06/02/2023

Scripting Shell

Exécutable par un interpréteur et qui retourne dans tous les cas une valeur :

- Code de bon fonctionnement
- Code d'erreur
- Si possible un code différent en fonction des erreurs

Toutes les commandes Shell (systèmes) utilisées ont un code retour, même s'il n'est pas forcément visible.
Exemple : certains composants électroniques produisent différentes mélodies en fonction des erreurs.

Les erreurs sont mises :

- En return en affichage
- Dans le flux d'erreur
- Dans le flux standard

`tar` : permet des fichiers bout à bout pour sauvegarder.

Langage Shell

Langage de commande et d'échanges avec le système. Il existe différents langages :

- Bash
- csh, tcsh (proche du C)
- ksh (korn shell)
- …

Autoexécutions :

Première ligne « `#! /bin/bash` ».
Il est nécessaire pour le rendre exécutable « `chmod u+x mon_script.sh` ».
Finir le programme avec « `exit 0` » pour dire que le script s'est bien déroulé.

Commandes :

Une des problématiques est de différencier les chaines de caractères et les commandes. *Exemple* : `date`. Il faut donc noter les commandes avec le symbole « $ », cela permet de récupérer la valeur d'une commande (ls, date) ou d'une variable (chaîne de caractères, entier…). *Exemple* : `$(date)`.
Cependant, pour exécuter un script « `./var.sh` » pas besoin de mettre le « $ » si ce script est dans le même dossier que le script exécuté, qu'il a le chemin absolu ou qu'il est dans le `PATH`. Il faut que le fichier soit accessible.

- « `echo` » permet d'afficher une variable.
- « `ls` » donnera la liste des fichiers/dossiers de là où le script est lancé.
- Les guillemets ne sont pas obligatoires.
- Tout est chaine de caractères.

Exemple :

```
var2=45
var2=$var2+359
echo $var2
```

Affichera « 45+359 »

Il faudrait noter : `var2=$(($var2+359))`

- Tableaux :
  - Déclaration :

```
tab = ('toto' 'titi')
tab[0] = 'bob'
tab[1] = 'alice'
```

  - Afficher toutes les données du tableau

```
echo ${tab[@]}
```

  - Avoir la taille du tableau

```
echo ${#tab1[*]}
```

- Attention à la syntaxe : les espaces sont importants.
- `read` : lecture du clavier, quand rien n'est précisé, jusqu'au retour à la ligne (\n). Il lit jusqu'à l'IFS.
- `cat` : lecture d'un fichier. *Exemple* : `$(cat toto.txt)`
- `\n` : retour à la ligne, par défaut dans les « `echo` ».

Commande	*Explications*	*Exemple*
`$?`	Valeur de sortie de la dernière commande.	Permet de récupérer si le programme s'est bien déroulé.
`$0`	Nom du script	
`$n`	Nième argument passé au script	
`$#`	Nombre d'arguments. Permet avant de lancer le script de savoir combien d'argument le script a besoin.	
`$*`	Permet de récupérer les arguments un à un.	`$1` : récupère le premier argument.
`$$`	PID du processus courant.	

Cartographie logicielle : pour savoir tout ce qu'on a à disposition au sein de son entreprise ou de sa collectivité.

Cartographie des droits : pour savoir qui a les droits sur quoi.

Il est important de savoir sur quelles données, quels serveurs le travail est fait.

Le script Shell :

- Variables non typées
- Variables non déclarées

} Pour résoudre ce « problème » certaines entreprises mettent dans le nom de la variable le type. *Exemple* : `int_hauteur`.

- Les variables peuvent être en lecture seule : `readonly`.
- Exploitation de variables d'environnement. « `env` » permet de savoir où sont situées les variables d'environnement.

Tests : (FINIR REVOIR SLIDE)

`-e`	Permet de savoir si le fichier existe.
`-f`	Permet de savoir si c'est un fichier.
`-d`	Permet de savoir si c'est un dossier.
`-s`	Non vide
`-r | -w | -x`	Propriété du fichier.

f1 -nt f2	Plus récent que
f1 -ot f2	Plus vieux que
Expr1 -a Expr2	Expression logique « et »
Expr1 -o Expr2	Expression logique « ou »
! Expr	Négation
-z	Chaîne vide
-n	Chaîne non vide
-lt	Strictement inférieur
-gt	Strictement supérieur
-le	Inférieur ou égal
-ge	Supérieur ou égal

Comparaisons :

Chaines de caractères :	Numériques :
!= ou = (pas ==)	-eq → égal -ne → différent

Structures conditionnelle et itératives :

Les bloques sont définis explicitement par des termes (pas d'indentation, pas de « {} »).

- if

```
if [ $# -ne 1 ]
then
      echo usage : $0 argument
      exit 1
fi
echo usage2
exit 0
```

Indique qu'il manque un argument.

- case

```
case $# in
0) echo aucun paramètre ;;
1) echo 1 paramètre $1 ;;
2) echo 2 paramètres $1 - $2 ;;
3) echo 3 paramètres $1 - $2 - $3 ;;
*) echo salut
esac
exit 0
```

C'est le **default**

- for

```
for variable in $ma_liste
do
      commands
done
```

Exemple :

```
for var in $*
do
      echo $var
done
```

$var va prendre la valeur des éléments de la liste successivement.

Exemple 2 :
Puisque ça n'est pas son répertoire il faut reconstruire le chemin.

```
if [ $# -ne 1]
then
      echo usage $0 repertoire
      exit 1
fi
for f in $(ls $1)
do
      if [ -f $1/$f]
      then
      echo $f est un fichier
elif [ -d $1/$f]
then
      echo $f est un répertoire
fi
done
```

A FINIR

- `while`

Tant que la condition est respectée.

```
while [ $cpt -le 10 ]
do
      echo $var
      let cpt++
done
```

- `until`

Jusqu'à ce qu'il y a derrière est respecté je continue.

```
until [ $cpt -le 10 ]
do
      echo $var
      let cpt++
done
```

Gestion des flux

- < : entrée standard, lecture d'un fichier
- > ou 1> : flux standard.
- 2> : flux d'erreur, sur l'écran.
- 2>&1 : redirection de la sortie d'erreur vers la sortie standard.
- > : écriture et création du fichier
- >> : ajout à un fichier et création si le fichier n'existait pas encore.

08/02/2023

Fonctions

- Création de fonctions sans paramètres :

```
nom_fonction(){
      echo exemple
```

```
        ls -l
}
```

L'appel de la fonction se fait sans parenthèses :

```
nom_fonction
```

- Si la fonction a des paramètres ils ne sont pas mis le prototypage. Les paramètres sont exploités comme dans des scripts.

```
nom_fonction(){
        echo $1
}
```

L'appel de la fonction se fait ainsi :

```
nom_fonction $path
```

Travail sur les chaînes

Extraction dans une chaîne

`${ chaine_où_on_travaille : position_à_partir_de_où_on_veut_extraire : nombre_de_caractères}`

Exemple : si `chaine = salut à vous tous` alors `${chaine :8 :4}` ne renverra que « `vous` ».

Troncature

Donner la chaîne et dire à partir de quel caractère on coupe. (FINIR EXEMPLE SLIDE)

IFS, Internal Field Separator

Le délimiteur est de base « `\n` » ou « `\r\n` ». Les caractères « `\r` » permet d'avancer historiquement sur les machines à écrire. Il est possible de la changer. Redéfinir l'IFS permet de définir un « split » de manière implicite. Cette méthode est pratique avec le traitement de fichiers CSV.

Il faut sauvegarder son ancien IFS pour le remettre en place à la fin du script :

```
oldIFS=$IFS
toto= 'oli;vier:toto'
IFS= ';'
echo $toto
for var in $toto → lecture ligne par ligne
do
        echo $var   → Fonctionne également en lisant des fichiers.
done
IFS=$oldIFS
```

Exemple slide 34 : il ne faut pas oublier de remettre la valeur standard à l'IFS dans la fonction `affich()` sinon la lecture du fichier ne se fera plus ligne par ligne.

Scripts à travers le réseau

Il y a un problème récurrent : **où est ce qu'on met le login et le mot de passe ?**

- SSH : échange de clefs et donc pas de login et de mot de passe. Seule solution recommandée.
- Phrase de passe :
    - Taper à chaque fois ses identifiants : n'est pas efficace car demande beaucoup d'interactivité.
    - Mettre les identifiants dans le script : pas recommandé
    - Mettre les identifiants dans un fichier : pas recommandé

WGET

Outil lié aux requêtes de téléchargement, d'aspiration d'un site. Exploite les protocoles http, https.

- Récupérer la taille du document, sans récupérer le document en lui-même. Utilise le verbe HEAD pour trouver la taille. La commande est : `--spider uri`
- WGET permet de reprendre le téléchargement d'un document : `wget --continue`
- WGET permet de limiter le téléchargement : `wget --limitrate`
- WGET permet de changer les liens dans une archicture. Par exemple redéfinir des chemins absolus en chemins relatifs : `wget --mirror --convert-links`

Quand on finit d'exécuter `wget` on sait ce qu'il s'est passé :

- 0 : succès
- 1 : erreur générique, il ne sait pas ce qu'il s'est passé.
- 2 : parse error, n'arrive pas à comprendre la ligne.
- 3 : erreur d'entrée ou de sorti de fichier. *Exemple* : pas le droit d'écrire dans le fichier.
- 4 : erreur de réseau.
- 5 : erreur d'authentification SSL, avec HTTPS.
- 6 : erreur d'authentification (login ou mot de passe).
- 7 : erreur dans http ou FTP.

- 8 : erreur retournée par le serveur. Correspond aux erreurs 400.

Curl

Bibliothèque de requête pour les URL. Ce type de librairie est disponible dans tous les langages.
Accès et création de ressource sur un serveur. Il est utile pour tester un environnement REST.
Il supporte les protocoles suivants, ainsi que leur version sécurisée :

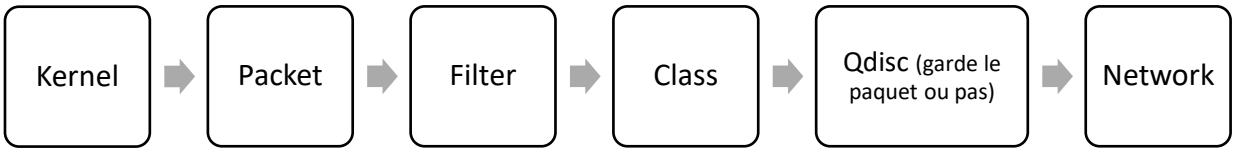
- FTP(S)
- HTTP(S)
- SMPT(S)
- POP(S)
- IMAP(S)
- SCP
- SFTP
- SMB

Tout peut être passé en paramètre.

- Il ressort sur la sortie standard. Il faut donc lui dire où écrire : `-o fichier.txt http://...`
- Refuser les redirections : `-L http://`
- Mettre des conditions sur la date : `-z`
- Permet de récupérer l'en-tête (HEAD) : `curl -I`
- Modification de la méthode : `curl -X POST http://...`
- Construction de l'URL GET avec les données (ici paramètre texte) fournies : `curl --data texte http://...`
- Rajouter un cookie : `--cookie`
- Repérage avec l'URL : `ftpuser:ftppass -O`
- Utile pour télécharger, pour faire de l'upload : `ftpuser:ftppass -T`

TC Traffic Control

Commande standard du paquet iproute2. Elle fonctionne sur une chaîne : Qdisc, Queue Discipline. C'est un gestionnaire de paquet, Qdisc, où chaque interface à un gestionnaire de paquet qui lui est associé. Ce gestionnaire est par défaut transparent. Mais c'est sur lui qu'on dépose des stratégies pour détruire, retarder, les paquets. Il est en effet possible de mettre en place un système de buffer et un système de filtrage sur le trafique entrant et sortant.



Permet :

- La modification du comportement d'une interface réseau (virtuelle ou physique).
    - Limiter, lors de l'utilisation de VM, **la bande passante** (fonction la plus utilisée).
    - Permet de favoriser certaines interfaces vis-à-vis des certaines interfaces.
    - La commande `ping` ne permet pas de tester ce type de configuration. Utilisation de `iperf`.
        - `tbf` : exploitation d'un filtre et non de l'émulateur netem
        - `rate` : taux de transfert
        - `burst` : taille maximale en rafale sans perte de paquets (taille de la file de transfert, taille jusqu'à quel moment on perd des paquets).
        - `latency` : latence maximale
- Les tests:
    - Créer de la faute
        - Provoquer une perte de paquets

          ```
          sudo tc qdisc add dev ens160 root netem loss 10%
          ```

        - Provoquer une corruption de paquets (met des bits à 1 ou à 0)

          ```
          sudo tc qdisc add dev ens160 root netem corrupt 10%
          ```

        - Provoquer la duplication de paquets

          ```
          sudo tc qdisc add dev ens160 root netem duplicate 10%
          ```

- L'ajout un délai constant à une interface. Utilisation d'un émulateur pour redéfinir les propriétés du réseau. Pour tester la commande `ping` est utile. Il n'y a qu'un Qdisc par interface. *Exemple* :

  ```
  sudo tc qdisc add dev ens160 root netem delay 100ms
  ```

  Tout ce qui va sortir par l'interface aura un délai de 100ms.
  Pour récupérer l'état initial sans relancer (reboot) l'interface il est possible de supprimer le Qdisc rajouté avec la commande :

  ```
  sudo tc qdisc del dev ens160 root
  ```

  Il est possible de définir des délais avec des dérives (en suivant une loi normale, Pareto...). Par défaut
  Exemple : Distribution uniforme (ici entre 90 et 110)

  ```
  sudo tc qdisc add dev ens160 root netem delay 100ms 10ms
  ```

- Il est possible de créer des règles plus fines :
    - Sur les utilisateurs (SMB, user…)
    - Sur la source

- Sur la destination. *Exemple* : permet d'adapter la qualité des vidéos renvoyées.

Il est possible de combiner les règles, les limites.

Attention aux tests en SSH. *Exemple* : avec la mise en place d'un `corrupt`, les échanges vont être corrompus. Même chose pour les délais.

13/03/2023

Scripting Python

Python 2 ou Python 3

Python 3 n'est pas la suite de Python 2. Une différence majeure entre les 2 versions de Python est le codage des caractères. Les caractères en Python 3 sont codés en UTF-8.
Python est un langage interprété. Mais l'interpréteur est spécifique et propre à chaque système.

Avantages

- Moins rigide : permet de simplifier l'écriture des scripts.
- Extensible avec des modules. En effet, il existe énormément de fonctionnalités qui sont déjà codées, qui existent déjà. Attention cependant au choix des modules, certains ne fonctionnent pas.
- Plus rapide.
- Portable.
- Extension vers les applications avec l'existence de nombreuses API.

Modules

Python et Pearl sont des langages fait pour du Scripting System. Python offre des modules natifs et externes.

Pour les installer utiliser `pip3` ou télécharger directement les paquets sous Linux.

- Interaction avec l'interpréteur : module `sys`.
- Accès au système d'exploitation : module `os`.
- Accès à l'exécution : module `subprocess`.
- Gestionnaire de log : module `syslog`. Permet de provoquer des sources de log dans les scripts.
- Connexion SSH : module `Paramiko`. Implémentation python de SSHv2. Cela permet d'exécuter des commandes à distance de manière scriptée.

```
ssh = paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh.connect('monsvc.fr', username='oflauzac', password='iciETla')
stdin,stdout,stderr = ssh.exec_command("ls /etc/"
```

Dès cette commande : connexion à distance sur l'objet SSH établie.

- Modules autour de la compression :
  - `Zlib`
  - `Gzip`
  - `Bz2`
  - `Tar`
  - `Zip` et `zipfile` : permet de gérer les fichiers zip.
    Exemple :

```
zipf = zipfile.ZipFile("monzip.zip","w")
zipf.write("args.py")
```

On rajoute les fichiers qu'on souhaite zipper à la suite.

```
zipf.write("exec.py")
zipf.close()
```

Scripts

Arguments

Dans un premier temps il faut vérifier les arguments passés au script. `sys.argv` permet de récupérer les arguments. `argv[0]` est le nom du script.

Flux standards

- `stdin` par exemple avec `read.`
  Exemples :
  - `lg = sys.stdin.readlines()` → la lecture s'arrête seulement au caractère de fin de lecture : `Ctrl+D`.
- `stdout`
- `stderr` : par *exemple* ce dernier permet de stocker directement dans des logs.

Fichiers

- Ouvrir un fichier :
  - `fd.write(nom, flag, mode=0777)` → écrase le contenu
  - `open(nom, 'r')`
- Lecture :
  - `read(taille)` : lit tout dans une grande chaîne de caractères
  - `readlines()`
  - `readline()`
- Fermeture : `fd.close()`

Le système de fichier est l'ensemble des fonctions qui permettent de manipuler les fichiers et les répertoires. Il existe des nombreuses bibliothèques disponibles (`shutil, filecomp`…) en plus des bibliothèques natives.

Exemples :

- Permet de changer de répertoire : `os.chdir("..")`
- Permet de créer un dossier : `os.mkdir('NewRep')`
- Permet de vérifier que ce soit un fichier : `os.path.isfile("./NewRep")`
- Permet de copier des fichiers : `shutil.copy("./mdp.txt","toto.txt")`
- Permet de comparer les fichiers : `filecmp.cmp("./mdp.txt","./toto.txt")`

Exécutions

Possibilité d'appeler des scripts et des commandes externes.

`Subprocess`	Classe `Popen` (Devrait disparaitre)	`Run`
Permet d'exécuter en sous-processus. Fonctionnalité disponible depuis Python 2.4. C'est une fonction de haut niveau, elle est indépendante du système. Cela permet d'unifier les accès entre les différents systèmes d'exploitation. Exploitation de la classe `Popen`.	- `wait()` : attente de la terminaison du processus enfant. Action bloquante. - `poll()` : vérification de l'activité du processus. Il est possible d'en lancer en parallèle. Paramètres de création : - Taille des buffers	Simplification des commandes. `Run` est un outil d'intégration. Il est possible de lancer des commandes sous la forme d'une chaîne ou sous la forme d'une liste. `Run` gère la gestion des erreurs et des flux. *Exemple :*

	- Etat des flux
	- Définition de la commande :
	  ○ `String`
	  ○ `Liste`
	
	Exemple : `p = subprocess.Popen(["ls", "-al"], stdout=subprocess.PIPE)`
		`res = subprocess.run("ls -l", shell=True, check=True)`
		
		Affiche ce qu'il mettrait à l'écran.
		Permet de mettre la commande sous contrôle d'exécution.

Fin de script

Exploitation de `exit(0)`. Le code 0 signifie qu'il n'y a pas eu de problèmes.

Expressions régulières

Module standard : `re`.

- Recherche de motifs : `re.search(pattern, text).` Extensible avec `findall.`
- Séparation, pour les fichiers CSV par exemple, `text.split(';')`
- Remplacement : `re.sub(";","---",text)`

Scripting multi échelle

Montrer qu'il existe différentes solutions possibles pour une même problématique.

Notion de plateforme : script qui agit sur :

- Un serveur unique
- Une ferme de serveur (cluster)
  Script cluster : permet d'exécuter des scripts sur d'autres machines.

Permet de travailler en API Rest. Il suffit alors d'avoir la liste des end points.

Plateformes

- `LibVirt` : qui a un shell spécifique, `virsh`.
- `Proxmox` : surcouche de gestion pour LXC et QEMU. Permet d'avoir une couche graphique ainsi que des commandes au-dessus. Scripting en local (pct) ou scripting au niveau du cluster Proxmox : `pvesh`. Notion de script un peu plus haute puisqu'on tape sur des commandes intermédiaires. Une autre solution est d'utiliser l'API REST.
- Docker: scripting shell, ou avec Docker Compose.

14/03/2023

LXC et LXD

Les limites de LXC

LXC est basé sur les cgroups et les namespaces. Création de conteneurs à partir de template image. LXC est conçu pour gérer une seule machine. Ce n'est pas fait pour gérer des clusters puisque LXC n'a pas de démon, contrairement à Docker. Gérer un cluster LXC → gérer indépendamment chaque machine. Il n'est pas nécessaire d'être root pour gérer des conteneurs. Gestion du réseau avec :

- `lxc-net`.
- Mise en place de bridge système
- Mise en place de bridge OVS

Il est possible de scripter en LXC à partir de commandes systèmes. La problématique est sur la localisation des scripts.

Exemple : `lxc-create -n ctn01 -t centos`

Ces commandes sont des appels systèmes locaux.

La configuration réseau est compliquée à migrer. Il est facile de migrer les données mais il faut conserver les mêmes propriétés réseaux. Cela est donc à gérer à la main. Cela montre les limites de LXC.

LXD une vision shell cluster

LXD se place au-dessus de LXC en rajoutant un démon qui va permettre de gérer une ferme de clusters LXC avec une vision shell cluster.

LXD a sur chaque serveur un service spécifique. Il permet de :

- Gérer le réseau :
    - Création d'un réseau uniforme
    - Capacité d'isolation du réseau : permet de découper le réseau.
- Gérer le stockage

Possibilité de spécialiser les machines :

- **Machine disque** : les conteneurs sont mis à un endroit et peuvent être exécutés à un autre endroit. Cette machine a besoin d'un disque et d'un réseau rapide puisque tout le monde va écrire dessus en même temps.
- **Machine pour l'archivage** :
    - Les snapshots peuvent être mis à des endroits précis.
    - Les templates sont mis sur une machine spéciale

Hyperconvergé : tout remettre dans une seule même machine.

Installation

- Installation du paquet LXD (meta package)
- Sur chacune des machines lancer `lxd init`
    - LXD demande où on stocke les templates et les images
    - Permet de créer un cluster ou d'ajouter une machine à un cluster déjà créé.
    - …
- Une fois l'installation terminée, nous sommes sur le shell cluster : commande `lxc`.

Scripting avec les commandes lxd

Commandes (intégrables dans un script shell) :

- `lxc launch` : création et lancement d'un conteneur
- `lxc init` : création de conteneur
- `lxc start/stop`
- `lxc list` : permet de lister les conteneurs du cluster.
- `lxc exec` : exécution d'une commande dans un conteneur
- `lxc file` : gestion des fichiers des conteneurs

- `lxc network` : gestion du réseau. Le réseau est posé sur l'ensemble du cluster.
- `lxc image` : gestion des images de conteneurs
- `lxc config` : récupération / modification / édition de la configuration
- `lxc config show` : permet de récupérer toute la configuration de la machine.

Scripting avec Python

Il est possible de rajouter le module `pylxd` pour travailler en python. C'est une proposition d'une API surcouche de l'API REST. Permet d'exploiter les données au format JSON.

Cf. exemple dans le cours.

Scripting avec l'API REST

Les communications sont faites en HTTPS. L'administrateur n'a plus besoin d'avoir un compte sur le cluster ou de se connecter en SSH. De plus l'administrateur est libre du langage de programmation d'administration des machines.
Il est également possible, facilement, de mettre en place un dashboard, une interface graphique web pour gérer les machines.

Il faut un serveur web pour que l'API REST fonctionne. Le désavantage est l'augmentation des failles de sécurité avec le rajout du serveur web.

Attention, par défaut, le serveur écoute seulement sur 127.0.0.1. Il faut penser à changer ce paramètre pour qu'il écoute sur toutes les interfaces.

L'authentification se fait par :

- Mot de passe
- Certificat

L'API REST est basée sur 4 éléments (CRUD) :

- `Create`
- `Read`
- `Update`
- `Delete`

Pour rappel les librairies sont : `requests, json, urllib3` (cette dernière permet de gérer le HTTPS).

Penser à bien encoder en UTF-8. Cf. exemple dans le cours.

- Pour lancer un conteneur il faut générer une mise à jour du conteneur, avec `PUT`.
- Pour exécuter une commande sur le conteneur utilisation de `GET`. Souvent, on fournit une liste qui contient :
    - Le nom de la commande
    - Le premier paramètre
    - Le deuxième
    - etc.

  La `width` et le `height` est la « hauteur du terminal », la taille du buffer.

Provisionning

Le provisionning fait partie du domaine de l'automatisation. Historiquement, le provisionning a été défini comme l'approvisionnement de machines. Le provisionning devient un outil pour apporter du support et aider la gestion de conteneurs.

Solutions :

- Scripting manuel
    - Bash
    - Python / Pearl
- Solution client/serveur
    - Serveur de configuration
    - Agent de gestion sur les machines à administrer

  Exemples de solutions *:*
    - Ansible : passe par Python et SSH, sans agent. La machine qui donne les ordres est sur le même réseau que celles qui exécutent les ordres)
    - Soulstack : l'agent, qui est en publish/subscribe, regarde régulièrement les modifications à faire. Donc l'administrateur et les machines administrées peuvent être dans des réseaux différents.

  Ces solutions restent dans l'idée du script. Il faut « décrire » ce qu'il faut faire. Différent de Terraform où il est seulement nécessaire de dire ce qu'on souhaite à la fin. Plus besoin de déclarer toutes les actions que l'on souhaite réaliser.

Limite du scripting :

- Travail fastidieux
- Problème d'hétérogénéité
  Exemple : pour éviter ce problème une entreprise a seulement 3 types de machines qu'elle change tous les 5 ans.
- Pas d'unité de définition des opérations

16/03/2023

Solution client / serveur :

- Solution directe : un serveur + plusieurs clients
- Solution inversée : un client (station d'administration) initie la requête et attend la réponse + agents (les serveurs). *Exemple* : SNMP.

PUPPET et CHEF des anciennes solutions

PUPPET

Solution inversée.

Logiciel de configuration de serveurs esclaves. Il est programmé en RUBY. Il existe une version libre et commerciale (pour avoir l'extension VMWare ou Windows…).

Il y a :

- PUPPET master : une machine qui avait où tournait un démon PUPPET master
- Des clients

CHEF

Il est programmé en RUBY. CHEF a été utilisé pour provisionner des machines virtuelles. Il a été utilisé en mode mono poste. Présence d'un agent sur le client.

ANSIBLE

Solution pour travailler sur les machines et non sur les infrastructures. Il n'y a pas de vision globale. Tout est en ligne de commande. Il n'a pas d'agent.

Déploiement automatique :

- Installation de packages
- Installation de fichiers de configurations
- Configuration des utilisateurs systèmes
- Configuration des services

⇨ Tout ce qu'on peut faire en SSH

Architecture :

- Une station d'administration (sans forcément être super utilisateur)
- Machines à administrer : la machine administrée ne sait pas qu'elle est administrée par ANSIBLE.

Fonctionnement :

Depuis la machine d'administration ; connexion à la machine à administrer. Puis, exécution des commandes de configuration en ligne de commande (`ad hoc command`) ou Playbook YAML.

Le but est **d'exploiter les modules ANSIBLE.** Par *exemple* exploiter le module qui gère les paquets pour que ce paquet installe les paquets. Il existe de nombreux modules.

Exemples en lignes commandes :

```
ansible ans2 –a 'touch toto.txt' –I hosts.ini
ansible ans2 –m ping –i hosts.ini
```

Exemple dans un Playbook :

```
---
– hosts : ans2
  remote_user : nom_user
  tasks :
   – shell : echo Hello World >> test.txt

ansible–playbook –i host.ini play1.yaml
```

Il existe une solution « coffre-fort », ansible volt, avec un fichier hacher accessible avec un mot de passe qui détient les autres mots de passes hachés.

Installation :

- Sur la machine d'administration :
  - Installation python
  - Génération de la clef SSH
  - Copie de la clef sur les machines administrées
- Sur les machines administrées
  - Création d'un compte utilisateur
  - Python
  - Mise en place de sudo

Les cibles sont contenues dans le répertoire : **`/etc/ansible/hosts`**, dans un fichier hosts local.

fichier hosts (ini) :

```
ans1 ansible_connection=local
ans2 ansible_ssh_host=192.168.65.15

[local]
ans1
[distant]
ans2
```

Groupes

La commande **`--beacon`** permet d'exécuter des commandes en root si on ne met pas de paramètre derrière (nom), sinon exécute les commandes avec l'identité de l'utilisateur rentré en paramètre.

ANSIBLE se pose sur le système, la virtualisation et le réseau. Il est fait pour piloter des machines (pas plateformes → sinon voir Terraform)

22/03/2023

SALTSTACK

Architecture client/serveur avec l'exploitation d'une file ZeroMQ. Permet de manager des machines qui sont sur des architectures différentes.
Saltstack a des ports d'écoute qui sont les ports : 4505 et 4506.
Il y a un agent installé sur le client, appelé minion, salt-minion. La machine qui doit diriger l'ensemble est le master, appelé master.

Le serveur doit être configuré au niveau :

- Du réseau (interfaces, ports d'écoute…)
- Des ressources (fichiers ouverts, nombre de jobs…)
- De la sécurité (utilisateurs, certificats…)
- Des modules
- De l'architecture des fichiers
- Des logs

Configuration client/minion

Configuration dans le fichier : **`/etc/salt/minion`** du master. *Exemple :* présence de la ligne « **`master : 192.168.59.10`** » dans ce dernier fichier.
En effet, il faut que les minions s'enregistrent auprès du master qui doit les accepter.

`salt-key -L` : permet d'avoir sur les minions la liste des clefs acceptées, rejetées.
Le serveur master les accepte avec la ligne de commande : **`salt-key --accept=<id>`** ou **`salt-key --accept-all`**.

Exécution

Il est possible d'exécuter des commandes (de tests, d'exécution…) pour une machine particulière ou plusieurs.

Exemple : **`salt '*' sys.doc`**

→ Sur quelle machine exécuter la commande (* = sur toutes les machines enregistrées)
→ La commande à exécuter

Il est possible de programmer l'heure de l'exécution de commande avec :
`salt nom_machine at.at 10:25am commande`

Info bonus :
Le paquet **`Figlet toilet`** permet de créer des bannières dans le terminal.

`state.sls` : fichier qui regroupe des commandes.
Une tache est un enchainement d'opération cohérent.

Exemple :
Dans le fichier **`/srv/salt/net.sls`**

```
install_net_packages :
      pkg.installed :
            -pkgs :
                  -rsync
                  -curl
```

Il y a du LXC qui est installé en natif.

Conclusion

ANSIBLE et SALTSTACK sont deux outils différents. Ils sont différents dans leur structure et leur usage. Le choix dépend de l'architecture, de la configuration ainsi que de l'installation de modules complémentaires.

Scripting autour du WEB

WEB scraping

Le WEB scraping est l'extraction automatique d'éléments WEB à partir d'un lien WEB pour récupérer des données textuelles, multimédias ou des métadonnées. Il est possible d'exploiter, entre autres, des balises HTML ou d'utiliser des API.

Le WEB scraping permet :

- De contrôler l'état de son site (contrôle des liens)
- De faire de la collecte et de l'agrégation de données :
    - Référencement
    - Veille : *exemple :* permet de comparer des pages web et de ne pas récupérer les choses redondantes.
    - Collecte
    - Réorganisation de données

Attention aux formulaires et aux cookies → Si on souhaite continuer sur le même site il faut conserver les cookies.
Analyse du document manuellement ou avec DOM.

Il existe des API pour le scraping :

- BeautifulSoup : documentation ici.

BeautifulSoup

The Beautiful Soup is a python library which is named after a Lewis Carroll poem of the same name in "Alice's Adventures in the Wonderland".

Il permet l'exploitation de documents parsés :

- Parsing interne
- Interface avec des parsers évolués pour être plus efficace :
    - `lxml` : module python qui permet de gérer du XML.
    - `html5lib`

Mise à disposition de fonctionnalité pour exploiter les documents. Le problème est parfois de comprendre la structure des documents à analyser pour extraire ce que l'on souhaite.

1- Récupération de la page
    a. API request
    b. Lecture du fichier
2- Parsing de la chaîne obtenue
    Parser signifie structure les données
3- Accès aux éléments

Pour le détail des fonctions disponible cf. le lien plus haut.

TEST WEB

Tester si son site fonctionne correctement : le test fonctionnel.
Il permet, entre autres, de :

- Tester des formulaires (*exemple :* formulaire incomplet)
- Tester l'activation des liens
- De récupérer le temps nécessaire à l'achat d'un produit
- De tester la monté en charge

Le test fonctionnel peut être automatisé.

SELENIUM

SELENIUM est une suite d'outils. Il ne permet pas de tester le temps de latence, il ne fait pas de test protocolaire. En effet, on n'a pas accès aux requêtes http, aux protocoles.

L'outil présenté ici est SELENIUM GRID.

SELENIUM IDE – Exécution graphique

SELENIUM IDE est un plugin navigateur qui exécute des enregistrements. Il faut télécharger cette extension afin de capturer les éléments cliqués. Cette séquence peut être rejouée, ce qui permet sur une petite séquence de tester rapidement.

SELENIUM WEBDRIVER

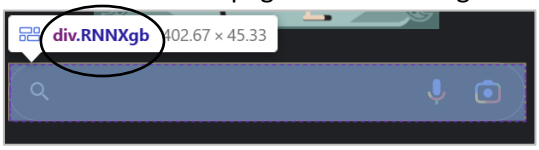
Exécution graphique

Il faut télécharger le bon driver de navigateur en fonction de son propre navigateur. Cela permet de tester son site sur plusieurs navigateurs. Il existe des API dans n'importe quel langage. SELENIUM WEBDRIVER permet d'injecter des actions dans un navigateur et permet de récupérer les pages pour les parser.

1. Création d'un service
2. Liaison avec le navigateur / driver
3. Accès à la page de base
4. Exécution des opérations

   Il est possible d'agir sur les éléments. *Exemple :* `click()`.

Le problème reste la détection des éléments.
Exemple : Nom généré « aléatoirement » sur la page d'accueil Google :



Exécution non graphique

Le même principe mais option disponible pour ne pas ouvrir de fenêtres graphiques.

Limites de SELENIUM

Pas applicable aux applications de bureau. Il n'y a pas de *reporting* ni d'outils d'intégration.

Test de performance

Tester la performance est tester la capacité de réponse en fonction d'une charge.

Exemple : est-ce que ma file MQTT arrive à gérer 1000 accès par seconde ?

Ce n'est pas seulement le test sur le nombre de requêtes mais également sur le chargement / l'upload de fichiers. Le but est donc de trouver ce qui « lâche » en premier et où (réseau, Apache, …).

Quelques métriques classiques :

- Réémission
- Débit
- Mémoire
- Temps de réponse
- Perte

  …

Il est nécessaire de mettre en place des scénarios. Il existe de nombreux outils Apache JMeter, Gatling, Locust.

Locust

Locust est un fichier Python. Création d'une classe de test. Il est possible de tester de l'API REST. Voir la documentation Python ici ou la documentation Locust ici.

Tutoriel Locust pour commencer : Your first test — Locust 2.15.1 documentation