# Fiche Ressource n°1 SAÉ32

---

# Module Python Paramiko

---

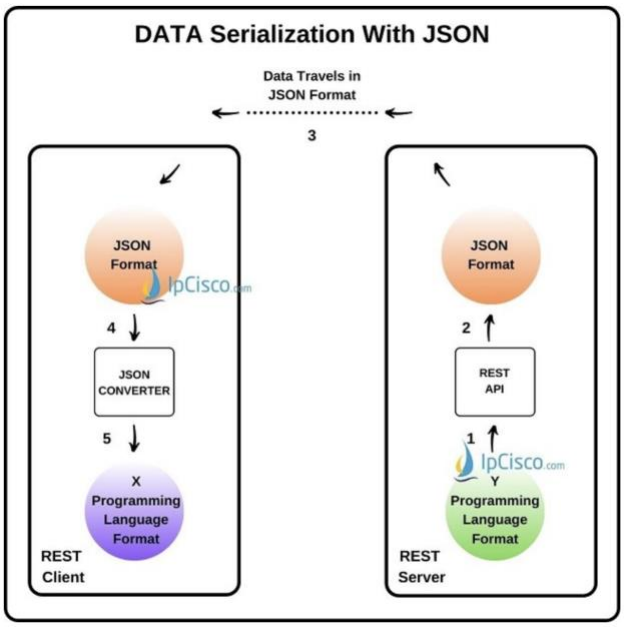
## 1. Introduction à l'automatisation des tâches d'administration des équipements réseaux avec Python

Vous avez jusqu'à présent configuré et dépanné les équipements réseaux (switch, routeur) en ligne de commande (CLI : Command Ligne Interface) ou via l'interface graphique (GUI Graphical User Interface).

Vous avez déjà dû réaliser que certaines tâches sont répétitives et que l'on peut facilement se tromper, notamment si on doit configurer un grand nombre d'équipements réseaux. De plus, on aimerait parfois pouvoir trier et regrouper le résultat de différentes commandes.

C'est tout l'objectif de l'automatisation des tâches d'administration ! Ce n'est plus l'utilisateur qui interagit avec la machine mais un programme créé par l'utilisateur, comme vous avez pu le voir pour l'administration de vos machines ou serveurs sous Linux avec vos premiers scripts bash.

Les switch et routeurs récents fournissent généralement des API « Application Prgramming Interface » qui permettent à un programme de communiquer avec lui et de récupérer les réponses dans une format standardisé et compréhensible par n'importe quel type de langage informatique (on parle de langage de sérialisation) XML, JSON, YALM, …



Cependant un grand nombre d'équipements réseaux ne disposent pas encore d'API et l'automatisation doit se faire en ligne de commande via une connexion SSH. Le résultat des commandes via la CLI est une chaîne de caractères et une des principales problématiques et donc de pouvoir récupérer l'information recherchée dans le résultat de la commande. Il faut parser cette chaîne de caractères ce qui n'est pas toujours très simple.

Python est un excellent candidat pour ce travail et il dispose d'énormément de librairies pour l'automatisation et le monitoring des réseaux (Paramiko, Netmiko, TextFSM, Ansible, …). On se contentera ici d'utiliser la librairie Paramiko qui permet d'émettre et recevoir des commandes via une connexion SSH mais vous pouvez dès à présent vos former sur Ansible si vous souhaitez améliorer vos compétences dans ce domaine :

https://www.youtube.com/playlist?list=PLn6POgpklwWoCpLKOSw3mXCqbRocnhrh-

## 2. Connexion SSH avec Paramiko

Le programme de connexion SSH avec Paramiko est illustré ci-dessous :

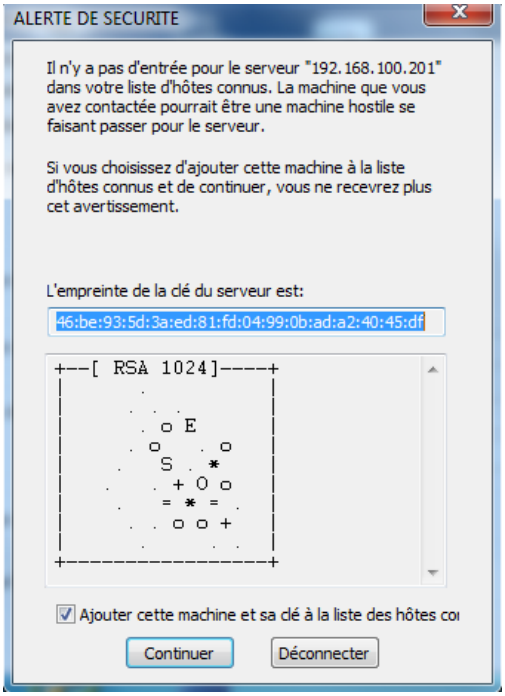
```
import paramiko


IP = "192.168.0.1"
ssh_port = 22
user = 'admin'
passw = '1234'


# Création de la connexion SSH
ssh_client  = paramiko.SSHClient()
ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh_client.connect(hostname = IP, port = ssh_port, username = user, password = passw)
print(f"Connecté en SSH à l'équipement d IP {IP}")
```

Le module paramiko permet de créer un client SSH pour se connecter à un serveur SSH via la classe SSHClient().

```
ssh_client = paramiko.SSHClient()
```

Une connexion SSH utilise un chiffrement asymétrique avec une paire de clés (privé/publique) et par défaut, l'instance de cette classe client rejettera les clés d'hôtes inconnus.

On doit donc créer une paire de clés et partager la clé publique avec le serveur ou plus simplement, ce que l'on fera ici, configurer une politique pour accepter les clés d'hôtes inconnus comme vous l'aviez fait manuellement sur Windows lors de vos connexions SSH aux routeurs Cisco :




La classe intégrée `AutoAddPolicy()` ajoutera les clés d'hôtes au fur et à mesure de leurs découvertes, il faut pour cela exécuter la méthode `set_missing_host_key_policy()` :

```
ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
```

La connexion est maintenant chiffrée et SSH a maintenant besoin des identifiants d'authentification pour savoir quel utilisateur spécifique souhaite se connecter. Cela peut être fait de plusieurs façons. Le moyen le plus simple consiste à utiliser la combinaison nom d'utilisateur et mot de passe ce que nous ferons ici avec la méthode `connect()` :

```
ssh_client.connect (nom d'hôte ou IP, port, nom d'utilisateur, mot de passe)
```

## 3. Canal Shell et canal exec

Une fois la connexion SSH établie, on peut envoyer des données chiffrées à travers cette connexion. On peut par exemple vouloir utiliser cette connexion pour avoir un shell entre 2 machines ou transférer des fichiers via SFTP. Les équipements doivent donc définir comment les paramètres de communication à travers cette connexion, on dit qu'ils doivent établir un canal « Channel » de communication. Il existe 2 types de canaux shell :

- **Session shell interactive** ou **canal shell** (commande invoke_shell avec Paramiko) : il est typiquement utilisé quand un **utilisateur** souhaite avoir un shell avec une machine à l'aide d'un client SSH comme Putty, Kitty , WinSCP, …La session est interactive et le résultat des commandes utilise donc par exemple la pagination (appuyer sur espace pour voir la suite du résultat de la commande), la

coloration et les confirmations interactives (Copy run start → Voulez-vous vraiment sauvegarder la configuration ?).

Le canal shell est une boîte noire avec une entrée et une sortie. L'entrée et la sortie n'ont pas de structure. Si vous exécutez par exemple une commande en l'envoyant à l'entrée, vous ne pourrez jamais dire quand elle s'est terminée. Si vous envoyez deux commandes à la file d'attente d'entrée, vous ne pourrez pas distinguer quelle sortie provient de quelle commande.

- Le **canal exec** (exec_command avec Paramiko) : il est typiquement utilisé quand un **programme** veut effectuer des commandes shell. Il ne faut donc pas une émulation de terminal, pour éviter la pagination, la coloration et les confirmations interactives. Ce canal permet de contrôler les flux d'entrée, de sortie et d'erreur standard d'une commande, comme si vous l'aviez exécutée localement.

Certains serveurs SSH comme OpenSSH permettent les 2 canaux mais d'autres ne prennent pas en charge l'un des canaux et/ou semblent le prendre en charge mais celui-ci ne fonctionne pas (généralement l'exec).

C'est le cas pour une grande partie des switchs qui ne prennent pas en charge le canal exec comme le DSLAM Zyxel (qui est globalement un switch avec des lignes ADSL à la place d'Ethernet). On devra donc utiliser le canal shell avec la commande « invoke_shell ».

```
channel = ssh_client.invoke_shell()
channel.send('show ver')
sleep(1)
response = channel.recv(8000)
channel.close()
```

La méthode `send` permet d'envoyer une commande dans le canal et la méthode `recv` de lire le buffer dans lequel est stockée la réponse (8000 correspond au nombre maximal d'octets qu'on lit dans le buffer de réception). Il est évidemment préférable d'avoir un temps d'attente `sleep(1)` pour être sûr que la commande a bien été envoyée et la réponse reçue avant de lire le buffer.

Remarque : on peut spécifier optionnellement les paramètres du terminal :

```
channel = ssh_client.invoke_shell(term='vt100', width=80, height=24, width_pixels=0, height_pixels=0)
```

On utilisera de préférence l'instruction `with` qui permet de d'ouvrir et fermer automatiquement le canal même en cas d'erreur :

```
with ssh_client.invoke_shell() as channel:
    channel.send('show ver')
    sleep(1)
```

```
response = channel.recv(8000)
```

## 4. Problématique des sorties paginées

Le canal shell est un canal interactif qui demande une interaction avec l'utilisateur et notamment si la réponse à une commande dépasse 1 page du terminal, il faut taper « espace » pour avoir la page suivante. Ce comportement n'est bien évidemment pas pratique quand c'est un programme et non un utilisateur qui se connecte à distance.

Pour résoudre ce problème certains équipements ont une commande pour désactiver la pagination et on commence donc par envoyer cette commande avant d'envoyer les commandes suivantes. Par exemple :

- Cisco: terminal length 0
- Huawei : screen-length 0 temporary
- Nokia/Alcatel : environment no more

Il ne semble pas y avoir de telle commande sur le DSLAM Zyxel que vous utiliserez (+1 à la SAÉ au premier binôme qui trouverait la commande) et on devra donc gérer la pagination à la « main » c'est-à-dire envoyer espace / entrée autant de fois qu'il y a de pages (ce qui dépend de la commande).

## 5. Script pour final pour la connexion SSH, l'envoi de commande et la réception de réponse

Un exemple de script est donné ci-dessous :

```
import paramiko

IP = "192.168.0.1"
ssh_port = 22
user = 'admin'
passw = '1234'
nb_pages = 6

# Création de la connexion SSH
ssh_client  = paramiko.SSHClient()
ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh_client.connect(hostname = IP, port = ssh_port, username = user, password = passw)
print(f"Connecté en SSH à l'équipement d IP {IP}")

# Création du canal est envoi des commandes
with ssh_client.invoke_shell() as channel:
    channel.send('show run')
    sleep(1)
```

```
output = ""
for i in range(nb_pages):
    try:
        page = ssh.recv(60000).decode("utf-8")
        output += page
        sleep(1)
        ssh.send(' '+'\n')
    except :
        pass
```