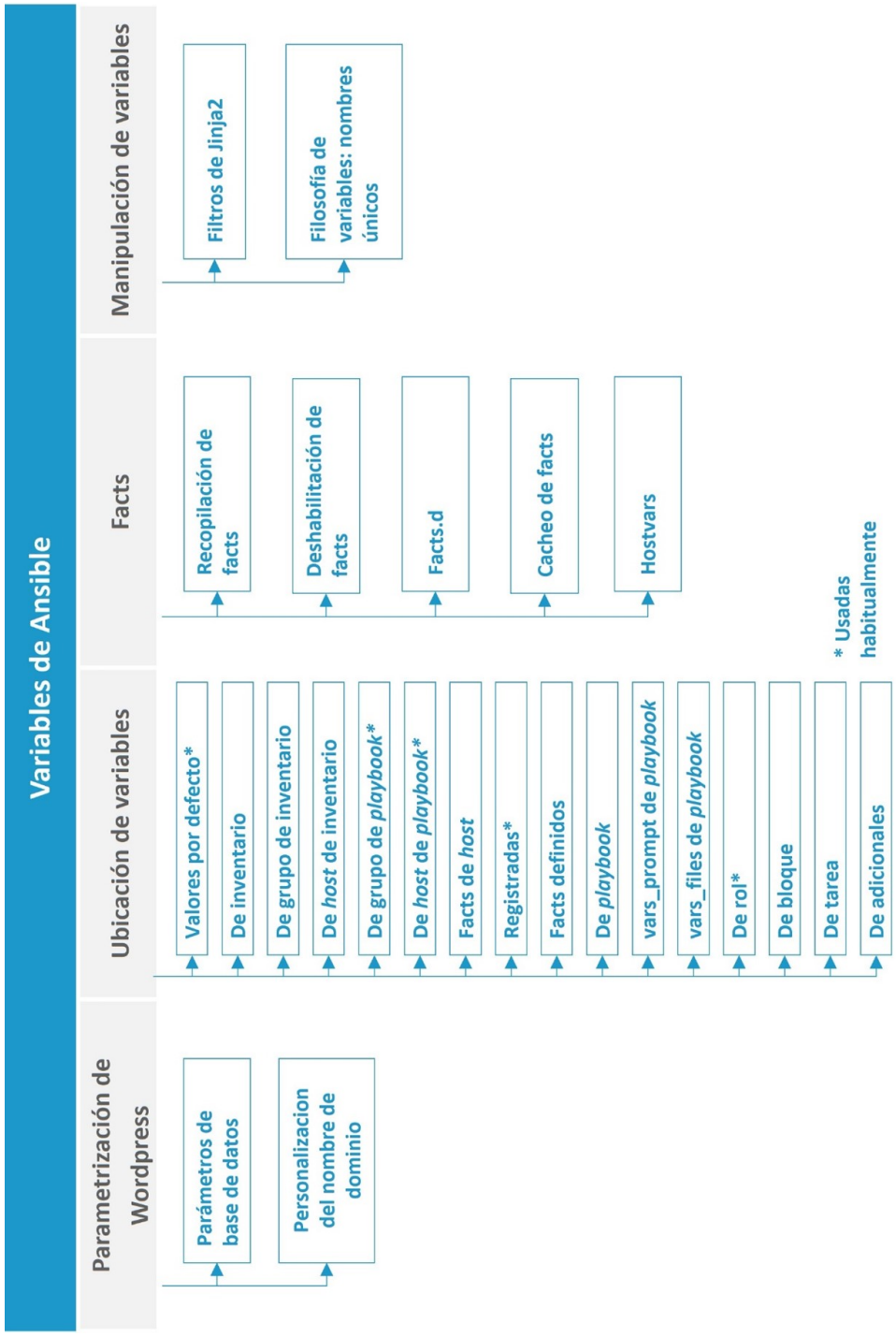
Herramientas de Automatización de Despliegues

Tema 9. Variables de Ansible

Índice



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 *playbook*s 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 desees 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 sustituyamos 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 `inventoryy` 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:

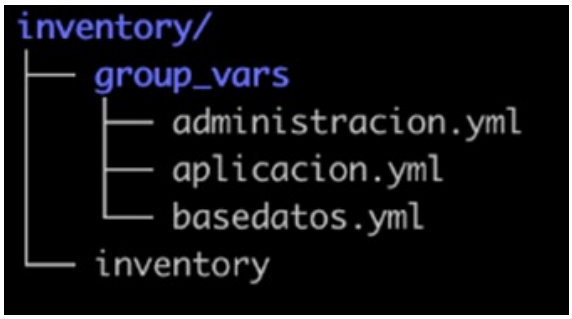


Figura 1. Ejemplo de directorios y ficheros para las variables de grupo de inventario. Fuente: elaboración propia.

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:

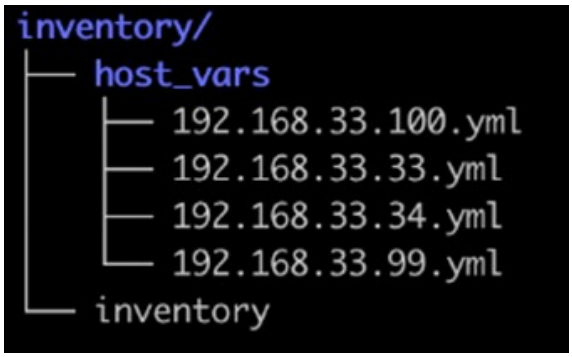


Figura 2. Ejemplo de directorios y ficheros para las variables de *host* de inventario. Fuente: elaboración propia.

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 coincida 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_<em>fact</em>
```

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 quieres 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 quieres 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:

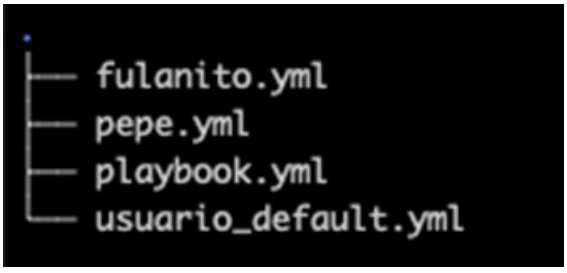


Figura 3. Ejemplo de ficheros para las variables de fichero. Fuente: elaboración propia.

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 quieres 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 *fact*er 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.

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 `{{sintaxis_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.

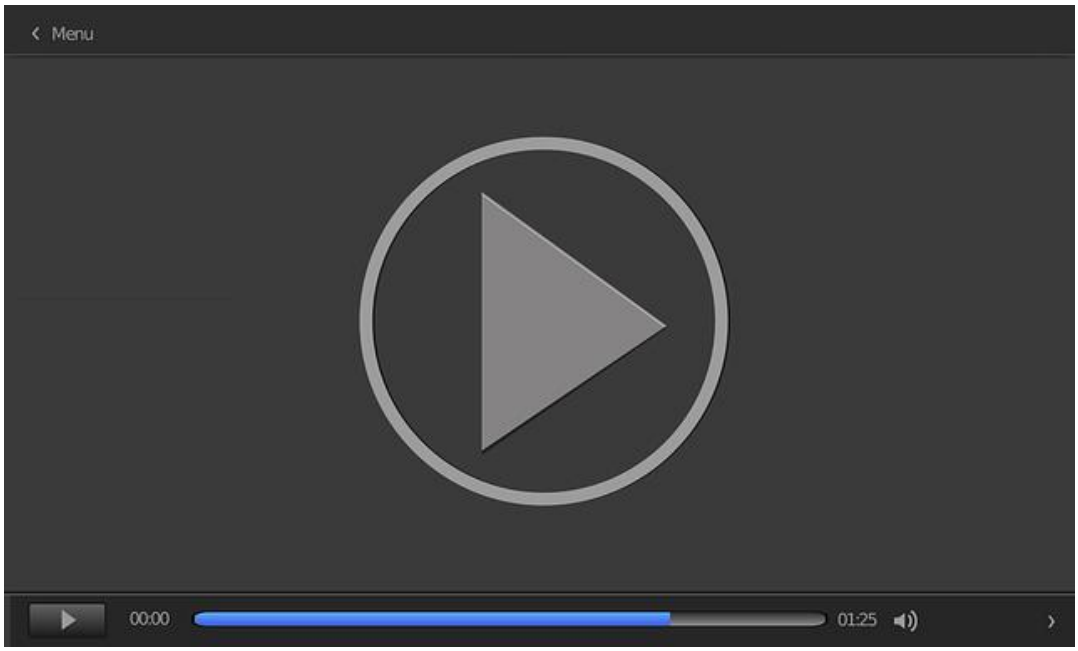
Filosofía de variables de Ansible

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:https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=803cbe67-e94c-4ac5-b918-abb601066173

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.

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.

Jinja

Pallets. (2007-2020). *Jinja*. https://jinja.palletsprojects.com

Jinja es el lenguaje de plantillas para Python que utiliza Ansible. La sintaxis permite mucho más que la mera sustitución de variables, tal como bucles, sentencias condicionales, etc., lo que le proporciona una gran potencia y flexibilidad.

Filtros Jinja

Pallets. (2007-2020). *Jinja*. https://jinja.palletsprojects.com/en/3.0.x/templates/#list-of-builtin-filters

Una de las funciones más potentes de Jinja son los filtros, con los que podemos manipular las variables que se manejan. Esta referencia es la documentación oficial relativa a filtros que nos proporciona información sobre todos los que hay disponibles.

Using Variables

Red Hat, Inc. (2019). *Using Variables. Ansible Documentation.* https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html

Esta es la página de la documentación oficial donde podrás encontrar toda la información relativa a las variables en Ansible.

Test

1. ¿Para qué puedes utilizar las variables en Ansible?

   A. Como valores para contenido, para rellenar una plantilla, por ejemplo.

   B. Como valores condicionales, para decidir si se ejecuta una tarea o no.

   C. Como valores para contenido y valores condicionales.

   D. Como valores para contenido únicamente, pero las que se definan así se pueden utilizar para condicionar acciones.

2. ¿Cómo referenciamos un rol desde un *playbook* pasándole parámetros?

   A. Incluyendo la opción `vars` en el `include` del rol.

   B. Incluyendo la opción `variables` en el `include` del rol.

   C. Incluyendo la opción `param` en el `include` del rol.

   D. Incluyendo la opción `parameters` en el `include` del rol.

3. ¿Cómo referenciar a una variable en un *playbook* o plantilla para incluir su valor?

   A. Especificando en nombre de variable entre dobles corchetes: `[[var]]`

   B. Especificando en nombre de variable entre dobles llaves: `{{var}}`

   C. Especificando en nombre de variable entre dobles barras: `//var//`

   D. Especificando en nombre de variable entre llaves y con un $ delante: `${var}`

4. ¿Qué ubicación de declaración de variables tiene la precedencia más baja?

   A. Variables adicionales (parámetros de ejecución).

   B. Variables de grupo de *playbook* (`groupvars`).

   C. Los valores por defecto de los roles.

   D. Variables de rol (parámetros del rol).

5. ¿Y qué ubicación de declaración de variables tiene la precedencia más alta?

A. Variables adicionales (parámetros de ejecución).

B. Variables de grupo de *playbook* (`groupvars`).

C. Los valores por defecto de los roles.

D. Variables de rol (parámetros del rol).

6. ¿Cómo podemos saber en Ansible el sistema operativo que está ejecutando una máquina?

A. Mediante una variable de inventario.

B. Mediante una variable de *host* de inventario.

C. Mediante un *fact* definido.

D. Mediante un *fact* de *host*.

7. ¿Para qué sirve una variable de tipo `vars_prompt`?

A. Para solicitar al usuario su valor en tiempo de ejecución.

B. Para solicitar al usuario su nombre y valor en tiempo de ejecución.

C. Para obtener el estilo de *prompt* del sistema que se utiliza.

D. Para que su valor esté disponible pronto, desde el inicio del *playbook*.

8. ¿Cómo puedes filtrar una lista de objetos contenidos en una variable Ansible?

A. Utilizando la opción `when` en la tarea.

B. Utilizando la opción `filter` en la tarea.

C. Mediante filtro de Jinja aplicado a la variable.

D. Mediante filtro de Jinja aplicado a la tarea.

9. ¿Dónde podemos encontrar en Ansible información sobre la máquina que vamos a manejar?

A. En las variables de entorno.

B. En variables prefijadas con "`facts`".

C. En variables prefijadas con "`ansible_`".

D. En el diccionario "`ansible_facts`" o en variables prefijadas con "`ansible_`".

10. ¿Cómo puedes acelerar la ejecución de *playbooks* con respecto a los *facts*?

A. Desactivándolos si no los necesitas, o activando la caché de *facts* si los usas.

B. Únicamente puedes desactivar la recopilación de *facts*.

C. Únicamente puedes activar la caché de *facts*.

D. Desactivándolos si no los necesitas, activando la caché de *facts* si los usas o usando *facts* locales.