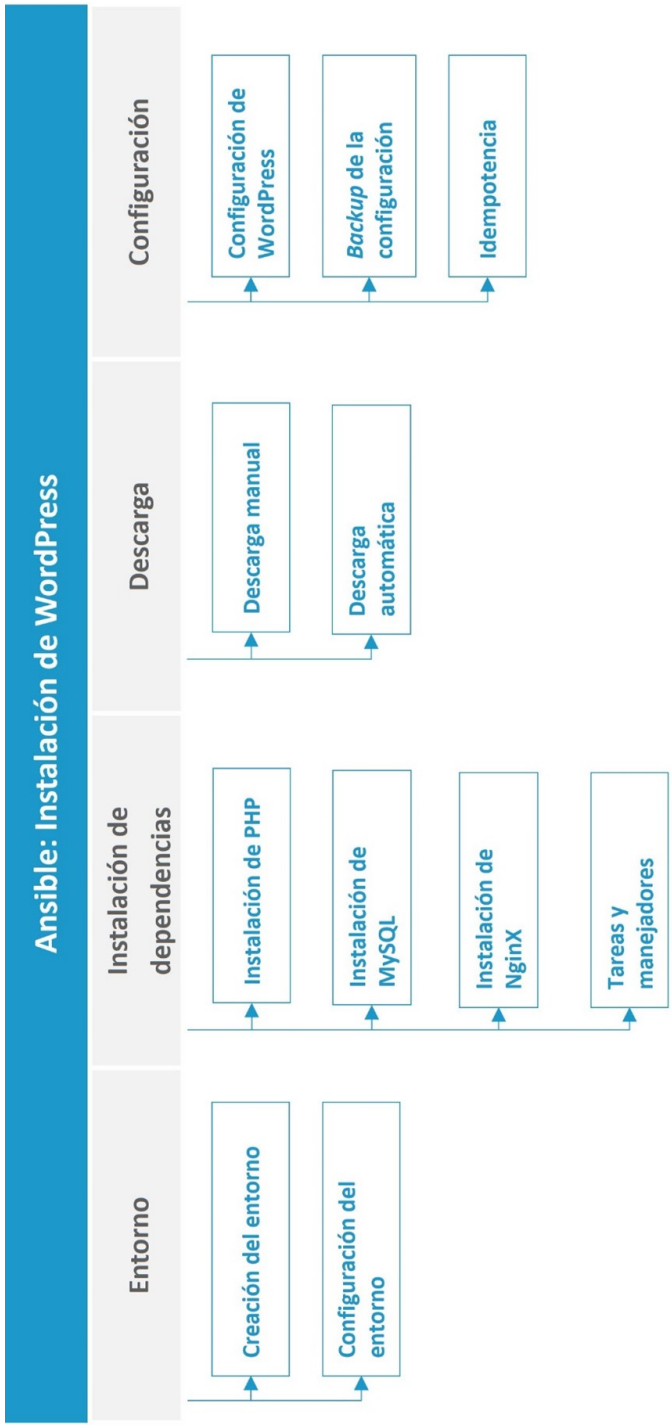
Herramientas de Automatización de Despliegues

Tema 7. Ansible. Instalación de WordPress

Índice

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.<em>playbook</em> ="provisioning/<em>playbook</em>.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 prerrequisito 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/<em>playbook</em>.yml
```

En el archivo `provisioning/playbook.yml`, especificamos en qué *host* o grupo de *host*s 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:

```
---
```

```
- <em>host</em>s: 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
```

Ideas clave

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 *host*s 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 de *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 rellenar 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 desees 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 <em>host</em>={{item}}

 password={{mysql_new_root_pass.stdout}}

 loop:

 - "{{ansible_<em>host</em>name}}"

 - 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 rellenar 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$ $scheme://$<em>host</em>$uri/ permanent;
 location ~* ^.+\.
(ogg|ogv|svg|svgz|eot|otf|woff|mp4|ttf|rss|atom|jpg|jpeg|gif|png|ico|zip|tgz|gz|rar|bz2|doc|xls|exe|ppt|tar|mid|midi|wav|bmp|rtf)$
{
 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;

}
```

Ideas clave

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 `fasctcgi_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 prerrequisitos 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
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 destruyeses el entorno y volvieses a crearlo, todavía estarías al 90 por ciento de completar la instalación y configuración de WordPress, ya que te faltaría por 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.

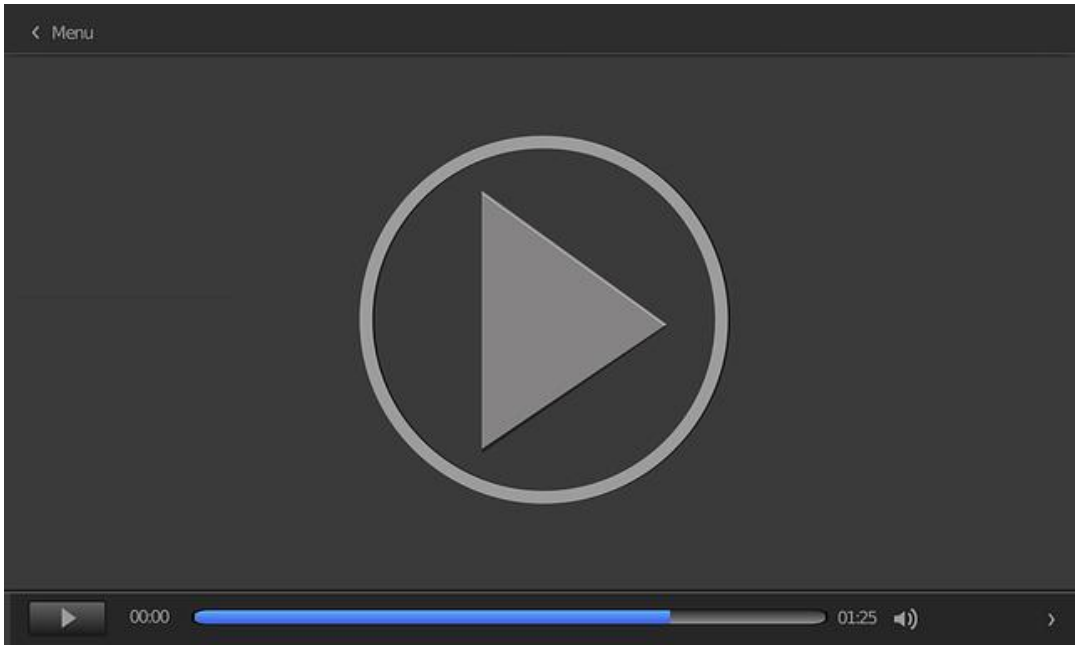
En nuestro caso, nunca va a ser necesario que el comando que comprueba si existe la base de datos devuelva `changed`, por lo que podemos especificar `changed_when: false` para que siempre nos devuelva OK. Modifica por tanto la tarea que comprueba la base de datos como se indica a continuación para que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

Ideas clave

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



Accede al vídeo:https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=6df341a4-d1cc-4f36-afe1-abb6011088e1

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.

Sintaxis y handlers en Ansible

Fuentes, D. (2018, agosto 17). Sintaxis y handlers en Ansible. *Blog Virtualizado.* https://blogvirtualizado.com/sintaxis-y-handlers-en-ansible/

Interesante artículo sobre los distintos formatos de sintaxis en Ansible y sus características, así como sobre los manejadores, explicados con un ejemplo práctico de su funcionamiento.

Documentación de referencia de Ansible

Red Hat, Inc. (2020). *Ansible Documentation.* https://docs.ansible.com/ansible/latest/index.html

En el sitio oficial de documentación Ansible es donde podrás encontrar la documentación de referencia más completa y actualizada de la herramienta, así como la versión de documentación correspondiente con la propia versión de la herramienta que estés utilizando. Es el primer recurso al que acceder en busca que cualquier concepto Ansible, para asegurarnos de su veracidad y actualidad.

Documentación de referencia de Ansible – Manejadores

Red Hat, Inc. (2020). *Intro to playbooks.* https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html#handlers-running-operations-on-change

Enlace directo a la documentación oficial más actualizada de Ansible sobre manejadores, donde puedes encontrar la explicación de su funcionamiento y sus consideraciones.

LAMP Stack with Ansible

Schirbel, M. (2019, enero 26). LAMP Stack with Ansible. *Blog de Marcelo Schirbel Gomes.* https://medium.com/@mschirbel/lamp-stack-with-ansible-7ad9316c8a7f

Este otro tutorial también aborda la instalación de un stack LAMP, pero esta vez lo realiza sobre varias máquinas virtuales gestionadas con Vagrant, lo que le proporciona otra perspectiva enriquecedora que conocer.

Cómo usar Ansible para instalar y configurar LAMP en Ubuntu 18.04

Heidi, E. (2020, febrero 20). *Cómo usar Ansible para instalar y configurar LAMP en Ubuntu 18.04.* Digital Ocean. https://www.digitalocean.com/community/tutorials/how-to-use-ansible-to-install-and-set-up-lamp-on-ubuntu-18-04-es

En este artículo en castellano puedes ver la manera de instalar un stack LAMP (Linux, Apache, MySQL, PHP) similar al que se ha realizado en este tema, aunque con algunas diferencias que te pueden aportar otra perspectiva de cómo instalar estas herramientas.

Test

1. ¿Cómo podemos hacer que Ansible ejecute en modo privilegiado?

   A. Utilizando la opción `become`.

   B. Especificando el usuario `root`.

   C. Utilizando la opción `privileged`.

   D. Indicando en el comienzo del *playbook* la opción `hosts: all`.

2. ¿Cómo podemos instalar varios paquetes en una misma tarea?

   A. No se puede, hay que copiar y pegar la tarea varias veces cambiando el nombre del paquete.

   B. Especificando el atributo `multiple`.

   C. Proporcionando una lista de nombres en el atributo `name`.

   D. Utilizando la tarea `apt-loop`.

3. ¿Cómo podemos ejecutar directamente un comando de sistema operativo?

   A. No se puede, hay que utilizar los módulos que ya existen.

   B. Creando un módulo `custom` que ejecute el comando que necesitamos.

   C. Utilizando una función Python equivalente.

   D. Utilizando el módulo `command` o `Shell`.

4. ¿Para qué sirve el uso de register en una tarea?

   A. Para activar el registro del sistema.

   B. Para registrar (mostrar) la operación por la salida estándar.

   C. Para guardar el valor de retorno en una variable.

   D. Ninguna de las anteriores.

5. ¿Qué diferencia hay entre los módulos copy y template?

A. No existe diferencia, son equivalentes.

B. `template` copia ficheros de plantilla al *host* destino, y `copy`, cualquier otro fichero.

C. `template` simplemente procesa la plantilla, y `copy` copia el fichero al *host* destino.

D. `template` procesa la plantilla y copia el fichero resultante al *host* destino, y `copy` simplemente copia el fichero.

6. ¿Para qué sirve la opción creates en una tarea command?

A. Para no ejecutar el comando si ya existe ese fichero.

B. Para que se cree el fichero que se indica.

C. Es meramente informativo, para indicar al usuario lo que se va a crear.

D. Es meramente informativo, para indicar a Ansible lo que se va a crear.

7. ¿Cómo podemos condicionar la ejecución de una tarea al resultado de otra?

A. Utilizando `when` en la tarea que condiciona.

B. Registrando el resultado de la tarea que condiciona con `register`, y utilizando `when` en la tarea condicionada.

C. Utilizando `when` en la tarea condicionada.

D. Utilizando `when` en la tarea que condiciona y register en la tarea condicionada.

8. ¿Cuándo se ejecuta un handler (manejador)?

A. En secuencia, por orden de definición en el *playbook*.

B. Después de todas las tareas.

C. Cada vez que se le invoca mediante `notify`.

D. Al final del bloque de tareas que lo notifica, una sola vez.

9. ¿Para qué se utiliza `ignore_errors`?

A. Para que los errores no se muestren por la salida de ejecución.

B. Para parar la ejecución cuando se produzca un error.

C. Para continuar con la ejecución del *playbook*.

D. Para que en la lista de errores al finalizar la ejecución no se contabilicen los ignorados.

10. ¿Cómo nos permite Ansible conservar la idempotencia al ejecutar comandos?

A. Usando la opción `changedwhen` para indicar cuándo un comando ha realizado cambios realmente.

B. Utilizando el módulo `shell`, en vez de `command`.

C. Al utilizar `command`, perdemos la idempotencia.

D. Solo podemos mantener la idempotencia condicionando el comando con `when` y `creates`.