1A

a) La commande `ansible -m ping -i "H1,H2,R1,R2" all` vérifie que les machines précisées sont joignables, cela est réalisé à l'aide d'un ping ssh.

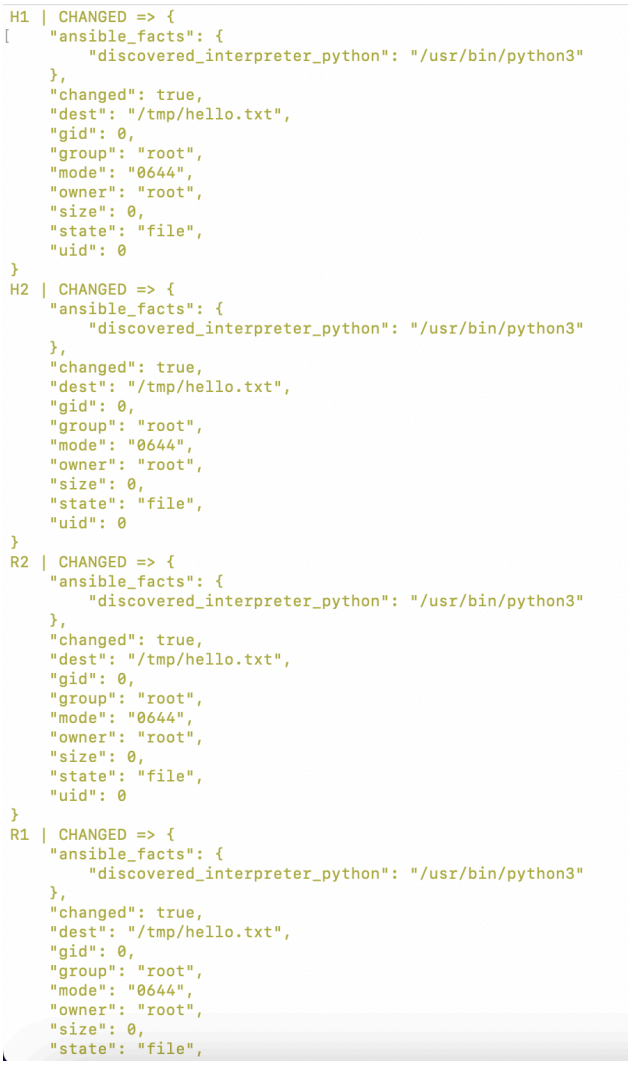
```
(base) badr@MacBook-Air-de-Badr Virtualisation % ansible -m ping -i "H1,H2,R1,R2" all

R2 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
H2 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
R1 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
H1 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
```

b) ansible all -i 'H1,H2,R1,R2' -m command -a 'uptime'

```
(base) badr@MacBook-Air-de-Badr Virtualisation % ansible all -i 'H1,H2,R1,R2' -m command -a 'uptime'
R1 | CHANGED | rc=0 >>
 07:26:32 up 16 min,  2 users,  load average: 0.00, 0.00, 0.00
H1 | CHANGED | rc=0 >>
 07:26:32 up 16 min,  2 users,  load average: 0.00, 0.00, 0.00
R2 | CHANGED | rc=0 >>
 07:26:32 up 16 min,  2 users,  load average: 0.00, 0.00, 0.00
H2 | CHANGED | rc=0 >>
 07:26:32 up 16 min,  2 users,  load average: 0.00, 0.00, 0.00
```

c)Donner la commande ad-hoc qui va créer le fichier `/tmp/hello.txt` sur toutes les machine : ansible all -i 'H1,H2,R1,R2' -m ansible.builtin.file -a 'path=/tmp/hello.txt state=touch'



d)Quelle est la différence entre les modules `command`, `shell` et `raw` ? Expliquez et donnez des exemples.

Command : pour exécuter des commandes simples sans interprétation de shell.
Shell : pour utiliser les fonctionnalités complètes du shell, y compris la manipulation des variables, les redirections, et autres.
Raw : pour les systèmes sans Python ou pour des commandes très basiques qui n'ont pas besoin de traitement. Exécuter directement sur et depuis la machine

2A

3.Les hôtes et les routeurs devront être séparés en deux groupes dans votre inventaire Ansible : `routers` et `hosts`.

```
[hosts]
H1 ansible_host=H1 ansible_port=2046 ansible_user=root
H2 ansible_host=H2 ansible_port=2051 ansible_user=root

[routers]
R1 ansible_host=R1 ansible_port=2038 ansible_user=root
R2 ansible_host=R2 ansible_port=2042 ansible_user=root
```

4. Tout changement dans la configuration IP devra redémarrer le service systemd `networking`, qui est un wrapper au-dessus d'ifup/ifdown.

Lors de l'implémentation de la configuration de chaque hosts, à l'aide de la balise "notify", on notifie qu'un redémarrage du réseau est nécessaire pour appliquer les changements. Elle ajoute le handler du même nom dans la liste des handlers à exécuter.

```
tasks:
  - name: Configuration des interfaces réseau pour les hôtes et routeurs
    template:
      src: "/Users/badr/Desktop/Virtualisation/Ansible/templates/template_interfaces.j2"
      dest: "/etc/network/interfaces"
    notify: redemarrage reseau
```

Le handler permet de redémarrer la configuration suivant, il se présente de la façon suivante:

```
handlers:
  - name: redemarrage reseau
    systemd:
      name: networking
      state: restarted
      daemon_reload: yes
```

5. A l'aide des balise notify, cela rend notre implémentation idempotente. Cela signifie que si il n'y a pas de changement dans la configuration Ansible ne réécrit pas les fichiers et ne redémarre pas les services.

6.

Pour réaliser mon fichier de configuration d'interfaces des différents hôtes, j'ai tout d'abord réaliser un template en jinja2 afin d'avoir un template générique:

```
# /etc/network/interfaces - Configuration réseau pour {{ inventory_hostname }}
auto lo
iface lo inet loopback

{% for iface_name, iface_attrs in interfaces[inventory_hostname].items() %}
# Parcourir le dictionnaire "interfaces" contenant la config
auto {{ iface_name }}
iface {{ iface_name }} inet static
    address {{ iface_attrs.ip_address }}
    netmask {{ iface_attrs.netmask }}
{% if iface_attrs.gateway is defined %}
    gateway {{ iface_attrs.gateway }}
{% endif %}
{% if iface_attrs.route is defined %} #Si des routes sont a spécifier, on les applique
    up /sbin/ip route add {{ iface_attrs.route.destination }} via {{ iface_attrs.route.gateway }}
    down /sbin/ip route del {{ iface_attrs.route.destination }} via {{ iface_attrs.route.gateway }}
{% endif %}
{% endfor %}
```

Ensuite dans le playbook, j'ai instancier des variables pour le nom des interfaces, l'adresse IP, le masque et la passerelle par défaut, indiqué dans le champ vars de la task nommé "Configurer les interfaces réseau des hôtes et routeurs". Elle se présente comme suit :

```yaml
---
- name: Configurer les interfaces réseau des hôtes et routeurs
 hosts: all
 become: yes
 vars:
   interfaces:
     H1:
       eth0:
         interface_name: eth0
         ip_address: 1.0.0.3
         netmask: 255.255.255.0
         gateway: 1.0.0.1
     H2:
       eth0:
         interface_name: eth0
         ip_address: 3.0.0.3
         netmask: 255.255.255.0
         gateway: 3.0.0.2
     R1:
       eth1:
         interface_name: eth1
         ip_address: 1.0.0.1
         netmask: 255.255.255.0
       eth0:
         interface_name: eth0
```

```
        ip_address: 2.0.0.1
        netmask: 255.255.255.0
        route:
          destination: 3.0.0.0/24
          gateway: 2.0.0.2
    R2:
      eth0:
        interface_name: eth0
        ip_address: 2.0.0.2
        netmask: 255.255.255.0
      eth1:
        interface_name: eth1
        ip_address: 3.0.0.2
        netmask: 255.255.255.0
        route:
          destination: 1.0.0.0/24
          gateway: 2.0.0.1
```

Cela permet de créer le fichier de configuration réseau des différentes machines. Exemple pour la machine H1:

```
root@H1:~# cat /etc/network/interfaces
# /etc/network/interfaces - Configuration réseau pour H1
auto lo
iface lo inet loopback

# Parcourir le dictionnaire "interfaces" contenant la config
auto eth0
iface eth0 inet static
    address 1.0.0.3
    netmask 255.255.255.0

    gateway 1.0.0.1
```

2B

Pour réaliser cela, j'ai ajouté deux task dans le playbook afin de vérifier, d'un côté si H1 pouvait pinger H2, et de l'autre si H2 pouvait faire de même. Les tasks se présente comme suit:

```
- name: Vérifier la connectivité entre H1 et H2
 hosts: H1
 tasks:
   - name: Pinguer H2 depuis H1
     command: ping -c 2 3.0.0.3
     register: ping_result
     ignore_errors: true

   - name: Afficher le résultat du ping
     debug:
       msg: "H1 peut{{ ' ' if ping_result.rc == 0 else ' pas ' }}pinguer H2"

- name: Vérifier la connectivité entre H2 et H1
 hosts: H2
 tasks:
   - name: Pinguer H1 depuis H2
     command: ping -c 2 1.0.0.3
     register: ping_result
     ignore_errors: true

   - name: Afficher le résultat du ping
     debug:
       msg: "H2 peut{{ ' ' if ping_result.rc == 0 else ' pas ' }}pinguer H1"
```

Le principe est simple, on exécute la commande ping sur chaque host en direction de son vis-à-vis, on affiche ensuite le résultat à l'aide d'un message de début en fonction du retour récupéré dans "*ping_result*".

Voici le résultat obtenue en cas de succès :

```
PLAY [Vérifier la connectivité entre H1 et H2] ***************

TASK [Gathering Facts] ***************************************
ok: [H1]

TASK [Pinguer H2 depuis H1] **********************************
changed: [H1]

TASK [Afficher le résultat du ping] **************************
ok: [H1] => {
    "msg": "H1 peut pinguer H2"
}

PLAY [Vérifier la connectivité entre H2 et H1] ***************

TASK [Gathering Facts] ***************************************
ok: [H2]

TASK [Pinguer H1 depuis H2] **********************************
changed: [H2]

TASK [Afficher le résultat du ping] **************************
ok: [H2] => {
    "msg": "H2 peut pinguer H1"
}
```

et en cas d’erreur:

```
PLAY [Vérifier la connectivité entre H1 et H2] ***************

TASK [Gathering Facts] ***************************************
ok: [H1]

TASK [Pinguer H2 depuis H1] **********************************
fatal: [H1]: FAILED! => {"changed": true, "cmd": ["ping", "-c", "2", "3.0.0.3"], "delta": "0:00:00.003627", "end": "2024-05-06 18:08:10.619321", "msg":
"non-zero return code", "rc": 2, "start": "2024-05-06 18:08:10.615694", "stderr": "ping: connect: Network is unreachable", "stderr_lines": ["ping: conne
ct: Network is unreachable"], "stdout": "", "stdout_lines": []}
...ignoring

TASK [Afficher le résultat du ping] **************************
ok: [H1] => {
    "msg": "H1 peut pas pinguer H2"
}

PLAY [Vérifier la connectivité entre H2 et H1] ***************

TASK [Gathering Facts] ***************************************
ok: [H2]

TASK [Pinguer H1 depuis H2] **********************************
fatal: [H2]: FAILED! => {"changed": true, "cmd": ["ping", "-c", "2", "1.0.0.3"], "delta": "0:00:00.003737", "end": "2024-05-06 18:08:13.076587", "msg":
"non-zero return code", "rc": 2, "start": "2024-05-06 18:08:13.072850", "stderr": "ping: connect: Network is unreachable", "stderr_lines": ["ping: conne
ct: Network is unreachable"], "stdout": "", "stdout_lines": []}
...ignoring

TASK [Afficher le résultat du ping] **************************
ok: [H2] => {
    "msg": "H2 peut pas pinguer H1"
}
```

3A,B,C

J’ai tout d’abord commencé par installer les packages nginx et wireguard sur les hôtes. J’ai ensuite générer les clés privé de chaque hôte à l’aide de la commande *wg genkey,*pour H1 :
root@H1:/etc/wireguard# sudo cat /etc/wireguard/private.key | wg pubkey | sudo tee /etc/wireguard/public.key
root@H1:/etc/wireguard# ls
private.key public.key

ainsi que H2:
root@H2:/etc/wireguard# sudo cat /etc/wireguard/private.key | wg pubkey | sudo tee /etc/wireguard/public.key
root@H2:/etc/wireguard# ls
private.key public.key

Ensuite j’ai créé deux template jinja2 pour la création du fichier de configuration wg0.conf dans chaque hôte respectif. Voilà à quoi ressemble ces dernières :

```
# Fichier de configuration WireGuard pour le tunnel wg0 sur H1
[Interface]
PrivateKey = {{ h1_private_key }}
ListenPort = {{ wireguard_listen_port_h1 }}
Address = {{ h1_address }}/24

[Peer]
PublicKey = {{ h2_public_key }}
AllowedIPs = {{ h2_address }}/32
Endpoint = {{ h2_public_ip }}:{{ wireguard_listen_port_h2 }}
```

```
# Fichier de configuration WireGuard pour le tunnel wg0 sur H2
[Interface]
PrivateKey = {{ h2_private_key }}
ListenPort = {{ wireguard_listen_port_h2 }}
Address = {{ h2_address }}/24

[Peer]
PublicKey = {{ h1_public_key }}
AllowedIPs = {{ h1_address }}/32
Endpoint = {{ h1_public_ip }}:{{ wireguard_listen_port_h1 }}
```

Puis, pour terminer, j’ai créé un playbook nommé playbook-config-wireguard.yaml qui permet l’instanciation des fichier des configuration dans les hôtes en fonction des clés et configuration adéquate.

Voici les différentes sorties suite à l'exécution du playbook:

```
PLAY [Configuration du tunnel WireGuard entre H1 et H2]
*****************************************************************************************

TASK [Gathering Facts]
*********************************************************************************************************************************************

ok: [H2]
ok: [H1]
TASK [Supprimer la route par défaut si elle existe]
*****************************************************************************************

changed: [H1]
changed: [H2]
TASK [Obtenir une nouvelle adresse IP avec dhclient]
*****************************************************************************************

changed: [H1]
changed: [H2]
TASK [Mettre à jour les paquets disponibles]
*********************************************************************************************

changed: [H1]
changed: [H2]
TASK [Installer le package WireGuard]
*************************************************************************************************

changed: [H2]
changed: [H1]
TASK [Installer le package Nginx]
*************************************************************************************************

changed: [H2]
changed: [H1]
TASK [Générer le fichier de configuration WireGuard pour H1]
*****************************************************************************

skipping: [H2]
changed: [H1]
TASK [Générer le fichier de configuration WireGuard pour H2]
*****************************************************************************

skipping: [H1]
changed: [H2]
RUNNING HANDLER [restart wireguard]
*********************************************************************************************************

changed: [H2]
changed: [H1]
PLAY RECAP
*********************************************************************************************************************************************
*
H1                 : ok=13   changed=11   unreachable=0    failed=0    skipped=1    rescued=0    ignored=0
H2                 : ok=13   changed=11   unreachable=0    failed=0    skipped=1    rescued=0    ignored=0
```

Donc dans l'ordre, le playbook supprime la route par défaut si elle existe, puis en ajoute une nouvelle à l'aide de la commande dhclient -v mgmt0 afin d'avoir un accès à internet. Puis, il télécharge les services Nginx et Wireguard sur les machines. Et pour terminer, il crée les configurations Wireguard dans les machines hôtes et relance le service Wireguard. A noter, que le fichier vault avec les clefs privé et public doit déjà être créé pour le bon fonctionnement du playbook.

Fichier de configuration Wireguard sur l'hôte H1:

```
root@H1:/etc/wireguard# cat wg0.conf
# Fichier de configuration WireGuard pour le tunnel wg0 sur H1
[Interface]
PrivateKey = uHLswVcPZNYFXMvZd9fF1GFFYcIa9HcDFz8kU8+XpEQ=
ListenPort = 51820
Address = 10.0.0.1/24
[Peer]
PublicKey = Z35T+X+RUfQwX7f6jlwM7WWLrFG8RVwBcw0Agbtt+kM=
AllowedIPs = 10.0.0.2/32
```

Fichier de configuration Wireguard sur l'hôte H2:

```
root@H2:/etc/wireguard# cat wg0.conf
# Fichier de configuration WireGuard pour le tunnel wg0 sur H2
[Interface]
PrivateKey = QLlgVg376okXd1g76tp+/EvFDTZF7r2ThsRNYDocnGc=
ListenPort = 51820
Address = 10.10.10.2/24
[Peer]
PublicKey = vsfCRnDsgxcRzxELrULgFvGqbplBWqIRls/W6YHQ5kI=
AllowedIPs = 0.0.0.0/0
Endpoint = 1.0.0.3:51820
```

Une fois les configurations générées, cela permet de réaliser un tunnel wireguard peer to peer à l'aide du cryptokey routing. Une adresse ip est liée à une clé spécifique ce qui permet l'authentification de l'hôte. Wiregurad applique également le principe du silence est d'or, c'est-à-dire qu'il ne répond uniquement au personne authentifié donc aucun moyen de savoir qu'un service Wireguard est en cours.

Afin de sécuriser mes clefs, j'ai créé un fichier vault.yml contenant les clés. La marche à suivre est la suivante:

Créé le fichier vault avec la commande:

```
ansible-vault create vault.yml
```

En spécifiant un mot de passe pour sécuriser l'accès à ce dernier:

```
New Vault password:
Confirm New Vault password:
```

Une fois le fichier vault crée, on stocke nos différente clés dans ce dernier avec la syntaxe suivante:

```
h1pub: Z35T+X+RUfQwX7f6jlwM7WWLrFG8RVwBcw0Agbtt+kM=
h1priv: uHLswVcPZNYFXMvZd9fF1GFFYcIa9HcDFz8kU8+XpEQ=
h2pub: vsfCRnDsgxcRzxELrULgFvGqbplBWqIRls/W6YHQ5kI=
h2priv: QLlgVg376okXd1g76tp+/EvFDTZF7r2ThsRNYDocnGc=
```

Une fois cela réalisé, nous devons ensuite pouvoir récupérer ces différentes clefs pour la création des configuration Wireguard depuis le playbook. Pour cela nous devons specifier les paramètres suivant dans le playbook qui nous permettront d'utiliser les différentes clés dans le playbook :

```
vars_files:
  - vault.yml
```

Ensuite, dans les template pour la création des configuration Wireguard, on peut directement utiliser les clés comme des variable :

```
# Fichier de configuration WireGuard pour le tunnel wg0 sur H1
[Interface]
PrivateKey = {{ h1priv }}
ListenPort = {{ wireguard_listen_port_h1 }}
Address = {{ h1_address }}/24

[Peer]
PublicKey = {{ h2pub }}
AllowedIPs = 0.0.0.0/0
Endpoint = {{ h2_public_ip }}:{{ wireguard_listen_port_h2 }}
```

```
# Fichier de configuration WireGuard pour le tunnel wg0 sur H2
[Interface]
PrivateKey = {{ h2priv }}
ListenPort = {{ wireguard_listen_port_h2 }}
Address = {{ h2_address }}/24

[Peer]
PublicKey = {{ h1pub }}
AllowedIPs = 0.0.0.0/0
Endpoint = {{ h1_public_ip }}:{{ wireguard_listen_port_h1 }}
```

La dernière chose à réaliser est d'exécuter le playbook en spécifier le paramètre suivant qui permet de demander le mot de passe vault avant l'exécution du playbook:

(base) badr@MacBook-Air-de-Badr Ansible % ansible-playbook -i inventory/hosts.ini playbook/playbook-config-wireguard.yaml --ask-vault-pass
Vault password:

Malheureusement, je travail sur un mac M1 et j'ai l'erreur suivante :

ERROR! ansible-vault requires the cryptography library in order to function

Il semblerait que les mac M1 ne supportent pas la library aes256 mais sur d'autre environnement, cette marche à suivre devrait fonctionner.

Lien du git du projet contenant tous les playbook, les template et l'inventory:
https://gitedu.hesge.ch/badr.boucheri/labo4_network_automation_avec_ansible.git