

# Automatisation du cloud (AWS) avec Terraform

- Rappel sur les systèmes d'exploitations
- Architecture du "cloud"
- Ressources AWS
- Automatisation du déploiement dans AWS
  - ligne de commande, boto (Python)
- Terraform
- Deploiement de conf.
- Integration continue

Les différents TP seront à livrer  
dans un dépôt git public  
github, gitlab, framagit

le mien :

<https://framagit.org/jpython/aws-terraform-1>

Si vous n'avez pas de compte sur  
l'une de ces plate-formes, pensez à  
créer un aujourd'hui.

# Cloud et architecture des OS

C'est quoi un OS ?

- \* Un logiciel qui présente une machine plus simple, plus générique, plus abstraite qu'une machine réelle
- \* L'OS prend en charge l'accès aux ressources physiques et fournit une API pour les applis
- \* RAM : mémoire virtuelle (MMU)
- \* CPU : temps de calcul est réparti entre les tâches
- \* Fichiers (par ex.) au lieu de blocs sur un disque

Première approche : Mainframe (IBM)

- L'OS présente une copie de l'environnement physique: machine virtuelle (VM/370 - 196x - zOS)

Autre approche : présenter une machine abstraite: UNIX (1970), GNU/Linux (91), Digital VMS (197x), MS Windows (NT et ultérieur)

Quel est le nom du "noyau", en général sous UNIX, ou sous Linux ?

ex. sous Linux: /boot/vmlinuz-...

sous certains UNIX: /vmunix

Et les conteneurs dans tout ça ?

La machine "abstraite" que présente UNIX aux applications (dev et aux admins) est basé sur des espaces de noms :

- système de fichier (racine / et des répertoire)
- ensemble de processus (/proc)
- ensemble de users et groupes
- réseau (hosts, networks, interf.)

Historiquement les mêmes espaces pour tout le monde (+ droits d'accès)

# Conteneurs

- chroot (appel système et une commande shell) permet de "mettre un prison" un processus (et ses descendants) : présenter comme racine une sous-arborescence
  - pour l'admin (install)
  - pour isoler des processus pour la sécurité

Généralisation de cette approche,  
segmentation des espaces de noms

Linux : cgroups -> LXC, Docker, etc.

Solaris : slices

## Cloud ?

- La réservation et l'utilisation de ressources de calcul mutualisés à distance

- IaaS : Infrastructure as a service
  - des machines virtuelles
- PaaS : Platform as a Services
  - un serveur Web et PHP
  - une base de données
  - ...
- SaaS : Software as A Service
  - gmail
  - salesforce, ...

## AWS : IaaS

(même concepts GCE, Azure, Open Stack)

- Instance de VMs
- Volumes de stockages
- VPC (Virtual Private Cloud) : un espace réseau où nos VMs communiquent
- Réseaux IP
- Règles de sécurité : clef d'accès SSH, des groupes de sécu (firewall)
- IPs publiques
- ...



Comment ça marche derrière ?

La plate forme physique exécute un hyperviseur qui fournit les VMs

- vmware
- virtual box
- Linux/XEN (AWS, non std)
- Linux/KVM (std, DS Outscale, Open Stack, ...)
- MS Windows/HyperV

L'hyperviseur montre des VMs similaires à des machines physiques avec des drivers optimisés (ex. virtio sous Linux)

Le minimum à savoir pour démarrer avec la console Web d'Amazon

- Un VPC est toujours nécessaire, mais vous en avez un par défaut
- Pour définir une VM il faut spécifier une AMI (Amazon Machine Image), une installation type d'un OS précis
- On doit permettre l'accès à cette VMs pour l'admin et la prod
  - ssh pour l'admin
  - http/https pour une appli Web
- Pour les accès ssh il nous faut une paire de clef d'accès

## TP : Créer une VM avec la console Web AWS

- Générer une keypair ssh et enregistrer la clef privée sur notre station (GNU/Linux, Mac OS, evt sous MS Windows)
- Définir la VMs avec un groupe de sécurité qui permet l'accès SSH (c'est par défaut)
- Se connecter avec un client SSH sur cette VMs.

Première étape:  
créer une paire de clefs SSH  
nommée firstkeypair

- on obtient un fichier  
firstkeypair.pem  
(c'est la partie privée de la  
keypair)

Seconde étape :

créer l'instance d'une VM

- t1micro (petite)
- associer à la keypair
- Quelle AMI ?

## Instances/Lancer une instance, Ubuntu Server 18.04

- type par défaut
- t2.micro
- associé à firstkeypair

à partir d'un UNIX (Linux, Mac, ...)

- reserrer les droits sur le fichier  
firstkeypair.pem

```
$ chmod go= .../firstkeypair.pem
```

```
$ ssh -i ....firstkeypair.pem ubuntu@ip
```

où ip est l'ip de l'instance visible sur  
la console:

```
ubuntu@ip-172-31-36-78:~$ echo Cool
```

## Examiner la conf de notre instance

- mémoire libre, espace disque
- message du noyau (dmesg | less)
- ligne de commande du noyau  
(cat /proc/cpuinfo)

\* Peut-on savoir qu'on est dans une VM? et chez AWS ?

\* On a rien demandé côté réseau, quelle est notre ip (locale) ? dans quelle plage est-elle ?

\* Remarquez que seuls les accès ssh de l'extérieur sont possibles (sg-...)

Premier livrable TP (dans votre dépôt GIT)

- Livrer une doc claire et succincte dans un répertoire TP1 dans votre dépôt git
- README.md (format markdown)

\* Installer un serveur Web sur la VM (Apache2, NGINX)

\* Vérifier sur l'hôte lui-même qu'une page Web est bien accessible

\* Modifier le groupe de sécurité pour que cette page soit visible du monde entier, testez.

Parmi les propriétés d'une instance il y a un champ "user-data"

- peut être un script (bash) qui va être exécuté lors du PREMIER lancement de l'instance
  - apt -y update && apt -y upgrade
  - apt install ansible
  - activer des débôt deb et installer des trucs
  - déployer des clefs ssh privées
  - ...
  - touch /tmp/cloud-init-ok
- peut être plusieurs fichiers (un peu comme un mail avec pièces jointes)



à l'intérieur d'une instance on peut faire des calls http (pas ssl!) sur 169.254.169.254

et on obtient (selon l'url) des infos sur notre instance

- son hostname (dns d'amazon)
- son ip, son sec. group
- son user-data
- etc. (meta-données)

\* Ce genre d'IP 169.254.0.0/16 vous avez pu en voir dans d'autres ctxt

\* Cette plage correspond "officiellement" (RFC de l'Internet) à quoi? Quel rapport avec la choucroute?

\* Pourquoi AWS a choisi ce genre d'IP?

## SSH : problèmes et solution

- SSH utilise la crypto à clef publique pour
  - authentifier l'hôte (srv) et échanger des clefs (partagées) de session
  - authentifier l'utilisateur si il y a
    - une clef publique sur srv:~/.ssh/
    - la clef privé correspondante côté client

par ex (à ne pas faire) sur le client:

```
ssh-keygen -t rsa
```

```
-> crée id_rsa / id_rsa.pub
```

```
ssh-copy-id -i id_rsa.pub id@srv
```

```
-> copie sur le srv dans
```

```
authorized_keys la clef pub.
```

à la première connexion ssh (client)  
va enregistrer la clef publique  
d'hôte dans ~/.ssh/known\_hosts qui  
va être pollué par des clefs obsolètes

à la première connexion il est  
demandé confirmation que la clef  
publique d'hôte est la bonne  
créer un script :

```
ssh -i ../clef-prive \  
    -o StrictHostKeyChecking=no \  
    -o UserKnownHostsFile=/dev/null \  
    user@ip instance
```

-> ou adapter .ssh/config

• autre intérêt de ssh : exposer localement (sur votre poste) un service actif dans une instance sans ouvrir d'accès dans un security group

- vous avez installé un site web, vous voulez le tester, mais pas le rendre ouvert

- on peut créer un tunnel chiffré entre un port sur NOTRE poste vers un port de n'importe quel instance avec SSH

par ex:

```
[ssh -i .../clef-prive \  
  -L 127.0.0.1:8080:127.0.0.1:80 \  
    user@instance -f -N
```

le port 8080 sur VOTRE poste mène au port 127.0.0.1:80 dans l'instance

AWS propose un accès via API REST (<https://json>) qui est utilisée par des outils comme aws (cli) s3cmd boto3 terraform ...

Avec un compte AWS "normal" vous pouvez obtenir la paire de clef (rien à voir avec ssh) d'auth. sur la console


En mode "Educate" il faut aller sur vocareum pour les obtenir, il y a un token en plus qui expire régulièrement

-> on va devoir mettre à jour régulièrement le fichier  = homedir

~/.aws/credentials

vocareum / account details / show

copier coller dans ~/aws/credentials

 region=us-east-1

output=json

Installer awscli :

debian/ubuntu: apt install awscli

Mac: brew install awscli

pour vérifier que tout est ok:

aws ec2 describe-instances | less

-> nos instances

----

 tant que ceci ne marche pas le reste ne marchera pas non plus...

awscli (commande aws) permet de gérer à 100% ce qu'on déploie et gérer la conf des instances et autres

awscli est écrit en python et utilise différent modules (botocore)

on peut faire du python directement  
Il suffit d'avoir python3 installé  
(apt install python3 / brew install python3)

Exemple avec boto3 (répertoire CLI de mon dépôt git) :

crée un venv (pas d'install au niveau système) :

```
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $ pip install boto3
[installe boto3 dans venv/...]
$ ./testb3.py
...
```

pour désactiver un venv python:

deactivate

rm -rf venv # supprime le venv



# Terraform

- Part d'une descriptions des ressources à gérer : instances, vpc, réseaux, volumes, keypairs, sec. grp., etc.
- Plusieurs fichiers (à votre convenance) dans un répertoire
- On demande d'appliquer la conf décrite
- Terraform crée les ressources et enregistre dans un fichier (tfstate) l'état obtenu
- On peut modifier les fichiers de description et demander à les appliquer  
-> terraform calcule et réalise les opérations

# Installation

Terraform est écrit en Go (golang de Google)

(Go a été écrit chez Google par les auteurs historique d'UNIX et C)

Supporte: AWS, GCE, Azure

On télécharge le binaire pour notre plateforme et on le copie dans le répertoire ou dans un répertoire de notre PATH

[terraform.io/downloads.html](https://terraform.io/downloads.html)

on télécharge on dezippe et on le copie dans le répertoire de travail ou un répertoire du PATH (/usr/local/bin)

le binaire de terraform étant accessible (dans le PATH) ou bien même copié dans le répertoire du projet TF (il faudra faire ./terraform dans ce cas)

```
provider.tf
```

```
provider "aws" {  
    profile = "default"  
    region  = "us-east-1"  
}
```

```
# installe le connecteur aws (ou autre)  
$ terraform init
```

Un premier fichier décrit réseau, vpc  
et communication avec l'extérieurs  
D'autre contiennent les instances,  
clefs, groupes de sec.  
On peut structurer comme on veut !

Tf/provider.tf  
Tf/vpc.tf  
Tf/instance.tf

J'ai mis les clef dans ssh-keys  
(et pas dans le git)  
J'ai mis le script d'init dans Script

id\_rsa\_tfkeypair1

Pour appliquer ce modèle :

`git pull` (être à jour)

```
$ cd ../ssh-keys  
$ ssh-keygen -t rsa  
( nommée ./id_rsa_tfkeypair1 )  
$ cd ../Tf  
$ terraform init  
$ terraform plan  
$ terraform apply
```

Magique, non ?

terraform graph

g n rer le sch ma des d pendances de  
no ressources

necessite graphviz pour obtenir une  
image

terraform graph | dot -Tsvg > graph.svg

terraform permet de définir des variables

- déclarées dans un des fichier \*.tf
- type (string, list, map)
- valeur par défaut éventuelle
- utilisables dans le fichier .tf : var.nom

les valeurs peuvent être définies :

par la ligne de comm:

```
terraform apply -var region=eu-west-1
```

par une var. d'env:

```
TF_VAR_region=... terraform apply
```

dans un fichier :

- terraform.tfvars
- \*.auto.tfvars
- autre nom mais à spécifier sur la ligne de commande (terraform -var-file="toto.tfvars")



Bonne pratique dans un VPC: avoir un bastion

bastion: instance qui est la plus petite

- une keypair spécifique
- une ip publique (flottante ou élastiques)
- le port 22 ouvert (sec. group) pour tout le monde ou (parano) pour une ip source précise (entreprise, vpn, etc.)
- aucun autre port ouvert, rien d'installer dessus sauf des utilitaires de tests, de déploiement de conf (ansible, SALT, ...)

les autres instance ont leurs keypairs (partagées ou pas), n'ont pas le port 22 ouvert, ont une IP publique seulement si nécessaire (serveur Web, etc.)

En partant des fichiers versionné dans TFex on va créer dans TP4 un plan tf qui

- applique cette pratique (bastion)
  - un bastion
  - deux keypairs (bastion et instance)
  - des sec. grp. bien définis
- utilise des variables (inspirées par mon TP3-sol) pour simplifier la maintenance et l'utilisation du plan