Ansible (IV)

Bucles

Sovint es voldrà fer moltes coses semblants en una tasca (com crear molts usuaris, instal·lar molts paquets o repetir una petició de xarxa fins que s'obtingui un resultat determinat, per exemple). Per estalviar una mica d'escriptura, les tasques repetides es poden escriure de forma abreujada de la següent manera:

```
- name: Loop playbook example
  hosts: all
  remote_user: usuari
  become: yes
  tasks:
  - name: Add several users (with the same password)
    user:
          name: "{{ item }}"
          password: "{{ '1234' | password_hash('sha512') }}"
          state: present
    loop:
    - testuser1
    - testuser2
```

Es pot veure que s'està usant la paraula-reservada *loop* associada a una determinada tasca, i combinada amb la variable especial *{{ item }}* Els valors de la llista indicada a la secció *loop* es poden escriure fent servir la notació de llista d'elements YAML o bé en forma d'array, així: *loop: ["testuser1", "testuser2"]* De fet, si aquesta llista es definís a la secció "vars" (o a un fitxer extern), es podria fer això també:

```
- name: Loop playbook example 2
  hosts: all
  remote_user: usuari
  become: yes
  vars:
    usuaris: [ "testuser1", "testuser2" ]
  tasks:
  - name: Add several users (with the same password)
    user:
          name: "{{ item }}"
          password: "{{ '1234' | password_hash('sha512') }}"
          state: present
    loop: "{{ usuaris }}"
```

NOTA: És possible crear diversos usuaris amb contrasenyes diferents en un bucle, però hem d'utilitzar les "cerques de diccionari", una construcció Ansible que encara no hem estudiat (ho veurem en els exercicis següents)
NOTA: Alguns mòduls (com "dnf" o "apt") poden agafar llistes directament en les seves opcions: això és més òptim que fer un bucle sobre la tasca (i té menys problemes d'interdependència). Podeu consultar la documentació de cada acció per obtenir més informació, però aquí hi ha un exemple:

```
- name: optimal dnf
  dnf:
    name: "{{list_of_packages}}"
    state: present
```

en comptes de

```
- name: non optimal, slower dnf
  dnf:
    name: "{{item}}"
    state: present
  loop: "{{list_of_packages}}"
```

Acompanyant a la directiva *loop:* també existeix la directiva opcional *loop_control:* , la qual serveix per definir diversos aspectes del comportament intern del bucle, com ara els segons que han de passar entre iteració i iteració (subdirectiva *pause: nº*) o el nom de la variable-comptador que es podrà utilitzar com una variable més a mostrar/utilitzar a les iteracions (subdirectiva *index_var: nomComptador*), entre d'altres. Per exemple, la següent tasca mostraria per pantalla els números "0", "1" i "2" (perquè el comptador comença per 0 sempre), de segon en segon:

```
- name: Bucle
  debug: msg="{{ item }} amb posició {{ comptador  }}"
  loop: ["un","dos","tres"]
  loop_control:
    index_var: comptador
    pause: 1
```

NOTA: A https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html#loops trobareu molta més informació sobre com utilitzar les diverses possibilitats que ofereixen els bucles.

1.-a) Digues què fa (no cal que ho provis si ja ho veus clar) el següent playbook:

```
- name: Loop example
  hosts: all
  remote_user: usuari
  become: yes
  tasks:
  - name: xxx
    copy: src="~/{{ item }}" dest="/etc/skel/{{ item }}"
    loop: ["file1.conf", "file2.conf", "file3.conf"]
```

b) ¿I si cambiem la línia *copy:* ... del playbook anterior per aquesta altra?

```
    file: path="/etc/skel/{{ item }}" owner="root" group="root"
```

c) D'altra banda, ¿què faria aquesta tasca?

```
- name: xxx
  lineinfile: path="/etc/sudoers" insertafter="^root" line="{{ item }}"
  loop:
  - "%admin\tALL=(ALL:ALL)\tALL"
  - "%users\tALL=(ALL:ALL)\tALL"
```

d) Executa el següent playbook i observa el contingut mostrat a pantalla de la variable registrada *pepe* . Veuràs que conté un array anomenat "results" amb un element per cada resultat obtingut en la tasca on s'ha omplert (en aquest cas hauran de ser dos elements perquè la tasca "Populate pepe" s'ha realitzat dos vegades a causa del *loop*). Comprova també que cada element contingui, entre d'altres, la clau "msg" (el valor de la qual és el missatge que ha mostrat el mòdul "debug" a la iteració corresponent):

```
- name: Register and loop example
  hosts: all
  remote_user: usuari
  tasks:
  - name: Populate pepe
    debug: msg="{{ item }}"
    loop: ["one", "two" ]
    register: pepe
  -  name: See pepe's content
    debug: msg="{{ pepe }}"
```

El ítems a recòrrer no només poden ser llistes de valors individuals, sinó també elements de diccionari (és a dir, llistes de parelles<->valors). D'aquesta manera, podrem fer servir la notació *{{ item.nomClau }}* per referir-nos al valor concret de la clau indicada (de l'element recorregut en aquell moment).

2.-a) Digues què fa (no cal que ho provis si ja ho veus clar) el següent playbook:

```
- name: Loop example II
  hosts: all
  remote_user: usuari
  become: yes
  tasks:
  - name: xxx
    copy: src="~/{{ item.src }}" dest="/etc/skel/{{ item.dst }}"
    loop:
    - { src: "file1.conf" , dst: "a.conf" }
    - { src: "file2.conf" , dst: "b.conf" }
    - { src: "file3.conf" , dst: "c.conf" }
```

b) ¿I si canviem la tasca del playbook anterior per aquesta altra?

```
- name: xxx
  file: path="/etc/skel/{{ item.pth }}" owner="{{ item.own }}" group="{{ item.grp }}"
  loop:
  - { pth: "a.conf", own: "root", grp: "root" }
  - { pth: "b.conf", own: "usuari", grp: "usuari" }
  - { pth: "c.conf", own: "manolo", grp: "manolo" }
```

c) ¿Què faria aquesta tasca?

```
- name: xxx
  blockinfile:
      path: "/etc/hosts"
      block: |
        {{ item.ip }} {{ item.name }}
  loop:
  - { name: "host1", ip: "10.10.1.10" }
  - { name: "host2", ip: "10.10.1.11" }
  - { name: "host3", ip: "10.10.1.12" }
```

d) ¿Què faria aquest playbook?

```
- name: Jo no sé
  hosts: victimes
  remote_user: usuari
  become: yes
  vars:
      salt: "{{ lookup('password','/dev/null length=6 chars=ascii_letters') }}"   #Veure nota de baix!
  tasks:
  - name: xxx
    user:
        name: "{{ item.name }}"
        password: "{{ item.pass | password_hash('sha512', salt) }}"
        state: present
    loop:
    - { name: "pepe", pass: "1234" }
    - { name: "ana", pass: "5678" }
    - { name: "fermin", pass: "9012" }
```

NOTA: El "lookup" "password" serveix per generar seqüències de caràcters aleatoris ,amb la longitud i el joc desitjat (en aquest cas, només lletres ASCII). S'ha d'indicar també la ruta d'un fitxer on es guardaria aquesta seqüència generada o bé (com s'ha fet) "/dev/null", si no es vol guardar per res. Notar com el valor de 'salt' canvia a cada usuari perquè el "lookup" es torna a calcular cada vegada (si volguéssim evitar això recordeu que es podria fer servir el mòdul "set_facts").
NOTA: Fixeu-vos, d'altra banda, que no ha calgut escriure "{{" i "}}" al voltant del nom "salt" que apareix dins de *password_hash()* perquè ja hi ha unes claus a l'exterior de tot que fan que no calgui escrire'n més

2BIS.- Crea un arxiu a la màquina controladora anomenat "usuaris.txt" amb el següent contingut...

Username,UID,Nom,Cognom,Grups,Password
mperez,2001,Manolo,Perez,,1234
lgomez,2002,Luisa,Gomez,,5678

... i tot seguit executa el següent "playbook" (ubicat a la mateixa carpeta que l'arxiu anterior). ¿Què passa a les màquines "víctima"?

```
- name: Un play
  hosts: victimes
  #ATENCIÓ: L'usuari remot ha de ser el mateix que l'usuari local de la màquina controladora
  remote_user: usuari
  tasks:
  - name: Llegir l'arxiu CSV
    read_csv: path=usuaris.csv
    #L'objecte "llista_usu" contindrà dins de la seva propietat "list" un array de JSON, on cadascun d'ells
    #correspon a un usuari dels llegits a l'arxiu CSV, i on sa clau és la capçalera del camp en qüestió
    register: llista_usu
    #La línia següent és necessària per indicar que l'arxiu CSV es troba a la màquina controladora
    #Una sintaxi alternativa a aquesta línia "delegate_to:localhost" seria la directiva "local_action:"
   #(veure https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_delegation.html#delegating-tasks )
    delegate_to: localhost
  - name: Crear usuaris
    become: yes
    user:
        name: "{{ item.Username }}"
        uid: "{{ item.UID }}"
        comment: "{{ item.Nom }} {{ item.Cognom }}"
        groups: "{{ item.Grups }}"
        append: yes
        password: "{{ item.Password | password_hash('sha512') }}"
        state: present
    loop: "{{ llista_usu.list }}"
```

3.-a) Consulta la documentació oficial del mòdul "**find**" per saber què faria aquest trio de tasques (les pots provar, si vols):

```
- name: First task
  find:  paths="/var/log" recurse=yes patterns="*.log,*.log.*" file_type="file" size=1m age=5d
  register: lerele
  become: yes
- name: Second task
  debug: msg="{{ lerele.files }}"
- name: Third task
  lineinfile:  path="{{ item.path }}"  line="XXX"
  loop: "{{ lerele.files }}"
  become: yes
```

NOTA: Noteu com *{{ lerele.files }}* no ve precedit de cap guió ni claudàtors perquè ella mateixa ja és la llista!!
NOTA: recordar que lineinfile añade al final si no hay regexep

aII) ¿Què faria aquest playbook (prova'l si no ho veus clar.)?

```
- name: A veure què passa
  hosts: localhost
  remote_user: usuari
  tasks:
  - name: Find CSV in Downloads
    find:
        paths: "~/Downloads"
        recurse: no
        patterns: '*.csv,*.CSV'
        register: result
 - name: Remove CSV files
   file:
        path: "{{ item.path }}"
        state: absent
   loop: "{{ result.files }}"
```

NOTA: El playbook anterior es podria executar com a tasca programada (amb un "timer" Systemd, per exemple) per tal de netejar la carpeta "Baixades" local (..fixa't en quina és la màquina "víctima") regularment.

b) ¿Què faria aquest playbook (prova'l si no ho veus clar)?

```
- name: Veiem què passa
  hosts: localhost
  remote_user: usuari
  tasks:
  - name: xxx
    lineinfile:
        path: hola.txt
        line: "{{ ansible_hostname }} --- {{ item.device }} {{ item.mount }} {{ item.fstype }}"
        create: yes
    loop: "{{ ansible_mounts }}"
    delegate_to: localhost
```

NOTA: Noteu com el fact *{{ ansible_mounts }}* no ve precedit de cap guió ni claudàtors perquè ell mateix ja és la llista!!

bII) Observa el contingut de l'arxiu "hola.txt" la primera vegada que executes el playbook anterior i després d'executar-lo unes quantes vegades més. ¿Ha canviat? ¿Per què? Si ara afegim el valor *{{ item.size_available }}* després de *{{ item.fstype }}* i executem de nou varis cops el playbook, ¿creus que ara apareixeran noves línies a l'arxiu "hola.txt" cada vegada o no? ¿Per què?

bIII) ¿Per a què serviria executar aquesta tasca?

```
- debug:  msg= "Mount Point {{item.mount}} is at {{item.block_used/item.block_total*100}} percent "
  loop: "{{ansible_mounts}}"
```

En alguna ocasión nos irá bien reintentar una tarea hasta que una determinada condición se cumpla. Para ello deberemos usar la opción *register* junto con el bucle *until*. Como ya sabemos, la opción *register*, escrita dentro de una determinada tarea, sirve para indicar en qué variable se guardará el resultado de dicha tarea; lo que hace la opción *until* es comprobar que ese resultado cumpla una determinada condición para saber si se ha de seguir reintentando de nuevo la ejecución de dicha tarea (o ya no y entonces seguir adelante con el resto del playbook). Para usar correctamente el bucle *until* antes hay que conocer, no obstante, cuál es el tipo de contenido que puede recibir la variable indicada en *register* tras haberse ejecutado la tarea para poder entonces escribir una condición en consecuencia: sobre las condiciones posibles hablaremos en el próximo apartado

4.-a) Dedueix què faria el següent playbook (i prova'l després a veure si estaves en el cert), tenint en compte que el subobjecte *stdout* de l'array *results* en aquest cas representa el missatge que hauria aparegut a la sortida estàndar (pantalla) en realitzar-se la tasca en qüestió en un terminal, i que el mètode *find* d'aquest subobjecte busca una determinada cadena en aquesta sortida (retornant -1 si no la troba):

```
- name: Until playbook example
  hosts: localhost
  remote_user: usuari
  tasks:
  - name: Repeat
    shell: echo "És xulo passejar"
    register: pepe
    until: pepe.stdout.find("És xula la platja") != -1
    retries: 5
    delay: 10
```

NOTA: Les opcions *retries* i *delay* són opcionals perquè ambdues tenen valors per defecte predefinits (3 i 5, respectivament). La primera limita la repetició de la tasca al màxim indicat (per evitar així bucles infinits); la segona indica el temps entre intent i intent (en segons)
NOTA: La variable registrada tindrà una clau anomenada "attempts" que tindrà el nº de reintents de la tasca

b) ¿Què fa aquest playbook (prova'l)?

```
- name: Until playbook example
  hosts: localhost
  remote_user: usuari
  tasks:
  - name: Pause play until a URL is reachable from this host
    uri:  url="http://1.2.3.4/lalala"
    register: pepe
    until: pepe.status == 200
    retries: 3
    delay: 2
```

bII) ¿I aquest (prova'l també)? Raona el resultat obtingut.

```
- name: Another until playbook example
  hosts: localhost
  remote_user: usuari
  tasks:
  - name: Pause play until a URL is reachable from this host
    uri:  url={{ item }} follow_redirects=none
    loop:
    - "http://www.google.com"
    - "http://www.github.com"
    register: pepe
    until: pepe.status == 200
    retries: 3
    delay: 2
```

Condicionales

A menudo la ejecución de una tarea depende del valor de una variable, un "fact" o el resultado de una tarea anterior. Para definir qué condición/nes se han de cumplir para ejecutar una determinada tarea se puede usar la directiva *when:*. Este ítem tiene como valor una condición del tipo:

nombreVariableOFact operador valor

... donde *nombreVariableOFact* **no** lleva las llaves {{ y }} alrededor y *operador* puede ser ==, != , < , > , => o <=. Por ejemplo: *when: ansible_distribution == "Debian"* También se pueden escribir, por otro lado, las siguientes condiciones especiales, las cuales sirven respectivamente para comprobar si está definida (o no) o si vale *True* (o *False*) una determinada variable o "fact":

nombreVariableOFact is defined
nombreVariableOFact is undefined
nombreVariableOFact
not nombreVariableOFact

NOTA: En relación a los valores posibles de las variables booleanas (si pueden valer *true* o *True*, *false* o *False*, *yes* y *no...)* recomiendo leer https://stackoverflow.com/questions/47877464/how-exactly-does-ansible-parse-boolean-variables El resumen es que si estos valores se definen en YAML se aceptan la mayoría de formas pero si se definen desde parámetros de terminal se ha de usar la forma Pythónica

Como hemos dicho, también se pueden definir condiciones a partir del estado final de una tarea anterior, que es algo que se utiliza mucho. Para ello deberemos usar el ítem *register* que ya hemos estudiado anteriormente. Como ya sabemos, este ítem, escrito dentro de una determinada tarea sirve para indicar en qué variable se guardará el resultado de dicha tarea. Si entonces utilizamos esa variable en un elemento *when* de una tarea posterior del mismo playbook, estaremos definiendo una condición que, basándose en el valor recogido en esa variable, servirá para decidir si se ejecuta (o no), esa tarea posterior. En este sentido, existen un conjunto de condiciones concretas predefinidas relacionadas con el estado de finalización de una tarea anterior que pueden usarse para ejecutar tareas posteriores dependiendo de si esa tarea anterior ha finalizado correctamente o no. Para ello debemos usar la variable indicada en el ítem *register* de esa tarea anterior, así:

nombreVariableRegistrada is succeeded
nombreVariableRegistrada is failed
nombreVariableRegistrada is changed (para confirmar si el resultado de la tarea cambia el sistema)

Otras condiciones que se pueden usar en una tarea para definir si se realizará o no son:

nombreVariable is directory
nombreVariable is file
nombreVariable is link
nombreVariable is exists
nombreVariable is mount
nombreVariable is samefile("/ruta/fitxer")
nombreVariable is match("ExprRegACoincidirAmbValorVariableTotalment")
nombreVariable is search("ExprRegACoincidirAmbValorVariableParcialment")

En cualquier caso, si quisiéramos incluir varias condiciones en una línea *when* podemos usar las palabras clave *and* o *or* junto con el uso de paréntesis, así por ejemplo: *when: (ansible_distribution == "CentOS" and ansible_distribution_major_version == "6") or (ansible_distribution == "Debian" and ansible_distribution_major_version == "7")*

NOTA: També es poden escriure múltiples condicions (les quals estaran "unides", això sí, amb un implícit "and" a no ser que al final d´alguna de les condicions s'escrigui explícitament l'operador "or") fent servir la sintaxi de llista YAML, així (en aquest exemple s'aprofita també per mostrar l'ús de filtres i també de l'operador "in"):

```
when:
- ansible_distribution in ["Fedora","Ubuntu"] or
- ansible_distribution | length > 5
```

NOTA: Per més informació https://docs.ansible.com/ansible/latest/user_guide/playbooks_conditionals.html
i https://docs.ansible.com/ansible/latest/user_guide/playbooks_tests.html

5.- a) ¿Què fa aquest "playbook"? Prova'l

```
- name: Exercice 5a
  hosts: all
  remote_user: usuari
  tasks:
  - name: xxx
    debug:  msg="System {{ ansible_hostname }} has gateway {{ ansible_default_ipv4.gateway }}"
    when: ansible_default_ipv4.gateway is defined
```

b) ¿Què fa aquest "playbook"? Prova'l

```
- name: Exercice 5b
  hosts: all
  remote_user: usuari
  tasks:
  - name: yyy
    shell: "cat /etc/os-release"
    register: resultat
  - name: zzz
    debug: msg="os-release file contains the word 'workstation'"
    when: resultat.stdout.find('workstation') != -1
```

NOTA: Una alternativa més sofisticada al mètode *find()* anterior seria aplicar el filtre *regex_search* a la sortida de la comanda executada, així, en concret: *when: resultat.stdout | regex_search('workstation')* (recordeu que a *when:* no s'escriuen les dobles claus i que *regex_search* retorna un valor de tipus "NoneType" -equivalent a *False*- si no troba cap cadena coincident a la indicada; qualsevol altre valor retornat -que serà una cadena coincident serà equivalent a *True*-)

c) ¿Què fa aquest "playbook"? Prova'l

```
- name: Exercice 5c
  hosts: all
  remote_user: usuari
  tasks:
   - name: www
     shell: "ls /opt"
     register: resultat
   - name: jjj
     debug: msg="Directory is empty"
     when: resultat.stdout == ""
```

d) ¿Què fa aquest "playbook"? Prova'l diversos cops seguits i raona el seu diferent comportament

```
- name: Exercici 5d
  hosts: all
  remote_user: usuari
  tasks:
  - name: kkk
    package: name=nmap state=present
    become: yes
   register: resultat
  - name: lll
    debug: var=resultat
    when: resultat is changed
```

dII) Canvia la (darrera i única) condició del playbook anterior per a què sigui ara la següent: *when: resultat is succeeded* i torna a executar aquest playbook varis cops seguits. ¿Notes alguna diferència de comportament entre aquests diversos cops? Per què?

e) Com a valor de *when* es pot indicar una llista de múltiples condicions; en aquest cas, s'hauran de complir totes (és a dir, es fa un AND) per a què es consideri que es pot executar la tasca pertinent. Sabent això, ¿què creus que faria aquesta tasca d'un determinat playbook?

```
tasks:
- name: Mistery
  command: "/sbin/shutdown -t now"
  when:
  - ansible_distribution == "CentOS"
  - ansible_distribution_major_version == "8"
```

NOTA: Recordeu que també podríem haver usat la sintaxi alternativa següent per indicar els "facts" sota la secció *when:*

```
- ansible_facts['distribution'] == "CentOS"
- ansible_facts['distribution_major_version'] == "8"
```

f) ¿Què passa si executes aquesta comanda *ansible-playbook -e "miflag=False" hola.yml* , on "hola.yml" és un playbook amb el següent contingut? ¿I si executes *ansible-playbook -e "miflag=True" hola.yml* ?

```
- name: When playbook with variables
  hosts: all
  remote_user: usuari
  vars:
    unaflag: "{{ miflag }}"
  tasks:
  - shell: "echo ' Hola!'  > ~/flag.txt"
    when: unaflag
```

g) Si es combina *when* amb *loop*, cal tenir en compte que *when* es processa per separat per cada ítem del bucle. Sabent això, dedueix què faria aquesta tasca d'un determinat playbook (i comprova-ho per veure si estaves en el cert):

```
- name: Mistery
  command: "echo {{ item }}"
  loop: [ 0, 2, 4, 6, 8, 10 ]
  when: item > 5
```

h) ¿Què faria aquesta tasca?

```
- name: xxx
  copy: content="<html><body>Hola</body></html>" dest="/var/www/html/index.html"
  when: ansible_hostname == "mwiapp01"
```

i) ¿Què faria aquest "play"?

```
- name: one
  package_facts:
- name: two
  debug: msg="Nmap no està instal·lat"
  when: "'nmap' not in ansible_facts.packages"
```

j) Consulta la documentació oficial del mòdul "**stat**" (sobre tot en els valors que retorna dins de la variable registrada) i digues què faria aquest "play":

```
- name: one
  stat:  path="~/one.txt"
  register: mivar
- name: two
  file: path= "~/one.txt"  state=absent
  when: mivar.stat.islnk is defined
```

k) ¿Què faria aquest "playbook"?

```
- name: Complete when playbook example
  hosts: all
  remote_user: usuari
  become: yes
  vars:
    docroot: /var/www/html/public
  tasks:
  - name: Add Nginx Repository
    apt_repository: repo='ppa:nginx/stable' state=present
    when: ansible_os_family == "Debian"
    register: ppaadded
  -  name: Install Nginx
    apt: name=nginx state=installed update_cache=yes
    when: ppaadded is succeeded
    register: nginxinstalled
    notify:
        - Start Nginx
  -  name: Create Web Root
    when: nginxinstalled is succeeded
    file: path={{ docroot }} mode=775 state=directory owner=www-data group=www-data
    notify:
        - Reload Nginx
  handlers:
  - name: Start Nginx
    service: name=nginx state=started
  - name: Reload Nginx
    service: name=nginx state=reloaded
```

Ansible permet definir què significa "error" en cada tasca mitjançant el condicional *failed_when:*. Aquesta construcció indica que si es produeix la condició especificada, es provocarà automàticament un "error", de manera que l'execució del "playbook" s'aturarà a la màquina infractora.

> NOTA: Cal tenir en compte (a l'igual que passa amb la directiva *when:*) que si hi ha múltiples condicionals *failed_when:* , aquests s'uneixen amb un implícit "i" (és a dir, la tasca només falla quan es compleixen totes les condicions); si es vol provocar un error quan es compleixi alguna de les condicions, s'han de definir les condicions en una cadena amb un operador "or" explícit.

6.- a) ¿Què fa aquest "playbook"? Prova'l

```
- name: Pirripipi
  hosts: all
  remote_user: usuari
  tasks:
  - shell: "ps -e"
    register: r
    failed_when: "'httpd' not in r.stdout"
  - debug: msg="Observa si aquesta frase és visible o no i dedueix per què"
```

b) ¿Què fa aquest "playbook"? Prova'l

```
- name: Perrepepe
  hosts: all
  remote_user: usuari
  vars:
    files_to_check: ["file1.txt", "file2.txt", "file3.txt"]

  tasks:
  - stat: path="{{ item }}"
    loop: "{{ files_to_check }}"
    register: file_check
    failed_when: file_check.stat.exists == False
  - debug: msg="Observa si aquesta frase és visible o no i dedueix per què"
```

c) ¿Què fa aquest "playbook"? Comprova el sentit de totes les comandes encanonades que apareixen i prova'l

```
- name: Parrapapa
  hosts: all
  remote_user: usuari
  tasks:
  - name: Inspect
    shell: "df -h /opt | sed -E '1d;s/ +/ /g' | cut -d' ' -f 4 | cut -d G -f1"
    register: space
    failed_when: "space.stdout| float < 4"
```

NOTA: Fixeu-vos que a la condició hem utilitzat el filtre *"|* ***float****"* per canviar el tipus de dada de cadena a decimal i així poder fer la comparació. Altres filtres similars serien *"|* ***int****"* (per canviar a sencer) o "| ***bool***" (per canviar a booleà)

cII) ¿I aquesta tasca (la pots afegir al "playbook" anterior?

```
  - name: Inspect2
    shell: "cat /proc/meminfo | grep -i memtotal | sed -E 's/ +/ /g' | cut -d' ' -f 2"
    register: memory
    failed_when: "memory.stdout| int < 8000000"
```

d) ¿Què fa aquest "playbook"? Prova'l

```
- name: Parrapapa
  hosts: all
  remote_user: usuari
  tasks:
  - uri:
      url: http://www.phoronix.com
      return_content: yes
   register: page
    failed_when: "'DANGER' in page.content"
```

També és possible, gràcies al mòdul oficial **fail** d'Ansible, forçar la fallada d'una tasca determinada, i amb l'ajuda de la clàusula *when:* estàndard, fer-ho només quan es compleixi una determinada condició. D'aquesta manera s'obtindria un resultat similar a la clàusula *failed_when:* emprada a l'exercici anterior.

7.-a) ¿Què fa aquest "playbook"? Prova'l

```
- name: Provocar fallada
  hosts: all
  remote_user: usuari
```

```
tasks:
- name: Obtenir llista de serveis del sistema
  service_facts:
- name: Comprovar si el servidor MariaDB està instal·lat
  fail: msg="MariaDB no està instal·lat"
  when: ansible_facts.services["mariadb.service"] is not defined
```

b) ¿Què fa aquest "playbook"? Prova'l

```
- name: Provocar fallada interactiva
  hosts: all
  remote_user: usuari
  vars_prompt:
    name: "confirmation"
    prompt: "Estàs segur de reiniciar? Respon amb 'SI'"
    default: "NO"
    private: no
  tasks:
  - name: Comprovar confirmació
    fail: msg="La confirmació d'execució del playbook ha fallat"
    when: confirmation != "SI"
```

Exemples grans a partir de tot el que hem vist fins ara

8.-a) ¿Quines tasques fa aquest "playbook": https://raw.githubusercontent.com/do-community/ansible-playbooks/master/setup_ubuntu1804/playbook.yml ?

NOTA: Hi ha dos mòduls al "playbook" anterior que no hem estudiat i que pots obviar si vols, els mòduls "**authorized_key**" (relacionat amb la gestió de SSH) i "**ufw**" (relacionat amb la gestió del tallafocs a sistemes Ubuntu).

b) ¿Quines tasques fa aquest "playbook": https://raw.githubusercontent.com/do-community/ansible-playbooks/master/apache_ubuntu1804/playbook.yml ?

c) ¿Quines tasques fa aquest "playbook": https://raw.githubusercontent.com/do-community/ansible-playbooks/master/lamp_ubuntu1804/playbook.yml ?

NOTA: Hi ha un parell de mòduls al "playbook" anterior que no hem estudiat relacionats amb la gestió del servidor MySQL (concretament, els mòduls "**mysql_user**" i "**mysql_db**"), però el seu significat és obvi

d) ¿Quines tasques fa aquest "playbook": https://raw.githubusercontent.com/do-community/ansible-playbooks/master/wordpress-lamp_ubuntu1804/playbook.yml ?

NOTA: Tots els "playbooks" anteriors s'han obtingut de https://github.com/do-community/ansible-playbooks

e) ¿Quines tasques fa aquest "playbook":https://raw.githubusercontent.com/ansible/ansible-examples/master/language_features/postgresql.yml? (fa servir algun mòdul no vist a classe com **postgresql_db, postgresql_user** i **postgresql_privs** però el seu significat és obvi)

NOTA: Aquest i molts altres "playbooks" d'exemple es poden trobar a https://github.com/ansible/ansible-examples/