX (que es una alternativa al servidor Apache) va a actuar como nuestro servidor web, recibiendo las peticiones HTTP de los usuarios y redirigiéndolas

a PHP, donde WordPress procesará la petición y enviará la respuesta. La configuración que se requiere para NginX es algo compleja. Lo vamos a ir haciendo una vez que tengamos NginX instalado. Ahora, vamos a instalar NginX añadiendo lo siguiente al final del *playbook*:

```
# nginx
- name: Install nginx
  apt: name=nginx state=present
- name: Start nginx
  service: name=nginx state=started
```

Ejecuta `vagrant provision` nuevamente para instalar NginX y arrancarlo. Si ahora pruebas a acceder a 192.168.33.20 desde el navegador web, se mostrará la página de bienvenida *Welcome to nginx*. A continuación, cambiaremos la configuración por defecto de NginX para, en lugar de mostrar esta página, redirigir las peticiones a WordPress. Por lo tanto, debes cambiar la configuración del *host* virtual NginX predeterminado para que reciba las peticiones y las reenvíe.

Ejecuta los siguientes comandos en el mismo directorio que el archivo `Vagrantfile` para crear el fichero de plantilla que utilizaremos para configurar NginX:

```
mkdir -p provisioning/templates/nginx
touch provisioning/templates/nginx/default
```

También es necesaria la tarea que copia este fichero en tu máquina virtual mediante el módulo `template`. Añadimos la siguiente tarea al *playbook* para realizarlo:

```
- name: Create nginx config
template:      src=templates/nginx/default          dest=/etc/nginx/sites-
available/default
```

Si ejecutas ahora `vagrant provision`, el archivo de configuración se machacará con un archivo vacío. Vamos a continuación a llenar el archivo de plantilla con la configuración NginX que es necesaria para poder ejecutar WordPress.

Edita la plantilla provisioning/templates/nginx/default e incluye en ella el siguiente contenido:

```
server {
    server_name book.example.com;
    root /var/www/book.example.com;
    index index.php;
    location = /favicon.ico {
        log_not_found off;
        access_log off;
    }
    location = /robots.txt {
        allow all;
        log_not_found off;
        access_log off;
    }
    location ~ /\. {
        deny all;
    }
    location ~* /(?:uploads|files)/*\.php$ {
        deny all;
    }
    location / {
        try_files $uri $uri/ /index.php$args;
    }
    rewrite /wp-admin$ $scheme://$host$uri/ permanent;
    location ~* ^.+\.(ogg|ogv|svg|svgz|eot|otf|woff|mp4|ttf|rss|atom|jpg|jpeg|gif|png|ico|zip|tgz|gz|rar|bz2|doc|xls|exe|ppt|tar|mid|midi|wav|bmp|rtf)$ {
        access_log off;
        log_not_found off;
        expires max;
    }
    location ~ [^/]\.php(/|$) {
        fastcgi_split_path_info ^(.+?\.php)(/.*)$;
        if (!-f $document_root$fastcgi_script_name) {
            return 404;
        }
        include fastcgi_params;
    }
}
```

```
    fastcgi_index index.php;
    fastcgi_param SCRIPT_FILENAME
$document_root$fastcgi_script_name;
    fastcgi_pass php;
}
}
```

Se trata de un fichero de configuración NginX bastante estándar que impide el acceso a archivos potencialmente confidenciales y deshabilita el log de peticiones comunes, como `favicon.ico` y `robots.txt`.

La forma en la que NginX gestiona las peticiones entrantes de PHP es redirigirlas a un procesador de PHP (PHP worker) y quedar a la espera de la respuesta. Para ello, necesita conocer la dirección del procesador de PHP. En la terminología de NginX, este procesador de PHP es lo que se denomina un `upstream`.

Vamos a añadir una definición de `upstream` en el fichero de configuración de NginX para que sepa dónde tiene que redirigir la petición. Incluye el siguiente fragmento al inicio de la plantilla, justo antes de la línea de apertura de la definición `server {`:

```
upstream php {
    server unix:/run/php/php7.0-fpm.sock;
}
```

Esto hará que cualquier petición recibida sea transferida al proceso que escucha en ese socket. Al `upstream` lo hemos llamado `php`, pero podría haberse llamado de cualquier otra forma, como por ejemplo `wordpress`, lo que sería:

```
upstream wordpress {
    server unix:/run/php/php7.0-fpm.sock;
}
```

Dentro del bloque `upstream`, se define el servidor al que redirigir la petición. En nuestro caso, estamos redirigiendo la petición a un socket que está ubicado en

/run/php/php7.0-fpm.sock. Para saber en qué socket está escuchando el proceso PHP-FPM, puedes acceder a la máquina con vagrant ssh y ejecutar el comando:

```
cat /etc/php/7.0/fpm/pool.d/www.conf | grep "listen ="
```

NginX sabe cómo usar este upstream, ya que se le indica qué debe buscar en el archivo de configuración. Las líneas que se muestran a continuación son las más importantes:

```
location ~ [^/]\.php(/|$) {
```

Esto quiere decir que la configuración que se especifica entre las llaves solo aplica cuando el recurso web solicitado en la URL termina en .php. Dentro de las llaves, hay una línea que contiene fastcgi\_pass:

```
fastcgi_pass php;
```

Los procesadores de PHP (PHP-FPM) utilizan el protocolo FastCGI para recibir peticiones y enviar respuestas. FastCGI no entra en el temario de esta asignatura, pero básicamente lo que aquí se indica es que cada vez que haya una petición terminada en .php, se va a enviar al upstream que se llama php utilizando el protocolo FastCGI. Esto es suficiente para que NginX y PHP se integren y atiendan las peticiones de los usuarios.

## 7.4. Tareas y manejadores

Al ejecutar ahora vagrant provision el fichero de configuración de NginX se actualizará. Sin embargo, el proceso de NginX necesita ser reiniciado para que los cambios que se hicieron en el fichero de configuración se recarguen. Podrías añadir la siguiente tarea para reiniciar NginX al final del playbook:

```
- name: Restart nginx
```

```
service: name=nginx state=restarted
```

Sin embargo, esto haría que se reiniciase NginX cada vez que ejecutes el *playbook*. La mejor manera de trabajar en Ansible con procesos que necesitan un reinicio cuando cambia algún parámetro de su configuración es usar un manejador. Los manejadores son iguales que las tareas, pero no se ejecutan por orden de aparición en el *playbook*, sino que pueden ser invocados desde cualquier otro lugar. Elimina ahora la tarea `Restart nginx` si la habías añadido y agrega el siguiente fragmento al final del *playbook*, al mismo nivel y sangría que `tasks`:

```
handlers:  
- name: restart nginx  
  service: name=nginx state=restarted
```

Este código usará el módulo `service` para reiniciar NginX cuando se active el manejador. Puede activarse cada vez que haya cambios en el archivo de configuración, modificando la tarea de la configuración de esta forma:

```
- name: Create nginx config  
template:      src=templates/nginx/default          dest=/etc/nginx/sites-  
available/default  
  notify: restart nginx
```

Si ejecutas ahora `vagrant provision`, el manejador no se va a ejecutar. Esto es debido a que se acababa de ejecutar `vagrant provision` anteriormente, por lo que la configuración de NginX ya se había generado, y Ansible detecta entonces que no se requieren acciones.

Hasta aquí hemos hecho bastantes cambios, pero ejecutando `vagrant provision` tras cada uno de ellos. Es un buen momento de probar un aprovisionamiento completo, ejecutando `vagrant destroy` seguido de `vagrant up` para confirmar que todo se instale y configure adecuadamente.

Después de ejecutar `vagrant up`, la nueva configuración se desplegará y NginX se reiniciará. Para probar esto, edita el fichero `hosts` en tu máquina local (no en la máquina virtual) y añade la dirección IP y el dominio que ha estado utilizando al final del fichero:

```
192.168.33.20 book.example.com
```

Todas las dependencias que necesitas para ejecutar WordPress están ya instaladas. Si deseas asegurarte de que todo está configurado correctamente, puedes iniciar sesión en la máquina virtual con `vagrant ssh`, y ejecutar los siguientes comandos:

```
sudo mkdir -p /var/www/book.example.com  
echo "<?php echo date('H:i:s');" | sudo tee  
/var/www/book.example.com/index.php  
exit
```

A continuación, podrás visitar <http://book.example.com> desde el navegador. Deberías poder ver la hora actual. Si aparece, ¡está funcionando correctamente!

## 7.5. Descarga de WordPress

Ya tenemos todos los prerequisitos instalados y el entorno preparado, por lo que se puede ya por fin descargar WordPress. Hay dos opciones distintas para hacerlo: se puede descargar WordPress manualmente y simplemente utilizar Ansible para copiarlo en el entorno, o se puede descargar directamente a través de Ansible.

Cada enfoque tiene sus ventajas e inconvenientes. Si lo descargas tú mismo, sabrás exactamente la versión que se está utilizando, pero entonces tendrás también que volver a descargar una nueva versión si deseas actualizarlo. Por el contrario, si decides descargarlo automáticamente con Ansible, siempre utilizarás la última versión, pero esto precisamente no te garantiza que todo vaya a funcionar de la

misma manera en que lo hicieron la última vez que se ejecutó el *playbook*, ya que ha podido cambiar cualquier cosa en el proceso de instalación o configuración que haga que ahora las tareas del *playbook* fallen, o no terminen la configuración. Se van a abordar ambas aproximaciones en las siguientes secciones.

## Descárgalo tú mismo

Para descargar WordPress manualmente, accede a <https://wordpress.org/> y descarga la última versión. Crea el subdirectorio files dentro del directorio provisioning y guarda ahí la descarga, con el nombre wordpress.zip. Alternativamente, puedes descargar la última versión de WordPress mediante el cliente HTTP curl de línea de comandos, tal como:

```
mkdir -p provisioning/files  
curl https://wordpress.org/latest.zip >  
provisioning/files/wordpress.zip
```

Lo siguiente será copiarlo en la máquina virtual y, dado que solo lo necesitas temporalmente, lo puedes copiar en el directorio /tmp mediante la siguiente tarea:

```
# Wordpress  
- name: Copy wordpress.zip into tmp  
  copy: src=files/wordpress.zip dest=/tmp/wordpress.zip
```

Ahora, cada vez que ejecutes Ansible, se copiará WordPress en la máquina virtual, y estará disponible para ser utilizado. Siempre usarás la misma versión, pero cuando quieras actualizar la versión que se usa, no tienes más que descargar la nueva versión y sobrescribirla sobre el archivo files/wordpress.zip. A partir de entonces, se utilizará la nueva versión cada vez que vuelvas a ejecutar Ansible.

## Descarga automática

La otra alternativa es hacer que Ansible lo descargue automáticamente. Puedes probar este método, pero en el resto de la documentación se asumirá que se ha utilizado el método manual para descargar WordPress. Por ello, ten en cuenta que esta sección se incluye solo como referencia.

Para esta descarga usarás los módulos `uri` y `get_url`. Aunque la descarga se hace a través de HTTPS, toda precaución es poca al descargar cualquier aplicación desde Internet y ejecutarla, más si cabe cuando se hace la descarga automatizada.

Comienza por utilizar el módulo `uri` para acceder al hash sha1 de la última versión de WordPress, almacenando el valor obtenido en la variable `wp_checksum`. Ansible utilizará la suma de comprobación (`checksum`) para asegurarse de que el contenido del fichero zip que se descarga es realmente lo que se espera, y no ha sido modificado:

```
# WordPress
- name: Get WordPress checksum
  uri: url=https://wordpress.org/latest.zip.sha1 return_content=true
  register: wp_checksum
```

Una vez que tienes la suma de comprobación SHA1 con la que comparar, se puede descargar el propio WordPress. Esta vez, se utilizará el módulo `get_url`. Especifica una URL, el destino donde se debe guardar el fichero y el `checksum`:

```
- name: Download WordPress
  get_url: url=https://wordpress.org/latest.zip dest=/tmp/wordpress.zip
  checksum="sha1:{{wp_checksum.content}}"
```

Tal como hemos indicado anteriormente, al usar este método vamos a obtener la última versión de WordPress cada vez que se ejecute el *playbook*. Aunque, tal como se ha dicho, de ahora en adelante se asumirá que se ha utilizado el método de

descarga manual de WordPress y copia al entorno, es útil saber cómo se puede descargar un archivo a demanda y comprobar su contenido mediante la suma de comprobación, y puede ser necesario en otras ocasiones, por lo que es conveniente conocer este mecanismo.

Llegados a este punto, los argumentos que se están pasando a Ansible son cada vez más largos y potencialmente pueden tener que partirse en varias líneas. Ansible también soporta un segundo formato para especificar los argumentos de un módulo pensado precisamente para argumentos más largos. Por ejemplo, la tarea anterior se puede especificar también con el siguiente formato:

```
- name: Download WordPress
  get_url:
    url: https://wordpress.org/latest.zip
    dest: /tmp/wordpress.zip
    checksum: "sha1:{wp_checksum.content}"
```

Las diferencias son meramente estéticas: cada argumento está ahora en su propia línea y el signo igual entre nombre del argumento y su valor ha sido substituido por dos puntos. Puedes utilizar el formato que más te guste, ya que ambos son equivalentes funcionalmente.

## Configuración de la instalación de WordPress

Ya tienes todas tus dependencias instaladas y WordPress descargado. Ha llegado el momento de descomprimir la aplicación y de poner en marcha el blog.

Lo primero que tendrás que hacer es descomprimir wordpress.zip. Ansible proporciona un módulo denominado `unarchive` que sabe cómo extraer los formatos de archivo comprimido más comunes:

```
- name: Unzip WordPress
  unarchive: src=/tmp/wordpress.zip dest=/tmp copy=no
```

```
creates=/tmp/wordpress/wp-settings.php
```

Ya deberías estar familiarizado con los argumentos de muchos de los módulos (tanto `src` como `dest` aparecen una y otra vez, por ejemplo). Puedes haber detectado que `copy=no` se ha añadido a los argumentos. Esto le indica a Ansible que el fichero ya está disponible en nuestro entorno.

Debes indicarlo para que el comando funcione, tanto si has descargado WordPress manualmente como si lo ha hecho Ansible de manera automatizada.

Si cambiases la tarea anterior para que se parezca al último fragmento sin el argumento `copy`, podrías entonces eliminar la tarea `copy` que se añadió, ya que Ansible copiaría el archivo origen en el entorno antes de descomprimirlo. No obstante, vamos a dejar la copia tal como está, e incluimos `copy=no` en la tarea de extracción.

Si ejecutas ahora el *playbook*, Ansible va a encontrar un error al intentar extraer WordPress:

```
Failed to find handler for \"tmp/wordpress.zip\". Make sure the required command to extract the file is installed.
```

Este error se produce porque, por defecto, `unzip` no está instalado en Ubuntu. Es una buena práctica contar con una tarea para instalar todas las utilidades comunes que se necesitan como primera tarea del *playbook*. Añade este fragmento al principio de la lista de tareas del *playbook* (antes de instalar PHP):

```
- name: Install required tools
  apt: name={{item}} state=present
  loop:
    - unzip
```

Si ejecutas Ansible tras haber agregado la tarea para instalar `unzip`, el *playbook* se ejecutará por completo correctamente. El archivo zip que hemos descomprimido contenía una carpeta `wordpress`, por lo que todos los ficheros necesarios se encuentran en `/tmp/wordpress`. Sin embargo, esta no es la ruta de aplicación que se especificó en la configuración de NginX, así que vamos a copiar todos los ficheros necesarios en la ubicación correcta. El módulo `copy` admite también la copia de directorios de un lugar a otro en el servidor remoto, si especificamos un directorio en el atributo `src`.

```
- name: Create project folder
  file: dest=/var/www/book.example.com state=directory
- name: Copy WordPress files
  copy: source=/tmp/wordpress/. dest=/var/www/book.example.com
```

Una vez ejecutado esto, prueba a visitar `http://book.example.com` en el navegador web; debería aparecer una pantalla de instalación de WordPress, en la que se indica que es necesario que se conozcan todas las credenciales de la base de datos para proceder con el proceso de instalación. No se debe otorgar a WordPress acceso root a la base de datos, así que debemos crear un usuario de MySQL dedicado para utilizarlo con WordPress, añadiendo lo siguiente al *playbook*:

```
- name: Create WordPress MySQL database
  mysql_db: name=wordpress state=present
- name: Create WordPress MySQL user
  mysql_user:      name=wordpress      host=localhost      password=bananas
  mysql_user:      priv=wordpress.*:ALL
```

Esto creará la base de datos `wordpress` y el usuario llamado `wordpress` con la contraseña bananas. El nuevo usuario tiene todos los privilegios sobre la base de datos `wordpress`, pero no tiene acceso a ninguna otra. Después de ejecutar nuevamente el *playbook* para crear la base de datos y el usuario, ya puedes volver al navegador web y continuar con el proceso de instalación.

Una vez proporcionados todos los detalles requeridos, WordPress mostrará un error indicando que no tiene permiso para escribir `wp-config.php`. Esto es algo bueno, ya que es peligroso permitir al servidor web que pueda escribir cualquier fichero.

En lugar de permitir que WordPress escriba el fichero `wp-config.php`, vamos a copiar el archivo de configuración y hacer que Ansible lo instale en su lugar. Crea el archivo `provisioning/templates/wordpress/wp-config.php` y pega el contenido del archivo de configuración en él.

Una vez hecho esto, tienes que añadir la siguiente tarea para copiar este archivo en la ubicación de destino correcta:

```
- name: Create wp-config
template: src=templates/wordpress/wp-config.php
dest=/var/www/book.example.com/wp-config.php
```

Tras haber añadido esta tarea, ejecuta el *playbook* nuevamente mediante el comando `vagrant provision` desde el terminal. Al hacerlo, puede que obtengas un mensaje de error parecido a este:

```
AnsibleError: ERROR! template error while templating string
```

Si recibes este mensaje de error, echa un vistazo a los contenidos de tu fichero `wpconfig.php`. Si ves algún lugar que tenga “{{” o “}}” (llaves dobles, de apertura o cierre) en una cadena, estos caracteres son especiales para Ansible, ya que en el sistema de plantillas Jinja se usan para indicar la posición de las variables. Pero WordPress puede haber generado esta cadena como parte de sus claves secretas. Si es así, edita el fichero y cambia esos caracteres por cualquier otro para que Jinja no intente interpretar esa cadena.

Una vez que el *playbook* se haya ejecutado entero correctamente, volvemos al navegador web y pulsamos «Ejecutar la instalación». Contesta a todas las preguntas

y pulsa en «Instalar WordPress». Si vuelves a visitar ahora <http://book.example.com> con el navegador web, deberías poder ver a WordPress funcionando con un mensaje de «Hello World» saludándote. Enhorabuena, has terminado la instalación y configuración de WordPress.

## 7.6. Backup de la configuración

Si ahora destruyes el entorno y volvieses a crearlo, todavía estarías al 90 por ciento de completar la instalación y configuración de WordPress, ya que te faltaría proporcionar los detalles sobre su sitio web al acceder con el navegador. Toda la información que se rellena en esta última fase queda almacenada en la base de datos, así que vamos a hacer una copia de seguridad (*backup*) para que Ansible la importe automáticamente.

Vamos a iniciar sesión en el entorno mediante `vagrant ssh` y ejecutar los comandos siguientes para crear una copia de seguridad en un archivo de SQL que utilizará posteriormente en el *playbook* para recargar esta configuración:

```
sudo su -  
mkdir -p /vagrant/provisioning/files  
mysqldump wordpress > /vagrant/provisioning/files/wp-database.sql  
exit  
exit
```

Finalmente, vamos a escribir una tarea que realice la importación de esta copia de seguridad en la base de datos. Debemos realizar algo de trabajo adicional para asegurarnos de que no se sobrescriben las bases de datos que ya existen, como, por ejemplo, para evitar reemplazar una base de datos de producción con una copia de seguridad de desarrollo.

Utilizaremos ahora una nueva función denominada `ignore_errors`. Generalmente, cuando un comando devuelve un código de retorno distinto de cero quiere decir que se ha producido algún problema, y Ansible devuelve un error. Al utilizar `ignore_errors` en un comando le indicamos a Ansible que no importa si el comando devuelve un código de retorno distinto de cero:

```
- name: Does the database exist?  
command: mysql -u root wordpress -e "SELECT ID FROM  
wordpress.wp_users LIMIT 1;"  
register: db_exist  
ignore_errors: true
```

Este comando ejecuta una consulta SQL para obtener el primer usuario de la base de datos de WordPress. Esto fallará si la base de datos no existe todavía, que es lo que utilizaremos para ejecutar la restauración de la base de datos. El comando almacena el resultado en `db_exist` para poder utilizarlo en tareas posteriores. Si vamos a necesitar restaurar la base de datos, deberemos primero copiar el *backup* en el entorno antes de importarla, por lo que necesitaremos las siguientes dos tareas para completar la importación:

```
- name: Copy WordPress DB  
copy: src=files/wp-database.sql dest=/tmp/wp-database.sql  
when: db_exist.rc > 0  
- name: Import WordPress DB  
mysql_db: target=/tmp/wp-database.sql state=import name=wordpress  
when: db_exist.rc > 0
```

Solo debemos realizar la copia y recuperación de la base de datos cuando `db_exist.rc` es mayor que 0, es decir, cuando el código de retorno (*return code*) ha indicado que se ha producido un error (la base de datos no existe). Si ejecutamos nuestro *playbook* ahora, deberíamos poder comprobar que estas tareas no se ejecutan, ya que la base de datos ya existe.

## 7.7. Idempotencia

Si ejecutas nuevamente `vagrant provision`, podrás ver que tiene una tarea que reporta cambios cada vez que se ejecuta, a pesar de que no los hay. Esto puede llegar a ser un problema, ya que podría activar manejadores o tener cualquier otro efecto secundario que no se desea. Es preferible que los *playbooks* indiquen «OK» o «saltado» para cada tarea en la salida de la ejecución del *playbook* si realmente no ha cambiado nada.

La tarea que siempre reporta cambios (*changed*) es el comando que comprueba si la base de datos ya existe. El módulo de comandos siempre informa que ha cambiado algo, ya que no sabe lo que realmente hace el comando. Este comportamiento se puede suprimir usando la opción `changed_when`, que nos permite controlar cuándo Ansible debe indicar si la tarea ha realizado cambios o no. Si la expresión que se ha proporcionado es verdadera, Ansible registrará que se realizaron cambios y activará cualquier manejador que necesite ejecutarse. Si por el contrario es falsa, Ansible registrará que no se ha realizado ningún cambio y no se activará ningún manejador.

El fragmento a continuación muestra un ejemplo de cómo utilizar `changed_when`. Se lista el contenido del directorio `/tmp` y se comprueba si aparece la palabra `'wordpress'` en la salida del comando. En caso afirmativo, Ansible informará de que la tarea ha realizado cambios:

```
- name: Example changed_when
- command: ls /tmp
  register: demo
  changed_when: '"wordpress" in demo.stdout'
```

Por el contrario, si `'wordpress'` no aparece en la salida del comando, Ansible informará de que la tarea no ha producido ningún cambio, y devolverá OK en la salida.

En nuestro caso, nunca va a ser necesario que el comando que comprueba si existe la base de datos devuelva changed, por lo que podemos especificar changed\_when: false para que siempre nos devuelva OK. Modifica por tanto la tarea que comprueba la base de datos como se indica a continuación para que el *playbook* vuelva a ser idempotente:

```
- name: Does the database exist?  
  command: mysql -u root wordpress -e"SELECT ID FROM  
  wordpress.wp_users LIMIT 1;"  
  register: db_exist  
  ignore_errors: true  
  changed_when: false
```

Llegados a este punto, podemos ejecutar vagrant destroy y confirmar la destrucción, y, a continuación, vagrant up para probar a levantar el entorno desde cero. El *playbook* se ejecutará y proporcionará automáticamente la instalación de WordPress completa. Ahora tardará algunos minutos, ya que realizará todos los pasos, instalando todas las dependencias, el propio WordPress y creando la base de datos a partir del *backup*.

En el vídeo *Mejores prácticas en Ansible* podrás seguir profundizando y aclarar conceptos.



Accede al vídeo

---

## 7.8. Referencias bibliográficas

Heap, M. (2016). *Ansible: from Beginner to Pro*. Apress.

Hochstein, L. y Moser, R. (2014). *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. O'Reilly Media.

Herramientas de Automatización de Despliegues

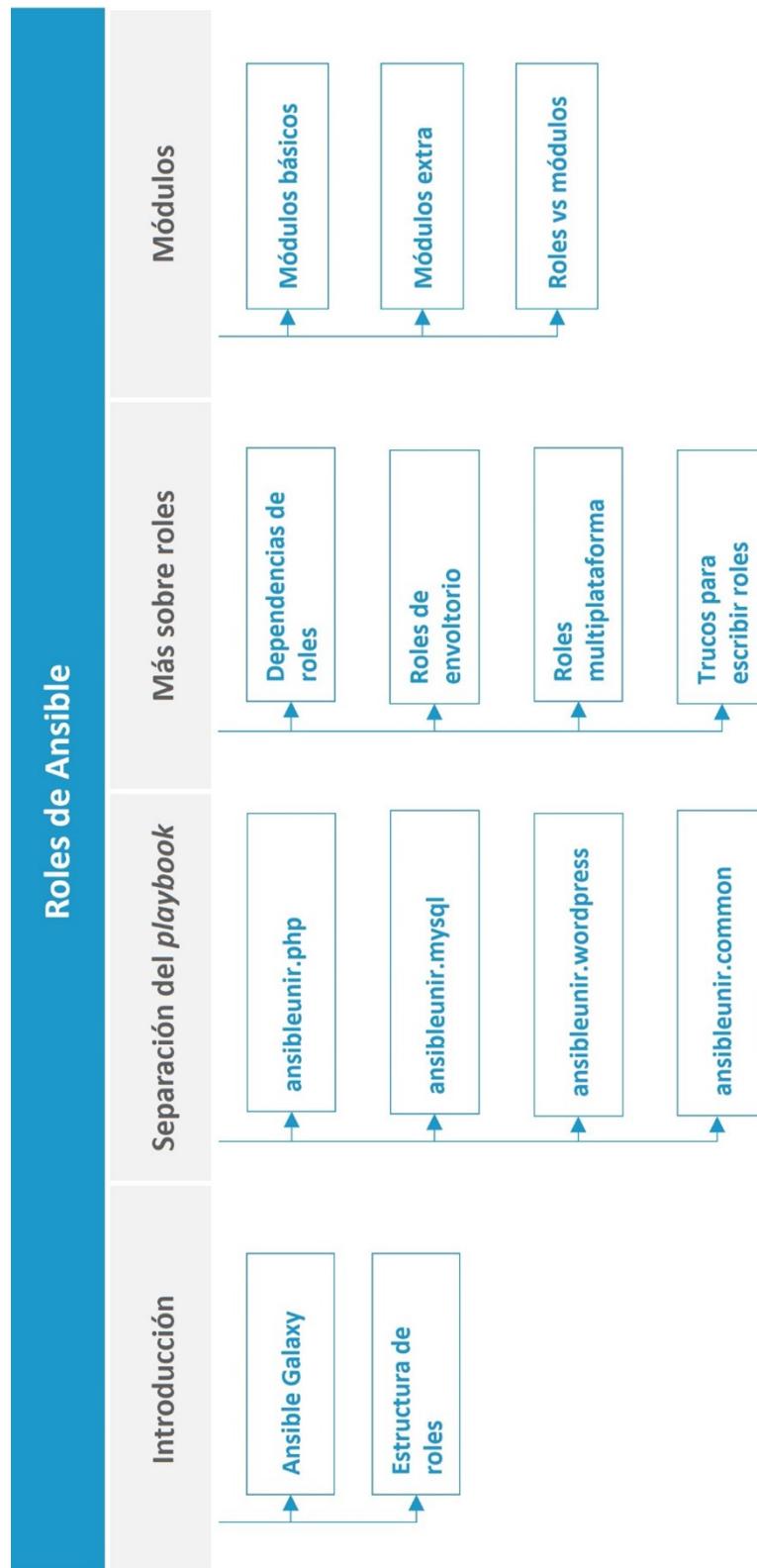
---

## Roles de Ansible

# Índice

<b>Esquema</b>	<b>3</b>
<b>Ideas clave</b>	<b>4</b>
8.1. Introducción y objetivos	4
8.2. Ansible Galaxy	5
8.3. Estructura de roles	7
8.4. Separación del <i>playbook</i> de WordPress	11
8.5. Dependencias de roles	19
8.6. Creación de roles para distintas plataformas	22
8.7. Trucos para escribir roles	24
8.8. Módulos en Ansible	25
8.9. Referencias bibliográficas	27

# Esquema



## 8.1. Introducción y objetivos

Los roles son el mecanismo proporcionado por Ansible que nos permite empaquetar las tareas, los manejadores y todos los demás archivos relacionados en componentes reutilizables que puedes compartir y reutilizar mediante su inclusión en un *playbook*.

Los *playbooks* y los roles son similares en cierto sentido, pero a la vez muy diferentes entre sí. Un *playbook* es un archivo independiente que Ansible puede ejecutar y contiene toda la información necesaria para establecer el estado de la máquina a lo que se desea. Esto quiere decir que un *playbook* puede incluir variables, tareas, manejadores, roles e incluso otros *playbooks*, todo en un mismo archivo.

Un rol podría considerarse como un *playbook* que se separa en diferentes archivos. En vez de tener un único archivo que contenga todo lo que necesitamos, vamos a tener un archivo para variables, otro para las tareas y otro para los manejadores. Sin embargo, no se puede ejecutar un rol por sí mismo; es necesario incluirlo dentro de un *playbook* junto con la información de los *hosts* donde ejecutarlo.

El objetivo que se pretende conseguir en este tema es el siguiente:

- ▶ Refactorizar el *playbook* del tema «Ansible: instalación de WordPress» para que se divida en diferentes roles:
  - Uno que instala PHP.
  - Otro que instala NginX.
  - Otro que instala MySQL.
  - Otro para WordPress.

Esto no solo permitirá que continuemos trabajando con los *playbooks* y familiarizándonos con ellos, sino que también a su vez los convertirá en reutilizables. Tal como está nuestro ejemplo actualmente, si optásemos por instalar Drupal en lugar de WordPress, tendríamos que duplicar la configuración de toda dependencia y cambiar el final del *playbook*. Una vez que construyamos los roles a lo largo de este tema, podremos reutilizarlos en múltiples proyectos siempre que necesiten instalar un NginX, PHP o MySQL.

A continuación, puedes ver al vídeo *Sistemas de gestión de la configuración*:



Accede al vídeo

---

## 8.2. Ansible Galaxy

Los **roles** son uno de los conceptos principales de Ansible. Tienen tanta importancia que incluso cuentan con su propio repositorio centralizado y una herramienta de línea de comandos para manejarlos. Ansible Galaxy es el repositorio web donde los usuarios de Ansible pueden subir roles que ellos mismos han desarrollado para que otros usuarios a su vez puedan descargárselos y utilizarlos.

---

Para acceder a este repositorio y examinar los roles que contiene, puedes acceder a <https://galaxy.ansible.com> donde hay más de 20 000 roles disponibles. ¡Consúltalos y utilízalos en tus proyectos propios!

---

Como puede ocurrir con cualquier repositorio público, podemos encontrar contribuciones tanto buenas como malas. Al realizar una búsqueda puedes ordenar los roles del resultado según el número de descargas, ya que cuanto mayor sea este número, más probable es que el rol sea de calidad. Cada rol tendrá un enlace a su

código fuente, por lo que una vez que hayas profundizado lo suficiente en Ansible, deberías ser capaz de entender lo que está haciendo cualquier rol.

Al descargar un rol, puedes optar por instalarlo en tu máquina globalmente o solo localmente para un proyecto. Lo mismo que con cualquier otra dependencia, es recomendable que sea local a tu proyecto por si se da la circunstancia de que proyectos diferentes quieran usar la misma dependencia, pero con diferentes versiones. Para descargar un rol del repositorio, utiliza el comando `ansible-galaxy` proporcionando el parámetro `-p roles` para que el rol se instale en un directorio denominado `roles`. Debes ejecutar `ansible-galaxy` desde el mismo directorio donde se encuentra el archivo `playbook.yml`.

En Ansible Galaxy puedes encontrar algunos creadores de roles muy prolíficos, aunque uno que destaca significativamente es Jeff Geerling (*geerlingguy*). Sus roles son siempre de una gran calidad. Vamos a usar uno de los roles de Jeff como ejemplo para mostrar cómo se debería descargar:

```
ansible-galaxy install geerlingguy.git -p roles
```

Esta línea creará la carpeta **roles** en caso de que no exista y descargará el rol en ella. Para usar este rol, debes incluirlo desde un *playbook*. Como ya sabemos, primero le indicamos a Ansible en qué servidores se va a ejecutar, y ahora, a continuación, también le proporcionaremos una lista de roles para que Ansible los incluya:

```
---
- hosts: all
  roles:
    - geerlingguy.git
```

Si ejecutas ahora este *playbook*, el rol va a intentar instalar `git` en la máquina. Para probarlo, puedes incluir la sección `roles` en el *playbook* justo antes de la lista de tareas y a continuación ejecutar `vagrant provision`. En un *playbook* los roles se van a

ejecutar antes que las tareas, por lo que la salida del rol se encontrará en la parte superior de la salida del comando `vagrant provision`.

Debería ser similar a la siguiente imagen:

```
PLAY ****
TASK [setup] ****
ok: [default]

TASK [geerlingguy.git : Ensure git is installed (RedHat).] ****
skipping: [default] => (item=[])
TASK [geerlingguy.git : Update apt cache (Debian).] ****
ok: [default]

TASK [geerlingguy.git : Ensure git is installed (Debian).] ****
changed: [default] => (item=[u'git', u'git-svn'])

TASK [geerlingguy.git : include] ****
skipping: [default]
```

Figura 1. Ejemplo de la salida de la ejecución de un rol. Fuente: elaboración propia.

El rol de Jeff se encarga de instalar git tanto en RedHat como en los sistemas operativos basados en Debian. Debido a que esta máquina virtual ejecuta Ubuntu (que es derivado de Debian), se excluye la tarea RedHat y se ejecuta la tarea Debian para instalar los paquetes correspondientes.

### 8.3. Estructura de roles

Ahora que has aprendido cómo ejecutar un rol desde un *playbook*, vamos a crear un rol propio. Mediante la herramienta `ansible-galaxy` de línea de comandos, puedes también crear un nuevo rol, y lo vamos a hacer en la carpeta de roles. Al crear roles, existe una convención de nombres que debes seguir. Los nombres de los roles suelen tener el formato `<identificador>.<nombreRol>`; tal como `geerlingguy.git`. Simplemente con el nombre de rol, ya se sabe que es un rol de Jeff Geerling que instala Git.

Para el proyecto de ejemplo, vamos a utilizar `ansibleunir` como identificador, lo que significa que un rol que instala PHP se llamará `ansibleunir.php`. Vamos a crear ese rol; para ello, ejecuta los siguientes comandos en la misma carpeta que el fichero `playbook.yml`:

```
mkdir -p provisioning/roles  
cd provisioning/roles  
ansible-galaxy init ansibleunir.php
```

El comando `ansible-galaxy init` habrá creado un subdirectorio llamado `ansibleunir.php` bajo el directorio `roles` que contendrá la estructura y los ficheros básicos de un rol de Ansible:



Figura 2. Estructura de directorios y ficheros de un rol. Fuente: elaboración propia.

La mayor parte de los directorios que se crean son opcionales, pero vamos a explicar para qué se utiliza cada uno y qué funcionalidad proporciona al añadir contenido en los ficheros que incorpora.

- ▶ Cada rol debe contar con un **fichero README** que describe el propósito del rol, su funcionalidad y las variables que se podrán utilizar y personalizar en el rol.

- ▶ El archivo `defaults/main.yml` puede contener los valores por defecto que utilizarán las variables que maneja el rol, en caso de no haber proporcionado otro valor personalizado. También es posible definir variables en otras ubicaciones tales como `vars/main.yml` que actualizará cualquier variable que ya estuviera definida en `defaults/main.yml`, ya que tiene mayor prioridad.
- ▶ El directorio `files` es en el que se colocan los ficheros necesarios para la ejecución del rol. Podrá contener tanto elementos estáticos, ficheros de configuración, como cualquier otro tipo de fichero. Ten en cuenta que estos archivos no pueden ser manipulados de ninguna manera, solo se pueden copiar.
- ▶ `handlers/main.yml` es donde defines manejadores como `restart nginx`. El contar con todos los manejadores disponibles en un único lugar común es muy útil para cualquiera que utilice el módulo, para poder ver de un vistazo las acciones que están disponibles. Los manejadores se pueden invocar tanto desde el mismo rol, como desde otros roles y desde el *playbook* que incluye al rol.
- ▶ El archivo `meta/main.yml` contiene los metadatos del rol. Este archivo va a incorporar los metadatos que utiliza Ansible Galaxy si publicas el rol. También se pueden definir algunos parámetros, como la versión mínima de Ansible requerida, plataformas soportadas y cualquier otra dependencia de este rol.
- ▶ El archivo `tasks/main.yml` es el punto de entrada del rol. Incorpora la sección de tareas que está contenida en tu *playbook*. Las acciones que están definidas en este archivo son las que procesará Ansible cuando se ejecute el rol.
- ▶ El directorio `templates` contiene todas las plantillas que necesitan ser procesadas por el procesador de plantillas `jinja2` para así sustituir las variables necesarias en el archivo (y cualquier otro procesamiento soportado por `Jinja`) antes de copiarlo en el sistema de destino.

- ▶ El directorio tests es donde debes incluir los *playbooks* de prueba que usan el rol. Esto se utiliza para definir pruebas automatizadas del rol, que podrán ejecutarse mediante un sistema de integración continua, tal como Jenkins o Travis CI. Este tipo de sistemas (de integración continua) son herramientas que detectan cuándo realizas modificaciones en el código fuente de un programa y desencadenan acciones relacionadas; típicamente, la compilación o construcción de ese código fuente, la ejecución de pruebas para el proyecto, empaquetado de los binarios o cualquier otra cosa que se pueda expresar en un *script*.

De todos estos directorios que hemos explicado, no todos son requeridos. El rol puede ser de gran utilidad, aunque solo incorpore un fichero tasks. La mayor parte del tiempo trabajaremos en este directorio tasks, con algunos ficheros de apoyo en los directorios handlers y templates.

Cabe señalar que cada carpeta contiene un archivo dentro que se llama `main.yml`. Este es el nombre de archivo por defecto que carga Ansible cuando importa un rol. Esto es, para cargar las tareas de nuestro rol de PHP, Ansible buscará el archivo de tareas a ejecutar en la ruta `roles/ansibleunir.php/tasks/main.yml`. Se puede trabajar directamente en este archivo o se pueden crear archivos diferentes en el mismo directorio e incluirlos desde el fichero `main.yml`. La siguiente imagen muestra un ejemplo de un árbol de ficheros donde `extensions.yml` y `php.yml` están en el mismo directorio que `main.yml`.

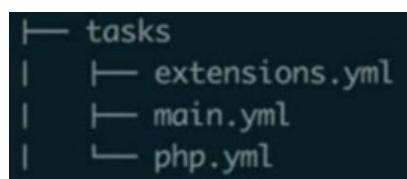


Figura 3. Directorio *tasks* de un rol con varios archivos de tareas. Fuente: elaboración propia.

Ahora cuentas con una clara separación de las tareas que instalan los paquetes PHP principales y de las tareas que instalan las extensiones, pero necesitas indicarle a Ansible la existencia de estos archivos, y lo harás desde el archivo `main.yml` que quedará de la siguiente manera:

```
---  
- include: 'php.yml'  
- include: 'extensions.yml'
```

Esto utiliza la sintaxis `include` de YAML, que incorpora un archivo YAML dentro de otro. Cuando Ansible se ejecuta, todos estos archivos se fusionarán, pero mientras estés desarrollando el *playbook*, conseguirás una clara separación de las responsabilidades.

A partir de la versión 2.0 de Ansible se introdujeron los `includes` dinámicos, que no se fusionan en el preprocesador (justo antes de la ejecución), sino durante la ejecución misma. Esto genera ciertos cambios de comportamiento, ya que se desconoce lo que realmente se va a incluir y ejecutar hasta que efectivamente se incluye durante la ejecución.

Para evitar que los cambios de comportamiento afectaran a *playbooks* ya existentes, en Ansible 2.1 se establecieron una serie de reglas para los `includes` dinámicos, para no romper la compatibilidad hacia atrás, aparte de añadir la opción `static` que permite indicar explícitamente cómo queremos que se comporte el `include`. Si queremos asegurarnos de incluir un fichero dinámicamente, debemos incluir lo siguiente:

```
- include: 'php.yml'  
  static: no
```

## 8.4. Separación del *playbook* de WordPress

Vamos ahora a separar el *playbook* monolítico de WordPress en varios roles únicos. Lo primero que vamos a hacer es crear los roles que se van a necesitar, para lo cual

nos situaremos en el directorio **roles**, y a continuación ejecutaremos los comandos siguientes para generar todos los roles vacíos que necesitamos:

```
ansible-galaxy init ansibleunir.nginx
ansible-galaxy init ansibleunir.mysql
ansible-galaxy init ansibleunir.wordpress
```

Una vez creados estos roles, debemos actualizar el archivo `playbook.yml` para que los incluya. La lista de roles se añade antes de la sección de tareas, con lo que el inicio del `playbook.yml` se parecerá al siguiente (si habías incluido el rol `git` anteriormente, es el momento de suprimirlo):

```
---
- hosts: all
  become: true
  roles:
    - ansibleunir.php
    - ansibleunir.nginx
    - ansibleunir.mysql
    - ansibleunir.wordpress
  tasks:
    - name: Install required tools
[...]
```

Los nuevos roles están todavía vacíos, por lo que, si ejecutamos ahora Ansible, no se realizará ninguna acción adicional relacionada con ellos, ni tampoco fallará la ejecución. Los roles vacíos se pueden incluir sin peligro en *playbooks*, ya que Ansible buscará las tareas definidas en ellos, y no hará nada al no encontrar ninguna definida. Se han incluido ya los roles vacíos para que cuando comencemos en la siguiente sección a mover las tareas desde el *playbook* a su rol correspondiente, Ansible las incluya automáticamente en la lista de tareas que ejecutar de forma transparente. Antes de continuar, deberías asegurarte de que el *playbook* está formateado y funciona correctamente, ejecutando `vagrant provision`. La ejecución debería terminar con éxito, ya que no suprimimos ninguna tarea. A continuación, vamos a ir moviendo las tareas de `playbook.yml` a los diferentes roles que hemos creado.

## ansibleunir.php

Vamos a empezar rellenando el rol `ansibleunir.php`. Las siguientes dos tareas que aparecen en el *playbook* están relacionadas con la instalación de PHP, por lo que las vamos a mover a `roles/ansibleunir.php/tasks/main.yml`, que contendrá entonces lo siguiente:

```
---
- name: Install PHP
  apt:
    state: present
    update_cache: yes
    name:
      - php
      - php-fpm
      - php-mysql
      - php-xml
- name: Remove apache2
  apt: name=apache2 state=absent
```

Al haber movido estas tareas, no deben ya existir en el *playbook*. Se incorpora en su lugar el rol `ansibleunir.php` bajo la sección de roles para que sea incluido en la lista de tareas que debe ejecutar Ansible. Una vez guardados tanto el rol como el *playbook*, puedes volver a ejecutar `vagrant provision`. Estas tareas se ejecutarán, al igual que lo hacían cuando estaban en el archivo `playbook.yml`. Sin embargo, ahora se puede ver que las tareas PHP tienen una cabecera diferente:

```
TASK [ansibleunir.php: Install PHP] *****
ok: [default]

TASK [ansibleunir.php: Remove apache2] *****
ok: [default]
```

Como se puede ver, ahora el nombre de la tarea es precedido por el nombre del rol, como en el caso de `ansibleunir.php: Install PHP`. Esto hace que sea fácil identificar

dónde están siendo incluidas las tareas cuando se ejecuta Ansible. Separando la parte de PHP de la instalación en un rol, hemos conseguido que la instalación de PHP sea reutilizable. Si en el futuro necesitásemos PHP en una máquina en algún otro *playbook*, podríamos añadir `ansibleunir.php` a la lista de roles a ejecutar y esto instalará todos los paquetes relevantes.

Seguiremos realizando la misma operación de refactorización para cada tarea en el *playbook* hasta que la sección tasks se quede solo con dos tareas: una que haga ping a tu máquina y otra que instale herramientas comunes.

### `ansibleunir.mysql`

El siguiente grupo de tareas que hay que mover son del rol `ansibleunir.mysql`. Tienes por tanto que abrir el archivo `roles/ansibleunir.mysql/tasks/main.yml` y mover todas las tareas relacionadas con MySQL del fichero `playbook.yml` al fichero de tareas del rol. Se trata de todas las tareas comprendidas entre las llamadas `Install MySQL` y `Create my.cnf`, ambas inclusive.

Recordemos en este punto que las tareas MySQL usaban una plantilla para llenar `my.cnf`, que tendremos asimismo que mover para que también forme parte del nuevo rol. Debes por tanto mover el archivo que se encuentra en la ruta `provisioning/templates/mysql/my.cnf` a `roles/ansibleunir.mysql/templates`. Esto puede hacerse mediante:

```
Mv provisioning/templates/mysql/my.cnf  
provisioning/roles/ansibleunir.mysql/templates
```

Para finalizar la operación, tienes que hacer un cambio en la tarea de plantilla del rol `ansibleunir.mysql`. El parámetro `src` de la tarea de plantilla es actualmente `templates/mysql/my.cnf`, pero dado que ahora la plantilla forma parte de un rol, esta ruta no es válida. Ansible automáticamente buscará en el directorio `templates` cuando se utiliza el módulo `template` dentro de un rol, por tanto, simplemente

tenemos que cambiar el parámetro `src` del módulo de plantilla al valor `my.cnf`. Una vez que hemos hecho este cambio, la tarea resultante en `roles/ansibleunir.mysql/tasks/main.yml` debería quedar como sigue:

```
- name: Create my.cnf
  template: src=mysq... dest=/root/.my.cnf
  when: mysql_new_root_pass.changed
```

Ahora puedes ejecutar `vagrant provision` de nuevo para asegurar que el nuevo rol funciona adecuadamente. Todas las tareas de MySQL deberían estar prefijadas ahora con `ansibleunir.mysql` en la salida.

Ya estamos a mitad de camino de la refactorización tras haber migrado PHP y MySQL, y ya hemos visto el patrón de cambios que se deben hacer para mover tareas de un *playbook* a un rol independiente. La migración de los roles restantes que instalan NginX y WordPress siguen los mismos pasos, tal como veremos a continuación.

## ansibleunir.nginx

Solo tenemos tres tareas para NginX: la de instalación de paquetes, la encargada de asegurar que NginX se ejecuta y la que copia la plantilla de configuración. Mueve ahora estas tareas de `playbook.yml` a `roles/ansibleunir.nginx/tasks/main.yml`. Una de estas tareas utiliza el módulo de plantilla (*template*). Como hicimos con el rol MySQL, tendremos que mover esta plantilla de modo que se aloje dentro del nuevo rol `ansibleunir.nginx` ejecutando:

```
mv provisioning/templates/nginx/default
provisioning/roles/ansibleunir.nginx/templates
```

Ahora corrige la tarea `Create nginx config` de modo que el campo `src` no empiece por la ruta del directorio. Es decir, debería quedar solo el `default`:

```
- name: Create nginx config
```

```
template: src=default dest=/etc/nginx/sites-available/default
notify: restart nginx
```

Esta tarea tenía también un manejador (handler) asociado, lo cual no habíamos encontrado todavía trabajando con roles. Así como se pueden definir tareas en un rol, pueden también definirse manejadores en el rol editando el fichero `handlers/main.yml`. Abre ahora `roles/ansibleunir.nginx/handlers/main.yml` y mueve a este archivo el manejador que se encuentra en `playbook.yml`. No se necesita la cabecera `handlers`, solo la definición del manejador propiamente dicha. Una vez movido, `handlers/main.yml` se verá así:

```
---
# handlers file for ansibleunir.nginx
- name: restart nginx
  service: name=nginx state=restarted
```

Ya puedes eliminar la cabecera `handlers` del `playbook.yml`, ya que debería estar ya vacía, y ejecuta nuevamente `vagrant provision` para probar el `playbook`. La ejecución debería completarse sin errores, quedando únicamente por migrar las tareas de WordPress.

## ansibleunir.wordpress

Este es el rol más complejo al contar con diez tareas diferentes, pero no es complicado de migrar. Se puede utilizar la misma estrategia que con los tres roles anteriores, migrándolo paso a paso hasta que se haya completado. Las tareas que debes mover van desde la que se llama `Copy wordpress.zip into tmp`, hasta la de `Import WordPress DB`. Mueve todas estas tareas, incluyendo estas dos de ambos extremos, del fichero `playbook.yml` a `roles/ansibleunir.wordpress/tasks/main.yml`.

Solo tenemos una tarea que usa el módulo de plantilla en este grupo, así que vamos a migrarla en primer lugar. Esto ya debería resultarte familiar. Primero tienes que mover el archivo `wp-config.php` para alojarlo dentro de su rol. Para ello, ejecuta:

```
mv provisioning/templates/wordpress/wp-config.php  
provisioning/roles/ansibleunir.wordpress/templates
```

Posteriormente, debes actualizar la tarea de plantilla; ahora el parámetro `src` solo debe contener `wp-config.php`, sin referencia al directorio:

```
- name: Create wp-config  
template: src=wp-config.php dest=/var/www/book.example.com/wp-  
config.php
```

Este rol no utiliza ningún manejador, por lo que no hay nada que migrar a este respecto. Sin embargo, hay otra cosa que tenemos que migrar. Tal como hemos hecho al mover los ficheros del módulo de plantilla, también tendremos que hacer lo propio con los archivos del módulo de copia. Los archivos utilizados por el módulo de plantilla se ubican en el directorio `templates`, y los archivos para el módulo de copia están situados en el directorio `files`. Hay dos tareas en `tasks/main.yml` que utilizan el módulo `copy`: una para copiar el archivo ZIP de WordPress y otra para copiar la copia de seguridad de la base de datos en el sistema. Lo primero es mover estos archivos para situarlos dentro de la ubicación correspondiente en el rol:

```
mv provisioning/files/wordpress.zip  
provisioning/roles/ansibleunir.wordpress/files  
mv provisioning/files/wp-database.sql  
provisioning/roles/ansibleunir.wordpress/files
```

También necesitas actualizar cada tarea `copy` para eliminar la referencia al directorio `files/` del parámetro `src`. Al igual que ocurre con las plantillas, Ansible busca los ficheros del módulo de copia directamente en su directorio correspondiente.

Una vez que hayas finalizado con estas operaciones, las tareas serán como se muestra a continuación:

```
- name: Copy wordpress.zip into tmp
  copy: src=wordpress.zip dest=/tmp/wordpress.zip
- name: Copy WordPress DB
  copy: src=wp-database.sql dest=/tmp/wp-database.sql
  when: db_exist.rc> 0
```

### [ansibleunir.common](#)

Llegados hasta aquí, podemos eliminar la tarea de ping, ya que únicamente servía para comprobar que Ansible lograba conectarse al *host*. En cuanto a la tarea que instala paquetes comunes, pueden guardarse tales tareas en otro rol. Vamos a crear este nuevo rol ahora ejecutando el comando `ansible-galaxy init ansibleunir.common` desde el directorio roles. Mueve a `tasks/main.yml` del nuevo rol la tarea que instala los paquetes comunes, y añade `ansibleunir.common` como primer rol de la lista de roles en `playbook.yml`. Dado que ahora la sección de tareas estará vacía, se puede suprimir perfectamente. ¡El *playbook* es mucho más pequeño ahora!

```
---
- hosts: all
  become: true
  roles:
    - ansibleunir.common
    - ansibleunir.php
    - ansibleunir.nginx
    - ansibleunir.mysql
    - ansibleunir.wordpress
```

Asegúrate de ejecutar una vez más `vagrant provision` para asegurarte de que todo sigue funcionando. Una vez hecho esto, ¡enhorabuena! ya has terminado de refactorizar el *playbook* en cinco roles reutilizables.

A partir de ahora, si necesitas PHP en algún otro proyecto, simplemente tienes que incluir `ansibleunir.php` en tu *playbook*. Si necesitas una base de datos, puedes incluir `ansibleunir.mysql`. Los roles son un mecanismo muy potente para encapsular tu lógica y, al mismo tiempo, acceder fácilmente si lo necesitas.

Como hemos visto, la creación de un rol de un *playbook* no es un procedimiento complicado. En esencia, es completamente mecánico. Solo has de seguir estos pasos:

1. Mueve las tareas al archivo `tasks/main.yml`.
2. Mueve los manejadores al archivo `handlers/main.yml`.
3. Si alguna tarea utiliza el módulo de plantilla, mueve los archivos necesarios al directorio de plantillas y cambia el atributo `src` del módulo para que sea relativo al directorio `templates`; por ejemplo, `src=tools/my-tool` buscaría la plantilla en: `roles/your.role/templates/tools/my-tool`.
4. Si alguna tarea utiliza el módulo de copia, mueve los archivos necesarios al directorio `files` y cambia el atributo `src` del módulo para que sea ahora relativo al directorio `files`; por ejemplo, `src=tools/my-tool` buscaría el archivo en: `roles/your.role/files/tools/my-tool`.
5. Mueve cualquier variable utilizada en el rol (en tareas o en plantillas) a `defaults/main.yml` (esto no fue necesario en nuestro caso).

## 8.5. Dependencias de roles

Tal como ha quedado, el *playbook* ejecuta e instala todas las dependencias necesarias para configurar la aplicación WordPress. Esto es debido a que hemos especificado todos los roles que instalan los prerequisitos en la lista de roles que deben ejecutarse. Esto es adecuado solo si somos nosotros mismos los que lo utilizamos, pero si lo utiliza alguien que no conoce las dependencias necesarias, puede omitir alguno de los roles necesarios cuando trate de usarlo.

Para evitar esto, existe la opción de dependencias en `meta/main.yml` que fue mencionada al explicar la estructura de directorios de un rol. Pues bien, puedes usarla para especificar las dependencias de un rol y hacer que Ansible las incluya de forma automática, en lugar de tenerlo que incluir tú en el *playbook*. Abre el archivo `roles/ansibleunir.wordpress/meta/main.yml` y borra todo el contenido (toda la información que contiene es opcional, así que lo podemos suprimir sin problemas) para introducir solo la información que necesitamos. Añade el siguiente contenido:

`dependencies:`

- `ansibleunir.common`
- `ansibleunir.php`
- `ansibleunir.mysql`
- `ansibleunir.nginx`

Esto es la lista de roles requeridos para ejecutar este rol. Después, modifica `playbook.yml` para que solo `ansibleunir.wordpress` esté en la lista de roles. Si ejecutas nuevamente `vagrant provision`, podrás apreciar que todas las dependencias se ejecutan antes de que se ejecute el rol `ansibleunir.wordpress`. Ansible analiza los metadatos de cada rol que encuentra y se asegura de que cualquier dependencia puesta en la lista se ejecute antes que el rol. Esto permite a quien lo use no tener que saber cuáles son las dependencias de nuestro rol; basta con que lo incluya en la lista de roles que desea ejecutar y Ansible ejecutará las dependencias de forma recursiva automáticamente antes que el propio rol que se incluyó.

## Roles de envoltorio

Cuando un rol se implementa para poder ser utilizado en diferentes situaciones, tiende a ser muy configurable. Pero, en ocasiones, las posibilidades de configuración de un rol hacen que sea más difícil de gestionar y utilizar que lo que cabría esperar. **Un rol de envoltorio** es un patrón que puede usarse para abstraer parte de esta configuración.

Al envolver un rol dentro de otro rol, se puede entender la intención de cómo un rol debería ser utilizado desde otro rol. Imagina que tienes un rol para saludar. Este rol se llama `ansibleunir.hola` y contiene los siguientes dos archivos:

```
# defaults/main.yml
---
your_name: World

# tasks/main.yml:
---
- name: Say hello
  debug: msg="Hello {{your_name}}"
```

Al ejecutar esto, se obtendrá la siguiente salida:

```
TASK [ansibleunir.hello: Say hello] ****
ok: [default] => {
  "msg": "Hello World"
}
```

Esto es debido a que tiene un valor por defecto Mundo para la variable `your_name`. Si quisieras cambiar el nombre utilizado, podrías poner una variable que lo sobrescribiera en tu *playbook*. Sin embargo, si siempre quisieras que dijera «Hola Unir», por ejemplo, el tener que proporcionar el nombre en cada *playbook* que incluyera este sería algo engoroso.

La alternativa más adecuada es la de envolver este rol en otro que contenga únicamente las variables que quieras poner. Para hacer esto, crea un rol llamado `ansibleunir.hola_unir` que no contenga ninguna tarea, sino que solo especifique `ansibleunir.hola` como una dependencia:

```
# meta/main.yml
dependencies:
- role: ansibleunir.hola
  vars:
```

```
your_name: Unir
```

Si ejecutas ahora vagrant provision:

```
TASK [ansibleunir.hello: Say hello] ****
ok: [default] => {
    "msg": "Hello Unir"
}
```

Esta es una manera muy conveniente de utilizar una configuración personalizada para un rol y codificarlo de forma que pueda guardarse en el control de versiones y usarse tantas veces quieras. Hemos visto este patrón mediante un ejemplo simple, pero se puede extraer a un rol MySQL, por ejemplo, para especificar los nombres de usuario y contraseñas de la base de datos que utilizamos habitualmente. Cuando necesites una base de datos específica en una máquina, solo tendrías que incluir el rol `ansibleunir.mysql_database`.

## 8.6. Creación de roles para distintas plataformas

No todas las plataformas disponen de los mismos comandos y utilidades. Utilizar el módulo `apt` para instalar cualquier paquete ha hecho nuestro trabajo más fácil, pero si tratamos de usar nuestro rol en otra plataforma, esto no funcionará. Siempre que sea posible, es recomendable evitar la mezcla de distintas plataformas en un mismo despliegue.

En esta situación hay que hacer algo más de trabajo. Apache2 es un ejemplo muy adecuado que podremos usar. Mientras que el paquete para Apache se llama `apache2` en sistemas basados en Debian, su nombre es `httpd` en sistemas basados en Redhat. En tal situación, se pueden crear tres archivos de tareas diferentes: `main.yml`, `install-debian.yml`, e `install-redhat.yml`. El fichero `main.yml` será el encargado de

incluir el fichero de variables adecuado y luego delegar la instalación en el *script* que corresponda.

```
# main.yml
---
- include_vars:"{{ansible_os_family}}.yml"
- include: install-debian.yml
  when: ansible_os_family =='Debian'
- include: install-redhat.yml
  when: ansible_os_family =='RedHat'
```

En vez de condicionar las tareas con *when*, se podía haber simplificado:

```
include:"install-{{ansible_os_family}}.yml"
```

Sin embargo, es preferible usar *when* en vez de incluir el fichero basado en la variable *ansible\_os\_family* para así ver claramente qué familias de sistemas operativos están soportados por el rol. También se incluye el fichero de variables correcto automáticamente. El fichero de variables define las mismas variables, pero asigna en cada caso diferentes valores.

```
# vars/Debian.yml
---
apache2_package_name: apache2

# vars/RedHat.yml
---
apache2_package_name: httpd
```

Posteriormente, en el *script* de instalación, se utiliza el módulo Ansible para el gestor de paquetes correspondiente según sea la familia de SO.

```
# tasks/install-debian.yml
---
- name: Install Apache
```

```

apt: name={{apache2_package_name}} state=present

# tasks/install-redhat.yml
---
- name: Install Apache
  yum: name={{apache2_package_name}} state=present

```

Este patrón es muy común, y es la forma aceptada generalmente para desarrollar un rol que funcione en múltiples familias de sistemas operativos.

Cabe también señalar que para estos casos existe en Ansible un módulo package ([https://docs.ansible.com/ansible/latest/collections/ansible/builtin/package\\_module.html](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/package_module.html)) que delega al gestor de paquetes correspondiente para cada sistema operativo. No se ha querido utilizar este módulo en el ejemplo para poder demostrar cómo utilizar módulos distintos según la familia del sistema operativo. Si lo escribieras tú mismo, lo deberías escribir como aparece debajo, y Ansible delegaría la labor al gestor de paquetes correspondiente para la familia de sistema operativo sobre el que se está ejecutando:

```

---
- name: Install Apache
  package: name={{apache2_package_name}} state=present

```

## 8.7. Trucos para escribir roles

Según Heap (2016), estas son las recomendaciones que hay que tener en cuenta a la hora de escribir roles:

- ▶ Cuando crees un rol, trata de asegurarte de que es utilizable sin necesidad de aportar nada más. Si tu rol instala un paquete específico de *software*, haz que se instale y configure una instalación básica sin intervención de usuario. Proporciona

puntos de extensión para los usuarios que quieran personalizar cosas más tarde, pero no les hagas proporcionar información de entrada. Utiliza `defaults/main.yml` para esto, ya que puede ser fácilmente sobreescrito.

- ▶ Escribe un rol sencillo que haga exactamente lo que debe hacer (y nada más!). Nos puede ocurrir que tratemos de hacer un rol tan genérico que lo compliquemos en exceso. La mayor ventaja que conseguirás de los roles es hacer justamente lo que hemos hecho en este tema: extraer la funcionalidad común en roles que pueden ser incluidos en otros *playbooks*.
- ▶ Hay dos clases de roles: rígidos y flexibles. Generalmente encontrarás roles flexibles en Ansible Galaxy, que han sido diseñados para ser reutilizados. Soportarán muchas variables diferentes y te dejarán usarlos como necesites.
- ▶ Cuando escribas roles tú mismo, o los encuentres en un escenario concreto (de un cliente, por ejemplo), verás que son completamente rígidos. Estos roles son específicos para un cliente, una aplicación particular, o un caso de uso de una aplicación. No necesitan puntos de extensión, por lo que, en vez de usar variables, se tiende a establecer los valores directamente en el rol. Se busca principalmente codificar los requisitos, y no tanto el hacer un rol reutilizable.

## 8.8. Módulos en Ansible

Los módulos, también denominados *plugins* de tareas (o *plugins* de biblioteca), son fragmentos de código que se pueden usar desde la línea de comandos o desde una tarea de un *playbook*. Ansible ejecuta cada módulo habitualmente en el nodo remoto destino de la ejecución y recolecta los valores de retorno.

Existen módulos disponibles en Ansible para las tareas más comunes de administración del sistema. Hay actualmente más de 500 módulos que se suministran

con Ansible, que es una cifra considerable si la comparamos con los 141 módulos que se suministraban con la versión de octubre de 2013.

No obstante, podemos escribir nuestro propio módulo (en bash o Python) y extender así la funcionalidad existente. Todos los módulos básicos de Ansible están desarrollados en Python. Hay dos grupos de módulos: `ansible-modules-core` y `ansible-modules-extras`.

### [ansible-modules-core](#)

Contiene todos los módulos básicos que se proporcionan con Ansible, tales como `apt`, `template` y `copy`, y es mantenido por el equipo principal de Ansible. Estos módulos son robustos y sufren importantes revisiones antes de incorporar cualquier nuevo cambio.

### [ansible-modules-extras](#)

Los módulos restantes se almacenan en este grupo, y aunque se distribuyen en la instalación estándar de Ansible, en realidad son mantenidos por la comunidad. El repositorio de módulos extra contiene algunos como `debconf`, `bundler` y `pagerduty`. Al contribuir con Ansible, si su cambio afecta a un módulo extra, el responsable de ese módulo es el responsable de revisar sus cambios, y no el equipo de Ansible.

### [¿Cuándo implementar un módulo?](#)

En ocasiones, no es una tarea fácil decidir si crear un rol o escribir un módulo. Si necesitas realizar varios pasos relacionados pero que no requieren realizar peticiones complejas a servicios externos, entonces un rol es la decisión adecuada. Un buen ejemplo de esto sería el de crear una base de datos, configurar los usuarios e importar un archivo SQL de ejemplo.

Si necesitas interactuar con una fuente de datos externa, como una API, entonces un módulo es la mejor opción, ya que tienes a tu disposición toda la potencia de un lenguaje de programación, en lugar de utilizar el módulo de comandos y curl.

Los módulos suelen estar diseñados para un cometido más concreto, aceptando una serie de variables que pueden utilizarse para configurar su comportamiento y permitir flexibilidad y reutilización.

## 8.9. Referencias bibliográficas

Heap, M. (2016). *Ansible: from Beginner to Pro*. Apress.

Hochstein, L. y Moser, R. (2014). *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. O'Reilly Media.

Herramientas de Automatización de Despliegues

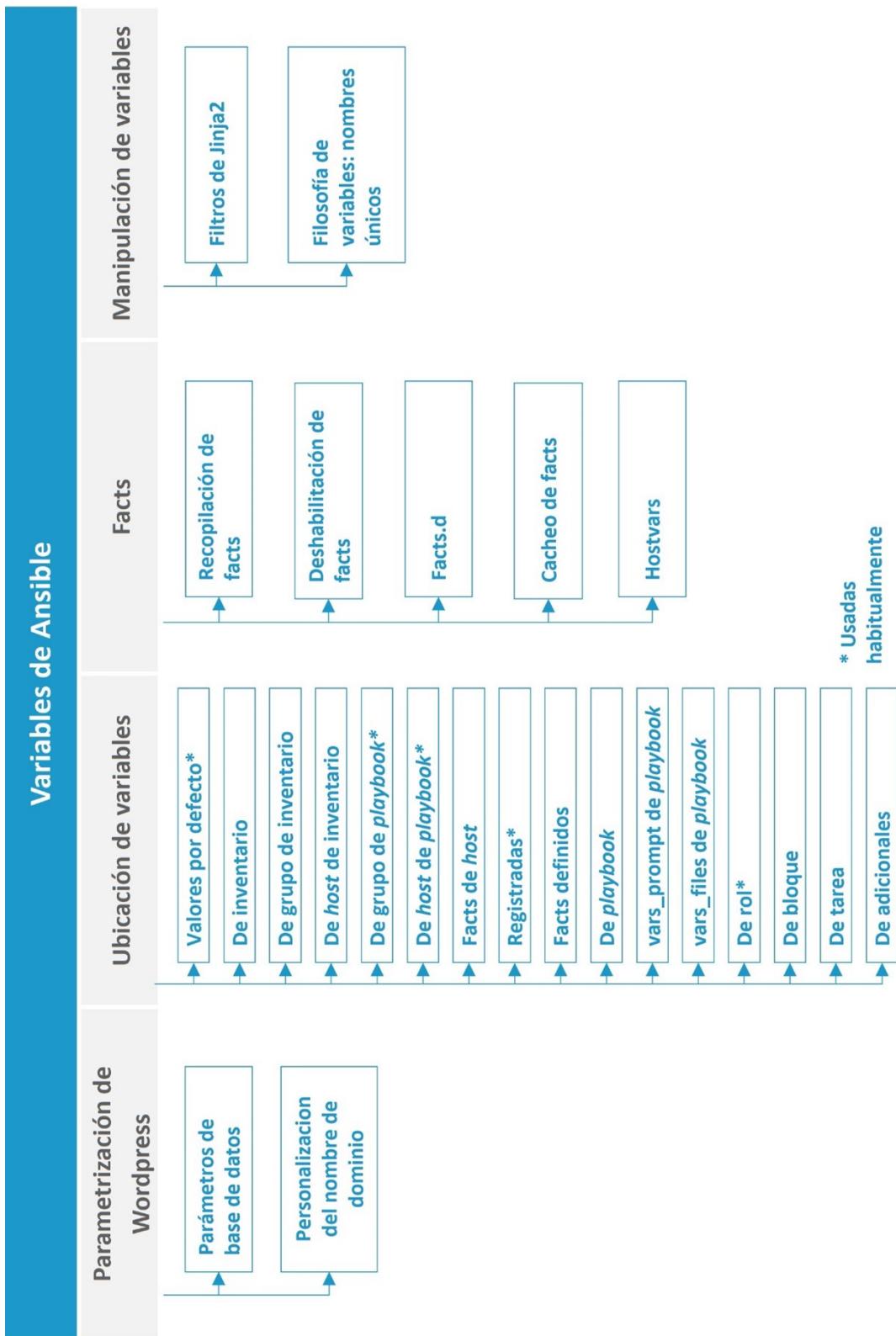
---

## VARIABLES DE ANSIBLE

# Índice

<b>Esquema</b>	<b>3</b>
<b>Ideas clave</b>	<b>4</b>
9.1. Introducción y objetivos	4
9.2. Parametrización del rol de WordPress	5
9.3. Ubicaciones de variables	11
9.4. Recopilación de <i>facts</i>	26
9.5. Manipulación de variables	30
9.6. Referencias bibliográficas	33

# Esquema



## 9.1. Introducción y objetivos

Las variables en Ansible nos ofrecen una gran flexibilidad, tanto a la hora de definirlas como a la hora de utilizarlas, bien sea en *playbooks* o en plantillas. Puedes utilizar variables para el **contenido** (por ejemplo, en una plantilla de fichero de configuración o para especificar una lista de paquetes para instalar en una tarea) o para controlar el flujo de ejecución, como una manera de decidir **qué acciones realizará tu playbook** (condicionando las tareas al valor de la variable).

Las variables en Ansible son siempre globales, lo que significa que, al declarar una variable en un rol, en un *playbook* o en cualquiera de las ubicaciones posibles (que veremos en este tema), se hace disponible para todos los *playbooks* y plantillas que se procesan durante la ejecución de Ansible. Esto tiene como consecuencia que las variables de un rol generalmente se prefijan con el nombre del rol.

Por ejemplo, si tuvieras en el rol PHP que hacer configurable la lista de paquetes para instalar, nombrarías a la variable `php_packages`, no solo `packages`.

Los objetivos que se pretenden conseguir en este tema son los siguientes:

- ▶ Aprender cómo se agrega el soporte de variables al rol de WordPress.
- ▶ Conocer las distintas ubicaciones que permite utilizar Ansible para definir variables y cómo hacer uso de ellas desde un *playbook*.

## 9.2. Parametrización del rol de WordPress

Vamos a hacer ahora nuestro rol de WordPress (el que vimos en el tema «Roles de Ansible») parametrizable para que podamos personalizar la instalación de una instancia. Tal como quedó en ese tema, la instalación que realiza siempre va a utilizar el mismo nombre de base de datos con las mismas credenciales, lo cual supone un riesgo de seguridad, aparte de que no permite instalar más de una instancia en el mismo entorno. Esto es debido a que tanto el nombre de base de datos como la contraseña están incluidos en el código (*hardcoded*) en las propias tareas que los requieren.

El *playbook* contiene actualmente:

```
---
- hosts: all
  become: true
  roles:
    - ansibleunir.wordpress
```

Además de especificar el nombre del rol que se va a incluir en el *playbook*, puedes aprovechar para especificar cualquier variable como parámetro que deseas utilizar en ese rol. Vamos ahora a actualizar el *playbook* para definir algunas variables y hacer así que sea más seguro. Para incluir el rol vamos a utilizar una sintaxis distinta, para poder indicarle a Ansible la entrada que indica el rol que debe incluir (añadiendo el prefijo `role:`). A continuación, incluiremos una sección `vars` para declarar las variables que se podrán utilizar en las tareas o plantillas del rol:

```
---
- hosts: all
  become: true
  roles:
    - role: ansibleunir.wordpress
  vars:
```

```
nombre_bd: mywordpressdb  
usuario_bd: mywordpressusr  
password_bd: Aproba2to2
```

Todas las variables que hemos incluido están relacionadas con la base de datos. Hay dos ficheros que deben actualizarse para utilizar estas nuevas variables sustituyendo los valores en el código:

- ▶ El fichero de tareas que crea el usuario y la base de datos.
- ▶ El fichero `wp-config`, que es el que lee WordPress para saber qué credenciales debe usar para acceder a la base de datos.

La manera de referenciar variables en Ansible es incluyendo su nombre entre llaves dobles, tal como `{{variable_name}}`, dado que es la sintaxis que define Jinja2 para la sustitución de variables. Ansible utiliza Jinja2 como lenguaje de plantillas, lo que permite disponer no solo de la sustitución de variables, sino de toda la funcionalidad de este motor de plantillas (hablaremos sobre él más adelante).

Vamos ahora a cambiar todas las ocurrencias del nombre de la base de datos, el usuario o la contraseña para que pasen a utilizar las variables que hemos definido para tal fin. Busca dónde se encuentran todas estas ocurrencias en el fichero `roles/ansibleunir.wordpress/tasks/main.yml` y sustituye cada una por la variable correspondiente. Los cambios se muestran resaltados en negrita:

```
- name: Create WordPress MySQL database  
  mysql_db: name="{{nombre_bd}}" state=present  
  
- name: Create WordPress MySQL user  
  mysql_user: name="{{usuario_bd}}" host=localhost password="{{password_bd}}"  
  priv="{{nombre_bd}}.*:ALL"  
  
- name: Create wp-config  
  template: src=wp-config.php dest=/var/www/book.example.com/wp-config.php  
  
- name: Does the database exist?  
  command: mysql -u root {{nombre_bd}} -e"SELECT ID FROM wordpress.wp_users  
LIMIT 1;"
```

```

register: db_exist
ignore_errors: true
changed_when: false
- name: Copy WordPress DB
  copy: src=files/wp-database.sql dest=/tmp/wp-database.sql
  when: db_exist.rc == 1
- name: Import WordPress DB
  mysql_db: target=/tmp/wp-database.sql state=import
  name="{{nombre_bd}}"
  when: db_exist.rc == 1

```

Si ejecutamos ahora `vagrant provision`, se crearán las bases de datos y usuarios con los nombres que hemos establecido a través de las variables. Una vez hecho esto, falta por actualizar el fichero `templates/wp-config.php` para también hacer referencia ahí a las variables. Tal como ya hemos indicado, la sintaxis con dobles llaves propia de Jinja también se utiliza en las plantillas de la misma forma que la hemos utilizado en el *playbook*:

```

/** The name of the database for WordPress */
define( 'DB_NAME', '{{nombre_bd}}');

/** MySQL database username */
define( 'DB_USER', '{{usuario_bd}}');

/** MySQL database password */
define( 'DB_PASSWORD', '{{password_bd}}');

```

Prueba a ejecutar `vagrant provision` después de realizar este cambio, e inicia sesión en la máquina virtual accediendo mediante `vagrant ssh` para ejecutar `cat /var/www/book.example.com/wp-config.php` y asegurarnos de que todos los valores establecidos a través de las variables están correctamente definidos. Una vez comprobado, ejecuta `exit` en la línea de comandos para cerrar la sesión de la máquina virtual.

Este ejemplo ha servido para demostrar cómo se pueden utilizar variables para hacer más seguro el despliegue mediante Ansible de una aplicación. Sin embargo, las variables tienen muchos otros usos. Actualizaremos a continuación el rol de Wordpress para que, aparte de las credenciales y nombre de la base de datos, se pueda también personalizar la ruta de instalación de la aplicación y el contenido inicial que se publicará en el sitio web.

## Personalización del nombre de dominio de WordPress

En este momento, la dirección URL en la que WordPress se ejecuta está codificada con el valor book.example.com en varios lugares. Esto también limita a que solo se pueda instalar una sola instancia de WordPress en un entorno. Ahora vamos a parametrizar el valor del dominio con una variable, sustituyendo el valor fijo que se encuentra a lo largo del propio código del rol. Vamos a editar el fichero playbook.yml para añadirle otra variable que le indique a Ansible el nombre de dominio que debe utilizar WordPress. La llamamos dominio\_wp y tendrá el valor book.example.com:

```
- role: ansibleunir.wordpress
  vars:
    nombre_bd: mywordpressdb
    usuario_bd: mywordpressusr
    password_bd: Aproba2to2
    dominio_wp: book.example.com
```

Una vez definida la variable, hemos de actualizar los ficheros donde estuviera el valor de dominio establecido directamente en código, para sustituirlo por la referencia a la variable. Si buscamos book.example.com a partir del directorio **roles** podemos encontrar que se usa en tres ficheros:

```
roles/ansibleunir.nginx/templates/default
roles/ansibleunir.wordpress/files/wp-database.sql
roles/ansibleunir.wordpress/tasks/main.yml
```

Comencemos por cambiar el fichero de configuración de NginX. Debemos modificar las ocurrencias de book.example.com y sustituirlas por la referencia a la variable dominio\_wp dentro del fichero default del directorio templates dentro del rol:

```
server_name {{dominio_wp}};  
root /var/www/{{dominio_wp}};
```

El siguiente fichero que debemos actualizar es wp-database.sql, que es bastante grande, por lo que es mejor que usemos «buscar y reemplazar» (*find & replace*), y sustituymos las ocurrencias que encontremos de book.example.com por la referencia a la variable {{dominio\_wp}}. La búsqueda debe encontrar unas ocho ocurrencias que debes cambiar.

Por último, es necesario actualizar el fichero de tareas. Lo mismo que hemos hecho con wp-database.sql lo vamos a hacer con este fichero, sustituyendo las ocurrencias de book.example.com por referencias a la variable {{dominio\_wp}}. Esta vez, la búsqueda deberá encontrar unos cuatro elementos a cambiar.

Una vez guardados todos estos cambios que hemos hecho, volvemos a ejecutar vagrant provision, que se completará con éxito sin reportar cambios. Todo lo que hemos hecho hasta ahora es cambiar cadenas codificadas por variables, no hemos cambiado sus valores.

Finalmente, vamos a especificar el título y el contenido predeterminados de la publicación. Nuevamente, editamos el *playbook* para añadir las variables que van a permitirnos esta personalización:

```
- role: ansibleunir.wordpress  
  vars:  
    nombre_bd: mywordpressdb  
    usuario_bd: mywordpressusr  
    password_bd: Aproba2to2  
    dominio_wp: book.example.com
```

```
titulo_inicial: Hola Hola
contenido_inicial: ">Este es un artículo de ejemplo. Cámbialo
por algo más interesante."
```

Tanto el título como el contenido del blog inicial están definidos en el fichero de base de datos que importamos desde el *playbook*, por lo que debemos hacer los cambios en el fichero wp-database.sql para sustituir los valores fijos que ahí se encuentran por referencias a las nuevas variables que hemos añadido. El blog inicial se titula **Hello world!**, así que buscaremos este texto en wp-database.sql y lo sustituiremos por la referencia a la variable: {{titulo\_inicial}}. Justamente antes del título veremos un campo con el contenido del artículo, que empieza por «**Welcome to WordPress**», que será lo que se muestre en el blog inicial. Para sustituirlo por la referencia a nuestra variable, debemos eliminar todo el contenido del párrafo, delimitado por los tags de HTML <p> y </p>, y reemplazarlo por {{contenido\_inicial}}.

Hay un último cambio que debemos realizar antes de dar nuestra tarea por finalizada, y es que el fichero wp-database.sql que hemos modificado se copiaba a la máquina remota mediante el módulo copy, ya que era un fichero normal y corriente, que no necesitaba ninguna transformación. El módulo copy no hace ningún tipo de procesamiento sobre el fichero, y simplemente lo copia al destino, y dado que ahora hemos incluido referencias a variables en el fichero, lo hemos convertido en una plantilla que debemos procesar a través del módulo template. Debemos, por tanto, cambiar la tarea que copia el fichero wp-database.sql y sustituir el módulo de copia por el de plantillas, modificando el fichero main.yml del directorio tasks del rol:

```
- name: Copy WordPress DB
  template: src=wp-database.sql dest=/tmp/wp-database.sql
  when: db_exist.rc == 1
```

Dado que ahora utilizamos el módulo template en vez de copy, y para que el módulo encuentre el fichero wp-database.sql a procesar, también es necesario moverlo de ubicación, del directorio files al directorio templates:

```
cd provisioning/roles/ansibleunir.wordpress  
mv files/wp-database.sql templates
```

Llegados a este punto, ya podemos volver a ejecutar el *playbook*, aunque como la base de datos ya está creada, el archivo modificado `wp-database.sql` no se va a importar. Dado que hace también un tiempo que no has eliminado y vuelto a construir la máquina virtual, es un buen momento para hacerlo y que se cree todo de nuevo.

Vamos a ejecutar desde el terminal, en el mismo directorio que el fichero de configuración Vagrant, el comando `vagrant destroy` y confirmamos el borrado, y a continuación ejecutamos `vagrant up`. Estos comandos destruirán y volverán a crear una nueva máquina virtual que se aprovisionará con Ansible. Puesto que se va a aprovisionar la máquina desde cero, instalando las herramientas, creando la base de datos, etc., la operación puede tardar varios minutos.

### 9.3. Ubicaciones de variables

Las variables no solo se pueden usar en *playbooks* o ficheros, sino que también se pueden definir en una infinidad de lugares diferentes. En la documentación de Ansible puedes encontrar todas las posibles ubicaciones de variables, así como su precedencia, a través de la cual podrás determinar las definiciones de variables que sobrescribirán el valor previo que la variable pudiera tener, en función de la ubicación en la que se ha declarado. Sin embargo, la documentación oficial no incluye una referencia de cuándo utilizar cada ubicación, por lo que aquí vamos a tratar de complementar dicha documentación repasando las posibles ubicaciones, junto con indicaciones sobre cuándo, según Heap (2016), es adecuado utilizarlas, y cuáles son más frecuentemente utilizadas.

Estos lugares están ordenados de menor a mayor precedencia, es decir, los valores por defecto de los roles tienen la precedencia más baja cuando se trata de establecer los valores de las variables y son sobrescritos por todo lo demás. Las variables de grupo de inventario sobrescriben los valores predeterminados de roles, pero son sobrescritos por el módulo `set_fact`.

### Valores por defecto de los roles (se usan habitualmente)

Se definen en el archivo `defaults/main.yml` dentro del rol. Estas variables tienen la precedencia más baja, por lo que una variable declarada en cualquier otra ubicación las sobrescribirá y, por ello, son muy adecuadas para establecer valores predeterminados. En `mi_rol/defaults/main.yml`:

```
nombre_usuario: Caracola
```

### Variables de inventario

Este tipo de variables ya las habíamos utilizado cuando creamos un fichero de inventario que empleamos con Ansible. La mayoría de las veces, en el fichero de inventario solo usarás variables específicas del inventario (como `ansible_user` o `ansible_ssh_private_key_file`), pero puedes establecer cualquier variable que quieras y solamente estará disponible para el *host* en la que la definas. Veamos un ejemplo:

```
192.168.33.33 nombre_usuario=Caracola
```

Estas variables también se pueden declarar fichero de inventario para un grupo o para un grupo de grupos, como se muestra a continuación:

```
[aplicacion]
192.168.33.33
192.168.33.34
```

```
[administracion]
192.168.33.100

[baseDatos]
192.168.33.99

[sitiosweb:children]
    aplicacion
    administracion

[aplicacion:vars]
    nombre_usuario=Caracola

[administracion:vars]
    nombre_usuario=Caracola

[baseDatos:vars]
    nombre_usuario=Pepe

[sitiosweb:vars]
    Version_php: 7
```

## Variables de grupo de inventario

Si queremos definir variables de grupo de inventario, debemos ubicar el propio fichero de inventario en un directorio. Crea un directorio llamado `inventory` y mueve el fichero de inventario ahí; Es decir, el fichero de inventario real se encontrará ahora en `inventory/inventory`.

El uso de este tipo de variables requiere de la creación de un subdirectorio denominado `group_vars`, que debe alojarse dentro del directorio `inventory`. Este subdirectorio puede contener ficheros de variables con el mismo nombre que el grupo en el que se quieren declarar. Dado el fichero `inventory/inventory`:

```
[aplicacion]
```

```
192.168.33.33
```

```
192.168.33.34
```

```
[administracion]
```

```
192.168.33.100
```

```
[basedatos]
```

```
192.168.33.99
```

Para poder definir variables de grupo para estos grupos que están definidos arriba, debemos crear la siguiente estructura de directorios y ficheros:

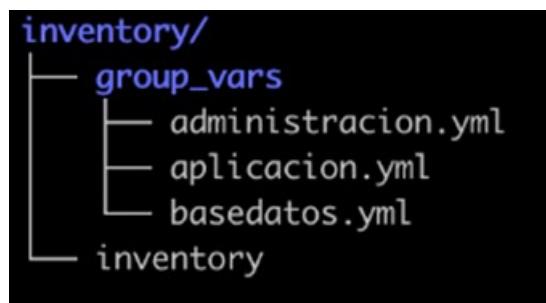


Figura 1. Ejemplo de directorios y ficheros para las variables de grupo de inventario. Fuente: elaboración propia.

En este caso, para definir una variable asociada al grupo de inventario de `basedatos`, debemos incluirla en el fichero `basedatos.yml` dentro del directorio `group_vars`.

## Variables de *host* de inventario

Estas variables son semejantes a las de grupo de inventario, salvo que en este caso se declaran a nivel de *host*. Tomando el mismo ejemplo de inventario:

```
[aplicacion]
```

```
192.168.33.33
```

```
192.168.33.34
```

```
[administracion]
```

```
192.168.33.100
```

```
[basedatos]
```

192.168.33.99

La imagen que se muestra a continuación contiene la estructura de directorios y ficheros que nos permitiría definir variables de *host* de inventario:

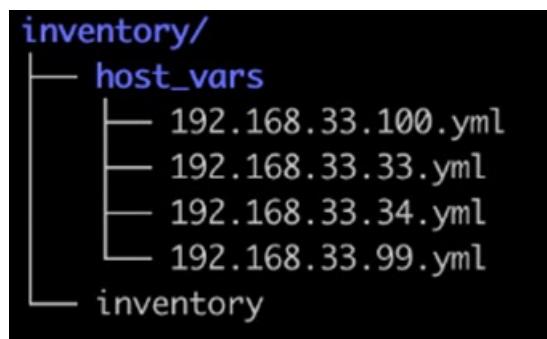


Figura 2. Ejemplo de directorios y ficheros para las variables de *host* de inventario. Fuente: elaboración propia.

Una variable que definamos en 192.168.33.33.yml estará únicamente disponible en el *host* 192.168.33.33.

Para ello, crea el fichero 192.168.33.33.yml dentro del subdirectorio host\_vars con esta variable:

```
nombre_usuario: Caracola
```

Esto es lo mismo que definir la variable en el fichero de inventario, en la propia declaración del *host*: 192.168.33.33 nombre\_usuario=Caracola. La única diferencia es que utilizar la definición el fichero propio de variables del *host* tiene mayor precedencia.

### Variables de grupo de *playbook* (se usan habitualmente)

Otra ubicación donde se pueden definir variables de grupo es en el subdirectorio group\_vars pero situado a nivel de *playbook*. Su funcionalidad es la misma que las definidas a nivel inventario, pero esta vez el subdirectorio cuelga del mismo

directorio en el que se encuentra el fichero `playbook.yml`, y su precedencia es también superior.

### Variables de *host* de *playbook* (se usan habitualmente)

Análogamente a las variables de grupo, las de *host* también pueden definirse al mismo nivel que el *playbook*. Nuevamente tienen la misma funcionalidad que las definidas a nivel inventario, pero esta vez el subdirectorio `host_vars` se encuentra ubicado en el mismo directorio en el que se encuentra el fichero `playbook.yml`, y su precedencia es también superior.

### Facts de *host*

El concepto de *fact* ('hecho') no es solo propio de Ansible, sino que lo usan otras muchas herramientas de gestión de la configuración. Un *fact* corresponde a un dato de la máquina que se está gestionando. Existen *facts* representando una variada y extensa cantidad de información de la máquina, tal como la dirección IP del *host*, el sistema operativo que ejecuta, la versión del sistema operativo, e incluso la memoria disponible en el sistema. Estos *facts* están disponibles como variables en Ansible, con la misma funcionalidad que cualquier otra variable.

Al ejecutar Ansible, el primer módulo que se procesa es el de configuración (`setup`), el cual es el encargado de recopilar los *facts* de la máquina que se gestiona. Si defines *facts* con los mismos nombres que los valores predeterminados de roles o las variables de grupo o de *host*, se sobrescribirán con los *facts* de la máquina (ya que los *facts* de la máquina tienen una prioridad más alta). Si por el contrario lo que haces es registrar una variable mediante la opción `register`, o utilizar el módulo `set_fact` asignando el valor a un nombre que coincide con el *fact* de *host*, este último quedará sobrescrito.

La variable `ansible_all_ipv4_addresses` es un ejemplo de *fact* de *host*, y contiene una lista con todas las direcciones IP versión 4 de la máquina. A todos los *facts* de *host* se les añade el prefijo “`ansible_`”, por lo que se hace complicado sobrescribirlos por accidente.

Para conocer todos los *facts* que se recopilan en una máquina, se puede simplemente ejecutar el módulo de configuración utilizando un fichero de inventario con la máquina, tal como:

```
ansible all -i fichero_de_inventario -m setup
```

### Variables registradas (se usan habitualmente)

Cuando trabajas dentro de un *playbook*, puedes guardar la salida de los módulos para usarla más adelante. Por ejemplo, para almacenar en la variable `hosts_info` toda la información del sistema de archivos sobre el fichero `/etc/hosts`, utiliza el siguiente fragmento de tarea:

```
- stat: path=/etc/hosts  
  register: hosts_info  
- debug: var=hosts_info
```

En caso de que la variable `hosts_info` estuviera definida en cualquier otra ubicación con una precedencia más baja que la de las variables registradas, se sobrescribiría su valor. Esto podría llegar a provocar errores difíciles de detectar, dado que la variable tenía un valor hasta ejecutar esta tarea, pero una vez ejecutada el valor ahora es diferente. El uso de prefijos en las variables puede ayudar a evitar esto.

## Facts definidos

Hay un tipo de *facts*, diferentes a los habituales de *host*, que pueden ser definidos por el usuario en un *playbook* para poderlos utilizar posteriormente. Veamos el siguiente ejemplo:

```
---
- hosts: all
  tasks:
    - set_fact: ejemplo_fact="Hola mundo"
    - debug: var= ejemplo_fact
```

En este ejemplo sencillo del uso del módulo `set_fact` hemos definido el *fact* de nombre `ejemplo_fact` con el valor “Hola mundo”, mientras que en la siguiente tarea hacemos referencia al *fact* que acabamos de crear para mostrar su valor en la salida.

En este otro ejemplo, en el que también se utiliza una función Jinja para la manipulación de variables, tomando la ruta de la salida del módulo `stat` y convirtiéndola en mayúsculas:

```
---
- hosts: all
  tasks:
    - stat: path=/etc/hosts
      register: info_maquina
    - set_fact: ejemplo_fact = "{{ info_maquina.stat.path|upper}}"
    - debug: var= ejemplo_fact
```

## Variables de *playbook*

También se pueden definir variables directamente en un *playbook* si lo que quieres es sobrescribir el valor de algunas variables cuando se incluyen roles o si únicamente quieres escribir un pequeño *playbook* y prefieres mantener todo en el mismo fichero y minimizar así el número total de ficheros que se necesitan.

Este tipo de variables se declaran en su propia sección vars, que se encuentra al mismo nivel que las de tareas:

```
---
- hosts: all
  gather_facts: false
  vars:
    nombre_usuario: Caracola
  tasks:
    - debug: msg="Hello {{nombre_usuario}}"
```

### Variables vars\_prompt de playbook

En ocasiones puede ser necesario obtener información que deba proporcionar el usuario en tiempo de ejecución, por tratarse de información sensible, como puede ser una contraseña o un dato que solo él conoce, como un usuario de acceso.

Se puede recopilar esta información de usuario final especificando una sección de vars\_prompt en el *playbook*. Una vez que ejecutes el *playbook*, Ansible mostrará por la consola la pregunta especificada y almacenará las respuestas como valor de esta variable, que se podrá utilizar a lo largo del *playbook* como cualquier otro tipo de variables. Veamos un ejemplo:

```
-- 
hosts: all
vars_prompt:
- name: nombre_usuario prompt: "Cómo te llamas?"
tasks:
- debug: msg="Hola {{nombre_usuario}}"
```

Cabe destacar que cuando Ansible nos solicita un valor para una de estas variables, no muestra los caracteres a medida que los tecleamos. Esto es así por si se da el caso de que se está introduciendo información confidencial. De lo contrario, estaría disponible y accesible para quien revisara el historial.

```
$ ansible-playbook -i /path/to/inventory playbook.yml
```

Cómo te llamas?:

```
PLAY ****
```

```
TASK [debug] ****
ok: [localhost] => {
    "msg": "Hola Fulanito"
}
```

## Variables `vars_files` de *playbook*

Ansible leerá si los hubiere los ficheros disponibles en `group_vars` y en `host_vars`, pero existe además la posibilidad de especificar otros ficheros de variables adicionales añadiendo en el *playbook* la sección `vars_files`:

```
---
- hosts: all
  vars_files:
    - mas_variables.yml
  tasks:
    - debug: msg="Hola {{ nombre_usuario }} {{ apellido_usuario }}"
```

Al ejecutar esto, Ansible leerá el fichero `mas_variables.yml` ubicado en el mismo directorio que el *playbook*. Este fichero tiene el mismo formato que los otros ficheros de variables `group_vars` y `host_vars`.

Se puede especificar una lista estática de ficheros de variables, aunque la potencia real de `vars_files` se ve claramente cuando se combina con otras variables, dado que puedes utilizar los valores de variables tales como `ansible_os_family` para incluir un fichero concreto de variables según su valor:

```
---
- hosts: all
  vars_files:
```

```
- "{{ansible_os_family}}.yml"
```

Esto lo interpretará Ansible con el valor que tenga en cada máquina, como puede ser Redhat.yml o Debian.yml, lo cual posibilita la selección dinámica de un fichero de variables en función de la familia del sistema operativo que se esté ejecutando en cada máquina.

También se podría por ejemplo leer mediante vars\_prompt la entrada del usuario por consola y utilizar su valor en vars\_files para leer el fichero de variables que haya indicado el propio usuario:

```
---
- hosts: all
  vars_prompt:
    - name: fichero_vars
      prompt:"Qué fichero de variables quieras incluir?"
  vars_files:
    - "{{ fichero_vars }}.yml"
  tasks:
    - debug: msg="Hola {{ nombre_usuario }} {{ apellido_usuario }}"
```

Este *playbook* solicitará al usuario el fichero de variables a incluir para utilizarlo como nombre de fichero en vars\_files, aunque si el nombre de fichero especificado no existiera, Ansible devolverá un error y no continuará la ejecución. Vamos a ver un ejemplo del resultado introduciendo el valor «fichero-invalido»:

```
Qué fichero de variables quieras incluir?:
ERROR! vars file fichero-invalido.yml was not found
```

Para evitar este error, puedes utilizar la funcionalidad de vars\_files que permite proporcionar una lista de ficheros, que Ansible recorrerá e incluirá el primero que encuentre, lo cual es muy útil en este caso de vars\_files basado en una entrada de usuario, o también cuando se basa en otro tipo de variables. Si se especifica un

nombre de fichero existente, lo usará, pero si no lo encuentra se utilizará `usuario_default` en lugar de devolver un error:

```
---
- hosts: all
  vars_prompt:
    - name: fichero_vars
      prompt:" Qué fichero de variables quieres incluir?"
  vars_files:
    - ["{{ fichero_vars }}.yml","usuario_default.yml"]
  tasks:
    - debug: msg="Hola {{ nombre_usuario }} {{ apellido_usuario }}"
```

Este es el listado de ficheros para el ejemplo:

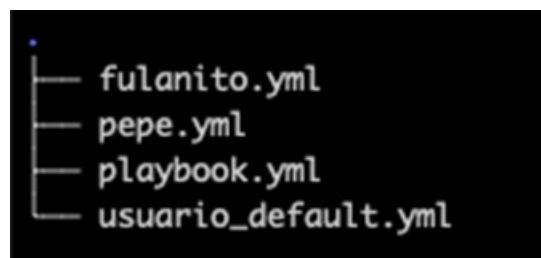


Figura 3. Ejemplo de ficheros para las variables de fichero. Fuente: elaboración propia.

Si introducimos en el prompt el nombre `pepe` al ejecutar el `playbook`, en la salida aparecerá el saludo a Pepe, al utilizar el fichero de variables `pepe.yml` que existe y contiene las variables correspondientes:

```
TASK [debug] ****
ok: [localhost] => {
  "msg": "Hola Pepe Perez"
}
```

Si por el contrario utilizamos el nombre `paco` en el prompt, al no existir el fichero `paco.yml`, se buscará el siguiente fichero de la lista, en este caso `usuario_default.yml`, que sí que existe, y se leerán sus variables:

```
TASK [debug] *****
ok: [localhost] => {
"msg": "Hola quienquiera que seas"
}
```

Combinando estos dos tipos de variables, `vars_prompt` y `vars_files`, podemos contar con distintos ficheros de variables y mantenerlos todos en el sistema de control de versiones, permitiendo posteriormente que el usuario elija en tiempo de ejecución qué configuración usar en cada momento.

## Variables de rol (se usan habitualmente)

Cuando utilizas un rol en tu *playbook*, se pueden especificar las variables que quieras emplear en el rol, como si le pasaras parámetros. Este tipo de variables ya lo hemos utilizado cuando hemos establecido los valores a utilizar para las variables necesarias en nuestro rol de WordPress:

```
---
- hosts: all
  become: true
  roles:
    - role: ansibleunir.wordpress
      vars:
        nombre_bd: mywordpressdb
        usuario_bd: mywordpressusr
        password_bd: Aproba2to2
```

## Variables de bloque

En Ansible se define bloque como un conjunto de tareas. El empleo de bloques permite manejar de una manera más cómoda los errores o los ajustes de un determinado grupo de tareas:

```
- hosts: all
```

```

tasks:
- apt: name=apache2 state=installed
  become: true
  when: resultado_tarea.rc == 0
- copy: content="Fichero ejemplo" dest=/var/www/hola.html
  become: true
  when: resultado_tarea.rc == 0

```

Si utilizásemos un bloque, podríamos especificar una única vez las opciones comunes de estas tareas, en este caso `become` y `when`:

```

- hosts: all
  tasks:
    - block:
      - apt: name=apache2 state=present
      - copy: content="Fichero ejemplo" dest=/var/www/hola.html
        become: true
      when: resultado_tarea.rc == 0

```

De la misma manera, se puede asociar una sección `vars` al bloque para definir ahí las variables comunes del conjunto de tareas.

## Variables de tarea

El siguiente tipo de variables en orden ascendente de precedencia son las que especificamos a nivel tarea:

```

---
- hosts: all
  tasks:
    - debug: msg="Hola {{nombre_usuario}}"
      vars:
        nombre_usuario: Pepe

```

Esto no parece demasiado útil, dado que podríamos utilizar directamente el valor mismo en la tarea. Pero si la tarea utilizara un mismo valor repetidas veces, podría

sernos útil definirlo como una variable para especificarlo una única vez y referenciarlo cada vez que sea necesario en la tarea. Un ejemplo de esto podría ser el paquete Apache2.

En los *hosts* basados en Debian, la configuración de Apache2 se encuentra en el fichero /etc/apache2/apache2.conf, mientras que en los *hosts* basados en RedHat, la configuración de Apache2 está en el fichero /etc/httpd/httpd.conf. En vez de tener que especificar varias veces apache2 o httpd, se puede utilizar en este caso una variable de tarea:

```
---
- hosts: all
  tasks:
    - template: src=webserver.conf dest="/etc/{{ nombre }}/{{ nombre }}.conf"
  vars:
    nombre: apache2
```

## Variables adicionales

Estas variables son las que especificamos como parámetro al ejecutar Ansible y son el tipo de variables que tienen la precedencia más alta, por lo que sobrescribirán a cualquier otra variable con el mismo nombre que se haya definido de cualquier otra forma:

```
ansible-playbook -i fichero_inventario playbook.yml -e
'nombre_usuario=Menganito'
```

Las variables adicionales se establecen con el parámetro `-e` al ejecutar ansible o ansible-playbook en la línea de comandos. Se pueden especificar múltiples indicadores `-e` para establecer tantas variables como se desee:

```
ansible-playbook -i fichero_inventario playbook.yml -e  
'nombre_usuario=Menganito' -e 'mi_nombre=Pepe'
```

También se pueden especificar las variables adicionales con el formato JSON:

```
ansible-playbook -i fichero_inventario playbook.yml -e  
'{"nombre_usuario":"Menganito", "mi_nombre":"Pepe"}'
```

Aunque, en caso de querer especificar un número considerable de variables adicionales, lo mejor es indicar un fichero como parámetro, tal como se indica a continuación, y definirlas ahí todas:

```
ansible-playbook -i fichero_inventario playbook.yml -e  
@muchas_variables.json
```

## 9.4. Recopilación de facts

Al ejecutar Ansible, el módulo de configuración (`setup`) es utilizado para recopilar información de la máquina que se gestiona, y para ello se utilizan todas las herramientas de recopilación de información que se encuentre disponibles, tales como `facter` y `ohai` que son dos de los más extendidos motores de *facts*. Al ejecutar el módulo de configuración en mi máquina OS X, me devuelve más de 3500 líneas de información de `ohai`.

En la siguiente imagen se puede ver un ejemplo de un fragmento de la salida.

```

localhost | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "192.168.0.4",
            "192.168.33.1"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::aa66:7fff:fe13:eff1%en0",
            "fe80::741d:61ff:fe91:14d4%awdl0"
        ],
        "ansible_architecture": "x86_64",
        "ansible_awdl0": {
            "device": "awdl0",
            "flags": [
                "UP",
                "BROADCAST",
                "RUNNING",
                "PROMISC",
                "SIMPLEX",
                "MULTICAST"
            ],
            "ipv4": [],
            "ipv6": [
                {
                    "address": "fe80::741d:61ff:fe91:14d4%awdl0",
                    "prefix": "64",
                    "scope": "0x6"
                }
            ],
            "macaddress": "",
            "media": "Unknown",
            "media_select": "autoselect",
            "mtu": "1484",
            "options": [
                "PERFORMNUD"
            ],
            "status": "active",
            "type": "unknown"
        },
        "ansible_date_time": {
    
```

Figura 4. Fragmento de la salida de la ejecución de la recopilación de *facts*. Fuente: elaboración propia.

La respuesta consiste en un diccionario denominado `ansible_facts` donde se incluye toda la información recopilada. A este diccionario se puede referenciar en los *playbooks* y plantillas como cualquier otra variable, indicando el nombre del diccionario y entre corchetes el nombre del *fact* entre comillas simples, para acceder a un dato específico, por ejemplo: `ansible_facts['ansible_architecture']`. Hay muchas de ellas que también se establecen como variables independientes, conservando el prefijo “`ansible_`”. También hay muchos *facts* útiles, como la arquitectura del sistema, la fecha/hora actual, la información ipv4 e ipv6 para todos

los adaptadores de red disponibles, etc. Incluso puedes calcular la cantidad de memoria libre en la máquina de destino con `ansible_memfree_mb`.

Una variable de *fact* interesante es `ansible_env`, donde podrás encontrar todas las variables de entorno definidas en el *host* destino. Es muy recomendable ejecutar el módulo de configuración y revisar toda la información disponible, lo que puedes hacer manualmente mediante:

```
ansible all -i fichero_inventario -m setup
```

## Deshabilitación de *facts*

Esta recopilación de *facts* no es gratis, ya que supone un tiempo considerable de procesamiento, el que se tarda en obtenerlos para cada *host*. Si no vas a necesitar ninguno de estos datos en tu *playbook*, puedes ahorrarte ese tiempo de procesamiento desactivando la recopilación de *facts* mediante `gather_facts: false`:

```
- hosts: all
  gather_facts: false
  tasks:
    - debug: msg="Hola Pepe"
```

## Facts.d

Si los *facts* que proporciona Ansible no te parecen suficientes, puedes crear tus propios *facts* o *facts* locales en la propia máquina que ejecuta Ansible. Ansible leerá todos los ficheros con extensión `*.fact` en el directorio `/etc/ansible/facts.d` y los hará accesibles como variables para el *playbook*. Es tu responsabilidad obtener estos *facts* en la máquina remota; se pueden poner allí a mano, escribir un *playbook* para llenarlos o de cualquier forma que creas conveniente. Pueden estar en formato INI, JSON o YAML, o ser un fichero ejecutable que interpretará el resultado como *facts*. Este mecanismo es útil para poder suministrar información adicional sobre cada *host* en donde se esté ejecutando para su uso en *playbooks*.

Veamos el siguiente fragmento:

```
[pepe]
usar_colores=1
sudo_sin_password=0
```

Si existe un fichero en `/etc/ansible/facts.d/users.fact` con dichos contenidos, estos *facts* estarán disponibles bajo la clave `ansible_local`. El nombre de fichero `.fact` se incluirá como clave dentro del diccionario `ansible_local` y, como valor dentro de este, se creará una clave por cada sección que se encuentre en el fichero:

```
"ansible_local": {
    "users": {
        "pepe": {
            "usar_colores": "1",
            "sudo_sin_password": "0"
        }
    }
}
```

## Cacheo de *facts*

Si necesitas hacer uso en tu *playbook* de los *facts* pero no estás dispuesto a pagar el precio en cada ejecución de la recopilación de los *facts*, puedes optar por el uso de la caché de *facts* para acelerar las ejecuciones sucesivas. Esta opción se habilita en el fichero de configuración de Ansible (`"ansible.cfg"`) y se debe definir si se va a utilizar ficheros Redis o JSON como soporte para la caché de los *facts*.

Para habilitar esta caché de *facts*, podemos utilizar el siguiente fragmento de configuración en el fichero `ansible.cfg`:

```
[defaults]
gathering = smart
fact_caching = jsonfile
```

```
fact_caching_connection = /directorio/cache  
fact_caching_timeout = 36000
```

Esto está configurando el mecanismo de recopilación de datos como inteligente, lo cual hace que se compruebe la caché de *facts* antes de volver a recopilarlos. Asimismo, estamos habilitando la caché mediante ficheros JSON que se almacenarán en la ruta especificada y finalmente establecemos que la caché es válida por un tiempo de 36 000 segundos (10 horas).

### Hostvars

Por último, con el diccionario `hostvars` Ansible nos proporciona acceso a información de los *hosts* diferentes a la máquina actual, lo que nos permite disponer de datos sobre otras máquinas que pueden ser útiles para la configuración a establecer desde nuestro *playbook*. Un ejemplo práctico de esto puede ser el de poder conocer la dirección IP privada de la máquina de base de datos accediendo por su nombre de *host* del inventario:

```
hostvars['basedatos.dominio.es']['ansible_eth0']['ipv4']['address']
```

El diccionario `hostvars` se rellena con cada *host* que Ansible va procesando, lo cual implica que solo podrás consultar la información de los *hosts* que ya se han visitado previamente. En caso de necesitar información de todas las máquinas antes de que se les haya accedido, un truco que se puede utilizar es el de habilitar la caché de *facts* y ejecutar un *playbook* con la periodicidad requerida que simplemente se conecte a cada máquina para recopilar sus *facts*.

## 9.5. Manipulación de variables

Tal como hemos mencionado, el motor de plantillas `Jinja2` es el encargado de la sustitución y manejo de las variables en Ansible. `Jinja2` es una herramienta de

plantillas para Python muy potente, de la que también se pueden encontrar actualmente disponibles versiones para muchos otros lenguajes de programación, tal como Twig para PHP o Nunjucks para NodeJS. El sistema de plantillas Jinja2 proporciona la sustitución de variables mediante {{syntax\_doble\_llave}}, pero también ofrece otras muchas funcionalidades, tales como las docenas de herramientas de manipulación de los valores de datos, llamadas **filtros**.

Hay más de 40 filtros incorporados en Jinja2, algunos de los cuales pueden ser de gran utilidad a la hora de manipular los valores de las variables de Ansible, tales como map, replace, rejectattr, selectattr, list, first y last.

En el siguiente ejemplo vamos a manejar una lista de empleados de una empresa, de la que únicamente queremos generar cuentas de usuario a los empleados del Departamento de Informática en los *hosts* que gestionamos. El *playbook* utiliza la lista de los empleados y aplica el filtro selectattr de Jinja2 para solo quedarnos con los que pertenecen al Departamento de Informática:

```
---
- hosts: all
  vars:
    empleados:
      - nombre: Pepe departamento: Informatica
      - nombre: Luis departamento: Informatica
      - nombre: Paco departamento: Finanzas
  tasks:
    - user: name="{{item.nombre}}" groups=developers append=yes
      loop:
        "{{empleados |
selectattr('departamento', 'equalto', 'Informatica') | list}}"
```

La lista de empleados podría obtenerse de cualquier otro sitio, pero aquí se ha puesto como variable codificada en el mismo *playbook* por simplicidad. Podrías por ejemplo tener una secuencia de comandos que extrajera la lista de empleados de una base de datos, por ejemplo, y se leyera del fichero resultante con vars\_files.

De cara a la construcción de *playbooks* genéricos, el uso de variables es fundamental. La separación de la lógica (*playbooks*) y los datos (mediante variables) es una buena práctica a la hora de definir infraestructura como código. Esto quiere decir que en los *playbooks* se utilizarán variables para definir todo y así los datos que se necesitan para definir por completo la configuración del sistema se encuentran en ficheros de variables. Este patrón de separación entre la lógica de configuración y los datos nos permitiría cambiar de herramienta de gestión de la configuración, manteniendo los mismos datos, o llegado el caso podríamos generar los ficheros de datos automáticamente para alimentar la lógica de configuración del sistema.

Utilizando `vars_files` podremos implementar este patrón de manera sencilla, como se muestra en el *playbook* a continuación, el cual no incluye la información sobre qué es lo que se quiere instalar (datos), sino únicamente la información que define cómo instalar los paquetes (lógica):

```
---
- hosts: basedatos
  vars_files:
    - mysql.yml
  tasks:
    - apt: name="{{ mysql_packages }}" state=present
```

El fichero de variables `mysql.yml` contiene la declaración de la variable `mysql_packages`:

```
---
mysql_packages:
  - mysql-server
  - python-mysqldb
```

Si alguna vez decides dejar de usar Ansible, será más fácil si utilizas este patrón. Al tener todos los datos separados en ficheros de variables, se podrán utilizar también como fuente de datos para otra herramienta, por lo que únicamente tendrás que migrar e implementar la lógica de configuración en la nueva herramienta.

## Filosofía de variables de Ansible

Ansible recomienda como buena práctica el definir cada variable una sola vez, siempre que sea posible. Para ello, trata de determinar en qué ubicación se debería declarar la variable para evitar tener que sobreescribirla posteriormente. Puede haber excepciones a esta regla, como puede ser la definición de valores por defecto en un rol.

Hemos visto en este tema la multitud de ubicaciones donde Ansible nos permite definir una variable, aunque en la mayor parte de las ocasiones lo más adecuado será declararlas en una de las ubicaciones más habituales.

A continuación, tienes el vídeo *Alternativas a Chef, Ansible y Puppet*.



Accede al vídeo

---

## 9.6. Referencias bibliográficas

Heap, M. (2016). *Ansible: from Beginner to Pro*. Apress.

Hochstein, L. y Moser R. (2014). *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. O'Reilly Media.