Ansible (III)

Variables

1.-a) Edita l'inventari de l'Ansible que tens instal.lat a la màquina "A" per tal de què ara tingui el següent contingut (on "ip.maq.B" i "ip.maq.C" han de substituïr-se pel seu valor adient, òbviament)...:

```
[victimes]
ip.maq.B varB=manolo
ip.maq.C
[victimes:vars]
varGlobal=pepito
```

NOTA: L'inventari anterior seria equivalent a tenir la següent estructura de fitxers...:
*Un fitxer "/etc/ansible/group_vars/victimes.yml" (o, alternativament "/ruta/playbook/group_vars/victimes.yml") amb el següent contingut: *varGlobal: pepito*
*Un fitxer "/etc/ansible/host_vars/ip.maq.B.yml" (o, alternativament "/ruta/playbook/host_vars/ip.maq.B.yml") amb el següent contingut: *varB: manolo*
...deixant llavors el fitxer d'inventari simplement així:

```
[victimes]
ip.maq.B
ip.maq.C
```

...i tot seguit executa la següent comanda, *ansible-playbook -e "varD=ana varE=felisa" hola.yml* , on "hola.yml" és un playbook amb el següent contingut. Observa quin és el contingut de cada variable mostrada (i raona per què):

```
- name: Playbook per veure els valors de variables definides a diferents llocs
  hosts: all
  remote_user: pepito
  #Aquesta secció 'vars' es podria haver definit en un fitxer apart anomenat group_vars/all.yml
  vars:
      varPlayD : "{{ varD }}"
      varPlayE : "{{ varE }}"
      varPlayF : "perico"
  tasks:
  - name: Veure les variables definides en l'inventori
    debug:  msg="varGlobal val {{ varGlobal }} i varB val {{ varB }}"
  - name: Veure les variables passades per paràmetre
    debug:  msg="varPlayD val {{ varPlayD }} i varPlayE val {{ varPlayE }}"
  - name: Veure les variables definides en el playbook
    debug:  msg="varPlayF val {{ varPlayF }}"
  - name: Veure les variables definides en la tasca
    vars:
        varTaskG : "juan"
    debug:  msg="varTaskG val {{ varTaskG }}"
  - name: Veure el valor d'un determinat "fact"
    debug:  msg="La distribució de la víctima és {{ ansible_facts['distribution'] }}"
```

NOTA: Fixa't en la sintaxis utilitzada a la darrera tasca del playbook anterior per accedir al valor d'un determinat "fact". Aquesta sintaxis es correspon al següent patró: *{{ ansible_facts["nomFact"] }}* Hi ha, no obstant, una altra sintaxi alternativa equivalent, que és així: *{{ ansible_nomFact }}*
NOTA: Fixa't també en què al playbook anterior no es realitzen les tasques per la màquina C degut a què ocorre un error al principi i ja no es continua per aquella màquina. L'error és que varB no es troba definida per ella. Aquest error es pot obviar amb l'opció *ignore_errors:yes*, la qual fa que, tot i haver-hi algun error, l'execució del "playbook" continua igualment a la màquina on s'ha detectat l'error

aII) ¿Quins canvis veus respecte la sortida de l'apartat anterior si ara executes la comanda *ansible-playbook -e "varD=luisa varE=pepa" hola.yml*

A més de fer-ho en l'arxiu d'inventori o en els arxius "group_vars/hosts_vars" es poden definir variables en un fitxer extern mitjançant la directiva *vars_files:* o, alternativament, fent servir el mòdul *ansible.builtin.include_vars* (la diferència més important entre els dos mètodes és que el primer, a l'igual que la directiva *vars:*, actua a nivell de "playbook" general (i s'interpreta, doncs, quan s'inicia l'execució del "playbook") i la segona actua a nivell d'una "task" en particular (i s'interpreta, doncs, quan s'inicia l'execució de la tasca en particular). Aquest detall pot fer que una variable, segons com estigui definida, pugui tenir un valor o un altre segons les circumstàncies.

2.-Digues què mostra l'execució del següent playbook...:

```
- name: Recollir variables d'un fitxer (sobreescrivint el valors definit al playbook...o no)
  hosts: all
  remote_user: pepito
  vars:
      favcolor: blue
  vars_files:
  - /home/usuari/colors.yml
  tasks:
  - name: Mostrar els valors
    debug: msg="El valor de la variable 'favcolor' és {{ favcolor }} i el de 'worstcolor' és {{ worstcolor }}"
```

...si tens (a la màquina controladora) un arxiu anomenat "/home/usuari/colors.yml" amb el següent contingut (i per què veus el que veus...tingues en compte la precedència entre *vars:* i *vars_files:* tal com s'explica a l'enllaç indicat a la nota):

```
favcolor: red
worstcolor: black
```

NOTA: Llegeix https://docs.ansible.com/ansible/latest/reference_appendices/general_precedence.html per recordar les precedències de les opcions de configuració i les variables segons on estiguin definides (al terminal, a l'inventari, als playbooks, etc)

Una altra possibilitat que ofereix l'ús de variables és, mitjançant la secció especial *vars_prompt*, demanar interactivament els seus valors per tal d'executar playbooks amb valors d'entrada diferents cada cop (per exemple, per introduir una contrasenya que no es vol deixar escrita).

3.-Prova el següent playbook, observa què passa i dedueix per a què serveixen les opcions "default" i "private".

```
- name: Preguntar dades interactivament (els mateixos valors per totes les víctimes)
  hosts: all
  remote_user: pepito
  vars_prompt:
  - name: "nom"
    prompt: "What is your name?"
    default: "Simply the best"
    private: no
  - name: "contra"
    prompt: "What is your password?"
  - name: "favcolor"
    prompt: "What is your favorite color?"
    private: no
  tasks:
  - name: Veure el valor introduït
    debug:  msg="Your name is {{ nom }}, your pass is {{ contra }} , your color is {{ favcolor }}"
```

NOTA: Si a través del paràmetre *-e* de la comanda *ansible-playbook* s'indica el valor d'una variable indicada a la secció "vars_prompt", no es preguntarà interactivament per ella perquè s'agafarà automàticament el valor indicat al terminal
NOTA: Si la llibreria Passlib (paquet "python3-passlib") està instalada, "vars_prompt" podrà xifrar el valor introduït afegint les següents opcions: "encrypt: sha512_crypt", "confirm:yes" and "salt_size:7". Per saber altres valors possibles per l'opció "encrypt" es pot consultar https://docs.ansible.com/ansible/latest/user_guide/playbooks_prompts.html#prompts

"Facts"

Hi ha tres tipus de dades que es poden guardar a les variables o "facts" recollits: una "llista" (és a dir, una matriu o array), un "diccionari" (és a dir, un objecte JSON) o un text. Per accedir a l'element en la posició "n" de la llista (aquí "n" representa el valor enter de l'índex, a partir de 0), cal emprar la notació *variableLlista[n]* ; per accedir a un element del diccionari cal escriure *variableDiccionari.nomElement*

NOTA: Ansible proporciona un filtre anomenat "type_debug" per obtenir el tipus de dades Python subjacent d'una variable o "fact": la sintaxi que cal utilitzar és: *{{ nomVariable | type_debug}}*

4.-a) Executa el següent playbook i digues de quin tipus són els "facts" *{{ ansible_architecture }}, {{ ansible_apparmor }} , {{ ansible_all_ipv4_addresses }} i {{ ansible_mounts }}* i quins són els valors que el playbook mostra de cadascun d'ells.

```
- name: Ansible Variable Example Playbook
  hosts: victimes
  remote_user: pepito
  tasks:
  - name: Data types of some facts
    debug:
        msg:
        - " Data type of 'ansible_architecture'  is {{ ansible_architecture | type_debug }} "
        - " Data type of 'ansible_apparmor' is {{ ansible_apparmor | type_debug }} "
        - " Data type of 'ansible_all_ipv4_addresses' is {{ ansible_all_ipv4_addresses | type_debug }} "
        - " Data type of 'ansible_mounts' is {{ ansible_mounts | type_debug }} "
  - name:  Printing a simple text value
    debug:  'msg="One: {{ ansible_architecture }}"'
  - name: Accessing an entire list
    debug: 'msg="Two: {{ ansible_all_ipv4_addresses }}"'
  - name: Accessing the first element of a list
    debug:  'msg="Three: {{ ansible_all_ipv4_addresses[0] }}" '
  - name: Printing a dictionary
    debug:  'msg="Four: {{ ansible_apparmor }}"'
  - name: Accessing an element of dictionary
    debug: 'msg="Five: {{ ansible_apparmor.status }}"'
  - name: Printing a list of dictionaries
    debug:  'msg="Six: {{ ansible_mounts }}"'
  - name: Printing an element of the first element of the list of dictionaries
    debug:  'msg="Seven: {{ ansible_mounts[0].device }}"'
```

NOTA: Les cometes simples encerclant l'opció *msg* del mòdul *debug* calen per escapar el símbol ":" (i així no tenir errors)

aII) ¿De quin tipus són els següents facts i quina informació contenen?: *{{ ansible_hostname }}, {{ ansible_distribution_release }}, {{ ansible_devices.sda.model }}, {{ ansible_processor_cores }}* i *{{ ansible_processor_count }}* . ¿I la variable especial *{{ ansible_facts }}* ?

Si en algun moment del playbook es vol accedir a un "fact" específic obtingut d'una màquina concreta, es pot utilitzar la notació següent:

Per accedir a un "fact" de text obtingut de la màquina 10.0.0.1:
"{{ hostvars['10.0.0.1']['ansible_architecture'] }}"

Per accedir a un "fact"-element de diccionari:
"{{ hostvars['10.0.0.1']['ansible_apparmor']['status'] }}"

Per accedir a un "fact"-llista:
"{{ hostvars['10.0.0.1']['ansible_all_ipv4_addresses'] }}"

Per accedir a un "fact"-llista de diccionaris:
"{{ hostvars['10.0.0.1']['ansible_mounts'] }}"

Per accedir a un "fact"-element de llista:
"{{ hostvars[‘10.0.0.1’][‘ansible_all_ipv4_addresses’][0] }}"
Per accedir a un "fact"-diccionari:
"{{ hostvars[‘10.0.0.1’][‘ansible_apparmor’] }}"

Per accedir a un "fact"-element d’un element d’una llista de diccionaris:
"{{ hostvars[‘10.0.0.1’][‘ansible_mounts’][0][‘device’] }}"

b) Havent llegit el paràgraf blau anterior i observant la sortida per pantalla del mòdul "setup" (executat amb la comanda *ansible* sobre la màquina 10.0.0.1), digues si aquesta expressió podria ser correcta o no:

hostvars[‘10.0.0.1’]['ansible_enp0s3']['ipv4']['address']

Filtres

A l’hora de treballar amb valors de variables o "facts" és molt útil saber que podem utilitzar els anomenats "filtres" d’Ansible (http://docs.ansible.com/ansible/latest/user_guide/playbooks_filters.html), els quals en permeten manipular aquests valors de múltiples formes diferents. En general els filtres Ansible s’escriuen així: *{{ nomVariable | nomFiltre }}* (de fet, l’expressió "type_debug" que vam veure en un apartat anterior no és res més que un filtre) i, entre els molts que hi ha, podem destacar els següents:

{{ unaVariable | default(5) }} : Assigna un valor per defecte (en aquest cas, "5") a la variable indicada si aquesta no està definida (en comptes de disparar un error i, per tant, aturar el play a la màquina en qüestió)
{{ unaVariable | length }} : Retorna el número de caràcters que té el valor de la variable indicada (si aquesta és de tipus cadena) o bé el número d’elements (si aquesta és de tipus llista) o bé el tipus de parelles clau←>valor (si aquesta és de tipus diccionari)

{{ unaVariable | b64encode) }} : Retorna la codificació base64 del valor de la variable indicada
{{ unaVariable | b64decode) }} : Retorna la decodificació del valor base64 de la variable indicada
{{ unaVariable | hash("md5") }} : Retorna el hash MD5 del valor de la variable indicada
{{ unaVariable | password_hash("sha512") }} : Retorna el hash SHA512 saltejat amb una sal aleatòria. Ideal per generar valors per l’arxiu */etc/shadow*
{{ unaVariable | password_hash("sha512","unaSal") }} : Retorna el hash SHA512 saltejat amb la sal indicada. Ideal per generar valors per l’arxiu */etc/shadow*

NOTA: Un mètode idempotent per generar hashes únics per sistema és usar una sal que sigui consistent entre execucions. Això es pot aconseguir usant una llavor consistent a l'hora de generar la sal, així:
{{ 'secretpassword' | password_hash('sha512', 65534 | random(seed=nomMaquinaAInventori)) }}

{{ unaCadena | upper }} : Retorna la cadena en qüestió tota en majúscules
{{ unaCadena | lower }} : Retorna la cadena en qüestió tota en minúscules
{{ unaCadena | regex_search("expReg") }} : Retorna la primera subcadena que concordi amb l’expressió regular indicada com a paràmetre de *regex_search()* dins de la cadena indicada. Si no es troba cap coincidència, no retornarà res. Es pot fer que la recerca sigui case-insensitive i/o al llarg de diferents línies (si la cadena a inspeccionar en tingués vàries) afegint el paràmetre corresponent, així: *{{ "ho\nLA" | regex_search("^bar", multiline=True, ignorecase=True) }}*
{{ unaCadena | regex_findall("expReg") }} : Retorna una llista amb totes les subcadenes que concordin amb l’expressió regular indicada com a paràmetre de *regex_findall()* dins de la cadena indicada.
{{ unaCadena | regex_replace("expReg","subcadena") }} : Retorna una cadena a partir de la cadena original on totes les ocurrències trobades de l'expressió regular indicada com a primer paràmetre de *regex_replace()* s'han substituït per la subcadena indicada com a segon paràmetre. Si no s'indiqués cap segon paràmetre, llavors s'eliminarà de la cadena original totes les ocurrències concordants amb l'expressió regular indicada. Per exemple, fent *{{ "localhost:80" | regex_replace(":80") }}* s'obtindrà la cadena "localhost"

NOTA: Si parts de l'expressió regular s'envolten de parèntesis, s'estaran fent grups que es podran indicar (seguint el número d'ordre en què hi apareguin, així: \\1, \\2, etc) a la subcadena substituïdora. Per exemple, per convertir la paraula "ansible" en "able" es podria fer*{{ 'ansible' | regex_replace('^a.*i(.*)$', 'a\\1') }}* o de forma similar, per convertir "foobar" en "bar" es podria fer *{{ 'foobar' | regex_replace('^f.*o(.*)$', '\\1') }}* En qualsevol cas, per saber totes les opcions i sintaxi de les expressions regulars admeses per Ansible es pot consultar la documentació oficial de la llibreria Python responsable: https://docs.python.org/3.8/library/re.html

{{ unaLlista | min }} : Retorna el valor mínim d'una llista (per ex: de *[3,4,2]* extreuria el 2)
{{ unaLlista | max }} : Retorna el valor màxim d'una llista (per ex: de *[3,4,2]* extreuria el 4)
{{ unaLlista | random }} : Retorna un valor aleatori d'una llista. També es pot fer servir per generar un número sencer aleatori entre 0 i l'indicat si s'escriu *{{ 5 | random }}* (en aquest cas el número indicat seria 5)
{{ unaLlista | unique }} : Retorna una llista sense valors repetits (per ex: de *[1,2,1,2]* treuria *[1,2]*)
{{ unaLlista | join(" ") }} :Retorna una cadena formada pels elements de la llista indicada i fent servir com a caràcter separador l'indicat com a paràmetre de *join()* (per ex: de *[hola,adeu]* treuria *"hola adeu"*)
{{ unaLlista | union(unaAltraLlista) }} : Retorna una llista-unió a partir de les dues indicades (per ex: de *[1,2]* i *[3,4]* treuria *[1,2, 3, 4]*)
{{ unaLlista | intersect(unaAltraLlista) }} : Retorna una llista-intersecció a partir de les dues indicades (per ex: de *[1,2]* i *[2,4]* treuria *[2]*)
{{ unaLlista | difference(unaAltraLlista) }} : Retorna una llista-diferència (és a dir, els ítems que existeixen a la primera llista indicada que no existeixen a la segona (per ex: de *[1,2]* i *[2,4]* treuria *[1]*)

{{ unaUrl | urlsplit("hostname") }} : Retorna el hostname d'una url donada (per ex: retorna "www.acme.com" de *"http://www.acme.com:9000/dir/index.html?query=term&lang=fr#fragment"*)
{{ unaUrl | urlsplit("port") }} : Retorna el hostname d'una url donada (per ex: retorna "9000" de *"http://www.acme.com:9000/dir/index.html?query=term&lang=fr#fragment"*)
{{ unaUrl | urlsplit("path") }} : Retorna el hostname d'una url donada (per ex: retorna "/dir/index.html" de *"http://www.acme.com:9000/dir/index.html?query=term&lang=fr#fragment"*)
{{ unaUrl | urlsplit("query") }} : Retorna el hostname d'una url donada (per ex: retorna "query=term&lang=fr" de *"http://www.acme.com:9000/dir/index.html?query=term&lang=fr#fragment"*)
{{ unaUrl | urlsplit("fragment") }} : Retorna el hostname d'una url donada (per ex: retorna "fragment" de *"http://www.acme.com:9000/dir/index.html?query=term&lang=fr#fragment"*)

NOTA: Si no s'indica cap paràmetre a *urlsplit()*, es retornarà un diccionari amb tots els camps

{{ unPath | basename }} : Retorna la darrera part d'una ruta (normalment el nom d'un fitxer o de la darrera carpeta; per ex: de *"/etc/asdf/foo.txt"* retornarà *"foo.txt"*)
{{ unPath | dirname }} : Retorna la part d'una ruta corresponent a les carpetes i subcarpetes (excepte la darrera; per ex: de *"/etc/asdf/foo.txt"* retornarà *"/etc/asdf"*)
{{ unPathDeLink | realpath }} : Retorna la ruta "real" a la què apunta el link indicat

5.-a) Digues, a partir d'haver llegit els paràgrafs blaus anteriors i abans de provar-les, quin creus que seria el valor del missatge mostrat pel mòdul *debug* a les diferents tasques següents, i després comprova que ho hagis encertat provant-les i observant el resultat:

```
- name: Ansible Filters Example Playbook
  hosts: victimes
  remote_user: pepito
  vars:
    llista: [1, 2, 3, 3, 2]
    cadena: "M'agraden els macarrons i els canelons"
  tasks:
  - name: U
    debug: msg="{{ cosa | default("lalala") }}"
  - name: Dos
    debug: msg="{{ cadena | password_hash("sha512","abcd") }}"
  - name: Tres
    debug: msg="{{ cadena | regex_findall("ca") }} i també {{ cadena | regex_search("^ca") }}"
  - name: Quatre
    debug: msg="{{ cadena | regex_replace("ca","kA") }}"
  - name: Cinc
    debug: msg="{{ llista | unique }}"
  - name: Sis
    debug: msg="{{ llista | join("+") }}"
  - name: Set
    debug: msg="{{ llista | intersect([1,3,4,3]) }}"
```

```
- name: Vuit
  debug: msg="{{ "https://www.google.com/search?source=hp&q=macarrons&uact=5" | urlsplit("path") }}"
- name: Nou
  debug: msg="{{ "https://www.google.com/search?source=hp&q=macarrons&uact=5" | urlsplit("query")  }}"
- name: Deu
  debug: msg="{{ "/etc/pam.d/cups" | basename }}"
- name: Onze
  debug: msg="{{ "/etc/pam.d/cups" | dirname }}"
- name: Dotze
  debug: msg="{{ {"a":"1","b":"2"} | length }}"
```

6.-a) Després de llegir la documentació oficial del mòdul "**package_facts**", digues què mostrarien a pantalla les diferents tasques del següent playbook i prova-les:

```
- name: Misteri
  hosts: victimes
  remote_user: pepito
  tasks:
  - name: Recollir dades sobre xxx i afegir-les a l'objecte xxx
    package_facts:
  - name: Mostrar xxx
    debug: msg="{{ ansible_facts }}"
  - name: Mostrar yyy (això és possible gràcies a haver executat abans el mòdul package_facts)
    debug: msg="{{ ansible_facts.packages }}"
  - name: Mostrar una quantitat que representa yyy
    debug: msg="{{ ansible_facts.packages | length }}"
  - name: Mostrar zzz
    debug: msg="{{ ansible_facts.packages['curl'] }}"
  - name: Mostra una quantitat que representa zzz
    debug: msg="{{ ansible_facts.packages['curl'] | length }}"
```

b) Després de llegir la documentació oficial del mòdul "**service_facts"**, digues què mostrarien a pantalla les diferents tasques del següent playbook i prova-les:

```
- name: Misteri2
  hosts: victimes
  remote_user: pepito
  tasks:
  - name: Recollir dades sobre xxx i afegir-les a l'objecte xxx
    service_facts:
  - name: Mostrar xxx
    debug: msg="{{ ansible_facts }} "
  - name: Mostrar yyy (això és possible gràcies a haver executat abans el mòdul service_facts)
    debug: msg="{{ ansible_facts.services }}"
  - name: Mostrar una quantitat que representa yyy
    debug: msg="{{ ansible_facts.services| length }}"
  - name: Mostrar zzz
    debug: msg="{{ ansible_facts.services['cups.service'] }}"
  - name: Mostra una quantitat que representa zzz
    debug: msg="{{ ansible_facts.services['cups.service'] | length }}"
```

Variables registrades

També es poden utilitzar variables per comprovar si la sortida que un determinat mòdul hagués mostrat a la pantalla d'un terminal és la que s'esperava. Per exemple, suposem que volem escriure una tasca que comprovi si un port està obert o no; això ho podríem aconseguir fent servir el mòdul *shell* (o *command*) i executant la comanda *ss -tln* però hauríem de tenir algun mètode per "capturar" la sortida de la comanda anterior i comprovar si aquesta coincideix amb la sortida desitjada o no. Per realitzar aquest tipus de proves, podem utilitzar la directiva *register* ; aquesta directiva, escrita dins d'una determinada tasca, serveix per indicar el nom de la variable on es guardarà, entre altres dades, la sortida estàndar generada per dita tasca.

Això d'"entre altres dades" significa que, en realitat, la directiva *register* emmagatzema a la variable indicada tota un diccionari JSON que pot incloure diversos camps, els quals poden ser accedits (posteriorment, en una altra tasca següent del mateix play) amb la sintaxis habitual *nomVariable.nomCamp* i entre els quals trobem els següents (la llista de camps concreta dependrà del mòdul executat en cada moment però a https://docs.ansible.com/ansible/latest/reference_appendices/common_return_values.html es poden consultar els més comuns:

**stdout*: El contingut de la sortida estàndard generat per la tasca en qüestió en forma d'una sola cadena
**stdout_lines*: El mateix però en forma de llista, on el contingut de cada línia de la sortida és un element
**stderr*: El contingut de la sortida d'error generat per la tasca en qüestió en forma d'una sola cadena
**stderr_lines*: El mateix però en forma de llista, on el contingut de cada línia de la sortida és un element
**rc*: El codi de retorn de la tasca en qüestió, si aquesta ha executat alguna comanda de terminal
**changed*: Valor booleà que indica si la tasca ha realitzat algun canvi o no
**failed*: Valor booleà que indica si la tasca ha fallat o no
**skipped*: Valor booleà que indica si la tasca ha sigut ignorada o no
**start* i *end*: Dates d'inici i final de la tasca en qüestió
**delta*: Temps transcorregut en l'execució de la tasca (és el resultat de "*end - start*")
**results*: Array que conté les diferents estructures aquí descrites per cadascuna de les tasques que s'hagin executat si aquestes s'han especificat dins d'un bucle

7.-a) Digues, sense provar el playbook següent, què creus que mostrarà el mòdul *debug* a pantalla i seguidament prova'l per comprovar si has encertat:

```
- name: Misteri misteriós
  hosts: victimes
  remote_user: pepito
  tasks:
  - name: Lelele
    shell: "cat /etc/fstab"
    register: tintin
  - name: Lololo
    debug: msg="{{ tintin.stdout_lines }}"
  - name: Lululu
    copy: src="~/{{ tintin.rc }}.txt" dest="~"
```

b) Digues, sense provar el playbook següent, què creus que mostrarà el mòdul *debug* a pantalla i seguidament prova'l per comprovar si has encertat:

```
- name: Misteri misteriós2
  hosts: victimes
  remote_user: pepito
  tasks:
  - name: Leeleelee
    command: df -h
    register: pimpim
  - debug: var=pimpim.stdout_lines
```

NOTA: En el cas de voler mostrar simplement el valor d'una variable i res més (sense cap missatge addicional), en lloc d'escriure *debug: msg="{{ nomVar }}"* es pot escriure com a forma alternativa (i més senzilla), *debug: var=nomVar* ; és el que hem fet al "playbook" anterior
NOTA: Podeu observar també com, de fet, la línia *name* que s'escriu sempre a cada tasca no és obligatòria
NOTA: La diferència principal entre el mòdul "shell" i el mòdul "**command**" és que aquest darrer, en no executar la comanda dins de cap shell, no pot emprar ni cap variable d'entorn ni cap dels operadors *, <, >, |, ; o & (propis del shell)

c) Les variables "registrades" no només tenen sentit per recollir el resultat d'executar mòduls com "shell" o "command". Per comprovar-ho, completa el següent playbook afegint una darrera tasca que utilitzi el mòdul "debug" per tal de mostrar quins són els camps de la variable "tonton" (registrada en realitzar una tasca mitjançant el mòdul oficial "**tempfile**"):

```
- name: A completar
  hosts: victimes
  remote_user: pepito
  tasks:
  - name: Crear una carpeta temporal
    tempfile: state=directory
    register: tonton
```

d) Un altre ús que podem trobar de les variables "registrades" és fent-les servir juntament amb el mòdul "**uri**" ja vist a exercicis anteriors. Per comprovar-ho, completa el següent playbook afegint una darrera tasca que utilitzi el mòdul "debug" per tal de mostrar quins són els camps de la variable "tuntun" (i digues quins són els més rellevants segons el teu parer):

```
- name: A completar2
  hosts: victimes
  remote_user: pepito
  tasks:
  - name: Realitzar una consulta GET
    uri: url=https://api.chucknorris.io/jokes/random
    register: tantan
```

dII) Recorda les opcions del mòdul "uri" per trobar per a què servia l'opció "return_content" i què implica que el seu valor per defecte sigui "no". A partir d'aquí, digues què fa (i què mostra per pantalla) el següent playbook en executar-se. ¿Què veuràs si, en aquest mateix playbook, canvies la URL de El Puig per *https://api.chucknorris.io/jokes/random* ?

```
- name: A veure
  hosts: victimes
  remote_user: pepito
  tasks:
  - name: Consulta
    uri: url="https://elpuig.xeill.net"  return_content=yes
    register: resposta
  - debug: msg="{{ resposta.content }}"
```

NOTA: Una manera alternativa d'obtenir una resposta d'una URL i guardar-la en una variable és fent servir el "lookup" "url" (https://docs.ansible.com/ansible/latest/collections/ansible/builtin/url_lookup.html), del qual en parlarem aviat

Inclusió de "tasks"

Igual que hem vist que es poden incloure variables definides en fitxers independents, també és possible incloure tasques en playbooks que estan definides en fitxers independents. D'aquesta manera, una (o unes) mateixa/es tasca/ques la podrem reutilitzar en diferents playbooks. Per aconseguir això, Ansible ofereix dos mètodes: mitjançant la paraula-clau *import_tasks: nomFitxer.yml* o bé mitjançant el mòdul *ansible.builtin.include_tasks: nomFitxer.yml* (on, en ambdós casos, el contingut de "nomFitxer.yml" serà una llista de tasques, similar per exemple a):

```
- name: Un mòdul
  debug: msg="Executo mòdul 1"
- name: Un altre mòdul
  debug: msg="Executo mòdul 2"
```

La diferència entre fer servir *import_tasks* i *include_tasks* és que el primer mètode és estàtic i el segon dinàmic. És a dir, mitjançant *import_tasks* Ansible preprocessa primer totes les tasques indicades en el fitxer extern i "copia-pega" el seu contingut allà on hi sigui abans d'executar cap tasca del playbook (i això vol dir que el contingut importat mai serà afectat per altres tasques prèvies en el playbook en qüestió). En canvi, mitjançant *include_tasks* Ansible processa les tasques indicades en el fitxer extern a mesura que va executant el playbook "en calent" (i això vol dir que el contingut inclòs podria ser afectat per les tasques que s'hagin executat prèviament dins del playbook; en concret, podrà executar-se o no segons el resultat de les tasques precedents). Una altra diferència és que *include_tasks* es pot escriure dins d'un bucle, aconseguint així que les tasques indicades al fitxer extern s'executin de nou a cada iteració (això amb *import_tasks* no es pot fer).

NOTA: Per saber més detalls sobre les diferències entre ambós mètodes, es pot consultar la taula comparativa: https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse.html#comparing-includes-and-imports-dynamic-and-static-re-use
NOTA: En el cas de voler incloure tasques com a "handlers" (els estudiarem més avall), cal tenir en compte els detalls descrits a https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse.html#re-using-tasks-as-handlers
NOTA: També és possible incloure "playbooks" sencers dins d'un altre "playbook" amb l'instrucció
- import_playbook:"nomPlaybook.yml"
NOTA: Es poden passar variables a tasques importades. Això és necessari, per exemple, si es vol executar el fitxer importat més d'un cop en un playbook amb valors diferents per alguna dada. Per exemple:

```
tasks:
- import_tasks: wordpress.yml
  vars:
    wp_user: timmy
- import_tasks: wordpress.yml
  vars:
    wp_user: alice
```

8.-a) Si tenim un fitxer anomenat "unfitxer.yml" amb el següent contingut...:

```
- name: Lololo
  debug: msg="{{ tintin.stdout_lines }}"
```

a) ...i, a la mateixa carpeta, tenim el playbook següent, digues, sense provar-lo, què creus que es mostrarà a pantalla i seguidament prova'l per comprovar si has encertat:

```
- name: Misteri misteriós
  hosts: victimes
  remote_user: pepito
  tasks:
  - name: Lelele
    shell: "cat /etc/fstab"
    register: tintin
  - import_tasks: unfitxer.yml
```

b) En el playbook anterior, ¿creus que es veurà algun canvi si substitueixes la paraula *import_tasks:* del playbook anterior per la paraula *include_tasks:*? Comprova-ho

"Lookups"

Ansible es capaz de obtener datos de fuentes externas (y guardarlos en variables dentro de un "playbook") mediante diversos "plugins" especiales llamados genéricamente "lookups" (https://docs.ansible.com/ansible/latest/plugins/lookup.html) . Por ejemplo, para almacenar en una variable el contenido de un fichero de texto (llamado "lorem.txt" y ubicado en la misma carpeta donde se encuentre el "playbook" en cuestión) se puede usar el "lookup" "file", así:

```
vars:
  nomVariable: "{{ lookup('file','lorem.txt') }}"
```

Por otro lado, si queremos, por ejemplo, obtener un valor determinado almacenado en un archivo CSV, podemos usar el "lookup" "csvfile". Por ejemplo, para conseguir el valor de la tercera columna (empiezan por 0) de la fila cuyo valor de su primera columna es "hola" (y cuyo delimitador de campos es la coma), se puede hacer así:

```
vars:
  nomVariable: "{{ lookup('csvfile','hola file=lorem.csv delimiter=, col=2') }}"
```

También podemos obtener el valor de una determinada directiva dentro de alguna sección concreta de un archivo INI mediante el "lookup" "ini". Por ejemplo, para conseguir el valor de la directiva "user" presente dentro de la sección "[config]"del archivo "lorem.ini" (ubicado, como siempre, en la misma carpeta donde se encuentre el "playbook" en cuestión) se puede hacer así:

```
vars:
  nomVariable: "{{ lookup('ini','user section=config file=lorem.ini') }}"
```

Si queremos obtener, en cambio, el valor de una determinada variable de entorno existente en la máquina "controladora" (es decir, donde estamos ejecutando Ansible), se puede hacer así:

```
vars:
  nomVariable: "{{ lookup('env','nombreVariableEntorno') }}"
```

Si lo que queremos és obtener la salida por terminal de un determinado comando "ad-hoc" ejecutado en la máquina "controladora", se puede hacer así:

```
vars:
  nomVariable: "{{ lookup('pipe',' cat /etc/fstab') }}"
```

NOTA: Hay muchos más "lookups" disponibles. La lista entera se puede consultar ejecutando el comando *ansible-doc -t lookup -l* y la documentación de un "lookup" particular ejecutando *ansible-doc -t lookup nombreLookup*

9.-a) Si escrius una tasca en un playbook que implementi el mòdul "user" d'aquesta manera, ¿què passaria quan l'executessis?¿Per què és necessari el paràmetre *+%s* de la comanda *date*?

```
- name: Misteri
  user:  name=pepe  expires="{{lookup('pipe','date +%s')}}"
```

b) ¿Quina diferència hi ha entre el que es veuria en executar aquesta tasca ...

```
- name: Misteri2
  debug:  msg="{{lookup('env','HOME')}}"
```

... i el que es veuria executant aquesta altra, la qual fa servir el "fact" *ansible_env*?

```
- name: Misteri2
  debug:  msg="{{ ansible_env.HOME }}"
```

10.-a) Prova el següent playbook i observa la sortida de les dues execucions del mòdul "debug":

```
- name: A veure
  hosts: victimes
  remote_user: pepito
  vars:
     #Les cometes escapades són necessàries per interpretar correctament els dos punts
     var_time: "{{ lookup('pipe', 'date \"+%H:%M:%S\"') }}"
  tasks:
  - debug: var=var_time
  - command: sleep 2
  - debug: var=var_time
```

b) Prova ara aquest altre playbook, observa quina diferència hi ha amb la sortida obtinguda a l'apartat anterior (fixa't en les dates que apareixen!) i dedueix, a partir d'aquí, quina utilitat té fer servir el mòdul oficial "**set_fact**" en lloc de simplement variables.

```
- name: A veure
  hosts: victimes
  remote_user: pepito
  tasks:
  - set_fact: var_time="{{ lookup('pipe', 'date \"+%H:%M:%S\"') }}"
  - debug: var=var_time
  - command: sleep 2
  - debug: var=var_time
```

"Tags"

Etiquetar una tarea permite incluirla, en el momento de ejecutar el "playbook" donde se encuentre definida, en la lista de tareas a ejecutar (o no, según lo que digamos), ignorando el resto de tareas no etiquetadas. Es decir, las etiquetas nos permiten elegir, según nos convenga en cada momento, qué partes del "playbook" querremos ejecutar o no, según estén etiquetadas o no (o al revés). También se pueden realizar composiciones como ejecutar tareas etiquetadas como "A" o "B", pero no "C". Para etiquetar una tarea basta con añadirle la palabra-clave *tags:* indicando una lista de valores (que representan las "etiquetas"), así:

```
- name: Do something really interesting
  debug: msg="Yes this does something really interesting"
  tags: ['interesting', 'awesome']
```

NOTA: Una sintaxis YAML alternativa sería, tal com ya sabemos:

```
- name: Do something really interesting
  debug: msg="Yes this does something really interesting"
  tags:
  - interesting
  - awesome
```

Una vez ya se hayan definido las etiquetas para las tareas de nuestro "playbook", se le puede decir a Ansible que ejecute las tareas etiquetadas con un valor determinado mediante el parámetro *-t*, así: *ansible-playbook myplaybook.yml -t interesting*

NOTA: El "playbook" de ejemplo anteriormente indicado se ejecutará, además de cuando se indique el parámetro *-t interesting* o el parámetro *-t awesome*, cuando, atención, no se haya indicado ningún parámetro *-t* o también cuando se indiquen múltiples valores de etiquetas que no estén definidas excepto como mínimo una (así: *-t boring,interesting* o así: *-t boring -t interesting*).

Para ejecutar las tareas que **no** estén etiquetadas con un valor determinado, se debería usar, además del parámetro *-t*, el parámetro *--skip-tags* Para saber qué etiquetas están definidas en un "playbook" se puede ejecutar el comando: *ansible-playbook miplaybook.yml –list-tags*

11.-Escriu un playbook que contingui dues tasques a executar a les víctimes: la primera ha d'instal.lar un paquet (el que tu vulguis) i la segona ha de posar en marxa el servei Cups. Aquesta segona ha d'estar etiquetada. A partir d'aquí:

a) Executa el playbook sense indicar cap etiqueta. ¿Quines tasques s'executen?

b) Executa el playbook indicant l'etiqueta de 2ª tasca amb el paràmetre *-t*. ¿Quines tasques s'executen ara?

c) ¿Què passa si ara executes el playbook indicant l'etiqueta d'aquesta 2ª tasca amb el paràmetre *--skip-tags*?

"Handlers"

Otra funcionalidad muy interesante de los "playbooks" es la posibilidad de disparar automáticamente tareas en el momento de ejecutar una tarea anterior. Esto se consigue mediante el uso del ítem *notify* de la tarea disparadora y el elemento *handlers*, el cual no deja de ser una "task" que solo se ejecuta si es "notificada". Veamos un ejemplo (sin mucho sentido, pero es solo un ejemplo) donde, al instalar el programa nmap, automáticamente se arranca el servicio Nginx:

```
- name: Playbook d'exemple de handlers
  hosts: all
  remote_user: pepito
  become: yes
  tasks:
  - name: Instal·lar nmap
    package: name=nmap state=present
    notify:
    - Pepe
  handlers:
  - name: Pepe
    service: name=nginx state=started
```

NOTA: Cal tenir en compte que si la tasca que dispara el "handler" no es realitza (o es realitza amb errors), el "handler" no es realitzarà tampoc!

12.- Escriu un playbook que:

a) Instal.li el servidor Apache a les màquines víctimes

b) Que just després d'aquesta instal.lació, mitjançant un "handler", habiliti el servei

c) Que copïi un fitxer "index.html" de la màquina controladora a la carpeta "/var/www/html" de les màquines víctimes

d) Que, just després d'aquesta còpia, mitjançant un altre "handler", posi en marxa el servei

Mòdul "template" i llenguatge Jinja2

Ja coneixem el mòdul "copy", que serveix per copiar fitxers de la màquina controladora a les màquines víctima però aquest mòdul pot ser insuficient si el que volem és escampar fitxers de configuració que necessitin ser personalitzats segons la màquina víctima particular on es copïin. En aquest sentit, Ansible proporciona un altre mòdul oficial anomenat "**template**", el qual permet implementar les anomenades "templates" (plantilles), que no són més que fitxers de text amb extensió ".j2" que representen l'"esquelet" d'un fitxer on hi apareixen "placeholders" (és a dir, "forats" marcats dins del text) que seran substituïts, en una determinada execució d'Ansible, i a cada màquina víctima per separat, pels valors concrets desitjats que convinguin (valors que, en una altra execució podrien ser diferents) .

NOTA: L'extensió ".j2" ve de "Jinja2" (https://jinja.palletsprojects.com), una llibreria Python que defineix un llenguatge de marques específic

Aquesta manera de treballar va molt bé, tal com hem dit, per replicar les mateixes accions d'un "playbook" sobre diverses màquines "víctima" però fent servir per cadascuna d'elles paràmetres diferents. Per exemple, si tenim el següent "playbook"...

```
- name: Playbook d'exemple de templates
  hosts: all
  remote_user: pepito
  vars:
     variable1: "Hola!"
     variable2: "Pepito Pérez"
  tasks:
  - name: Copiar plantilla
    template: src=plantilla.j2 dest=~/resultat.txt
```

... i el contingut de l'arxiu "plantilla.j2" (ubicat a la mateixa carpeta que el "playbook") és...:

```
{{ variable1 }}
Una línia qualsevol
Em dic {{ variable2 }}
```

NOTA: Es poden incloure comentaris (monolínia o multilínia, tant és) dins de qualsevol fitxer "j2" amb la següent sintaxi: *{# un comentari #}* D'altra banda, també es poden incloure condicionals (veurem més endavant com i per què) dins de qualsevol fitxer "j2" amb la sintaxi *{% if condicio %} ... {% endif %}* i bucles *{% for i in llista %} ... {% endfor %}*

... el que passarà és que es copiarà un arxiu anomenat "resultat.txt" a la carpeta personal de l'usuari "pepito" de cada màquina víctima amb el següent contingut (si aquest arxiu ja existís, llavors se sobreescriurà)

```
Hola!
Una línia qualsevol
Em dic Pepito Pérez
```

NOTA: El valor de l'opció *src=* pot ser també una ruta absoluta. D'altra banda, altres opcions interessants del mòdul "template" són *owner=*, *group=* o *mode=*, totalment equivalents a les del mòdul "copy" ja conegudes (recordeu que utilitzar les dues primeres, però, caldrà indicar *become: true*)

13.- a) En la màquina controladora, crea un arxiu anomena "index.html.j2" amb el següent contingut:

```
<!DOCTYPE><html><body>
<h1>{{ ansible_hostname }}</h1><h3>{{ ansible_os_family }}</h3>
<p>{{ apache_test_message }}</p>
</body></html>
```

NOTA: Fixa't com dins de la plantilla, a més de variables que poguem definir al playbook (de les diverses maneres possibles que ja coneixem: via secció "vars" o via secció "vars_files" o via paràmetre *-e*, etc...en aquest cas serà la variable "apache_test_message", definida en una secció "vars"), també es poden indicar "facts" (en aquest cas, "ansible_hostname" i "ansible_os_family")

b) Crea, també a la màquina controladora,

```
- name: Assegurar-se que el servidor Apache està instal·lat i iniciat amb una pàgina web personalitzada
  hosts: all
  remote_user: pepito
  become: yes
  vars:
    apache_test_message: Aquest missatge pot ser canviat
  tasks:
  - package:  name=httpd  state=present
  - template:  src=index.html.j2   dest=/var/www/html/index.html
  - service:  name=httpd  state=started  enabled=yes
```

NOTA: Un altre exemple interessant hagués sigut, en lloc (o a més) de tenir una plantilla per la pàgina inicial de l'Apache, una altra pel seu arxiu de configuració, on hi podríem situar els "placeholders" que necessitéssim per tal de configurar-lo "a la carta". En aquest cas caldria haver tingut en compte d'afegir un "handler" després d'executar el mòdul "template" que servís per reiniciar el servei (ja que si no la nova configuració, tot i que copiada, no s'aplicaria)

c) Accedeix amb el navegador de la màquina real al servidor Apache que hauria d'estar funcionant a la màquina B. ¿Què veus? ¿I si accedeixes al servidor Apache que està funcionant a la màquina C?

d) Repeteix l'apartat anterior però havent canviat primer el valor de la variable "apache_test_message" del "playbook" (i tornant-lo a executar). ¿Quin canvis veus a la pàgina web mostrada?