

Autom

Livraison et déploiement continu



Déroulé



Jour 1

Introduction

Machines physiques



Jour 2-3

Machines virtuelles



Jour 4-6

Conteneurs



Jour 7-8

Orchestration



Jour 9-12

Gestion de configuration logicielle



Jour 13-14

DevOps

« Infrastructure as code »



Jour 15-20

Projet





Livraison et déploiement continu

Une introduction



Problématiques de la livraison et du déploiement continu

- Ça marche sur mon poste (développeur), donc j'ai fini !
- Description des dépendances (versions des librairies, prérequis, ...).
- Imprécision des procédures de déploiement.



Problématiques de la livraison et du déploiement continu

- Je développe dans un environnement, je teste (si je teste...) dans un autre environnement, je passe en préproduction dans à nouveau un autre environnement, et je mets en production dans encore un autre environnement.

[Environnement = système d'exploitation, version des librairies, versions des dépendances, ...]



Problématiques de la livraison et du déploiement continu

- La personne qui déploie (en test ou en production) n'est pas la personne qui développe. Elle n'est pas dans le même service, n'a pas les mêmes dépendances hiérarchique, ne parle pas le même langage.



Problématiques de la livraison et du déploiement continu

- Ce qui est manuel n'est pas répétable de manière fiable et peu coûteuse.



Problématiques de la livraison et du déploiement continu

- Les opérations purement techniques de déploiement sont longues, fastidieuses, et il est facile de se tromper.



Problématiques de la livraison et du déploiement continu

- Configuration as code.



Virtualisation

Une introduction



Virtualisation vs. physique

- Déployer sur une machine physique, dédiée, c'est bien, parfois nécessaire...
 - Licence logicielle au processeur/au cœur (coucou M. Oracle ☺) ;
 - Sécurité (machine isolée, ou en réseau mais avec une meilleure isolation, ...).



Virtualisation vs. physique

- Déployer sur une machine physique, dédiée, c'est bien, parfois nécessaire, mais cela a quelques défauts...
 - Ça tombe en panne ;
 - Comment je fais si la charge augmente au-delà de la capacité du serveur ;
 - Ça coûte cher (énergie, place, taux d'utilisation, ...) ;
 - C'est ... physique ;
 - Et c'est potentiellement plus lent (démarrage, par exemple).



Virtualisation vs. physique

- Déployer sur une machine physique, dédiée, c'est bien, parfois nécessaire, mais cela a quelques défauts... sans pour autant être fonctionnellement plus limité que la virtualisation.

[*l.e.*, tout ce qu'on fait en architecture virtualisée, on peut le faire en physique, pas forcément de la même manière, et certainement pas avec la même souplesse, facilité, vitesse, coût, mais, on peut.]



Virtualisation

- C'est ... vieux !
 - Dès les années 60 – 1960, hein, pas 1860 😊 .
 - http://pages.cs.wisc.edu/~stjones/proj/vm_reading/ibmrd2505M.pdf,
<http://www.leeandmelindavarian.com/Melinda/25paper.pdf>
 - Cela peut être fait de manière matérielle ou logicielle, ou un mélange des deux.
 - Pendant longtemps, de manière matérielle sur les grands systèmes IBM (mainframe), logicielle sur le reste, merci les brevets 😊



Virtualisation

- Mais, au fait, c'est quoi ?
 - J'ai un ordinateur (physique ... mais pas forcément), et je le transforme en un ou plusieurs ordinateurs (virtuels, là, par définition) ;
 - Sur ces ordinateurs virtuels, je peux installer le même système, ou des systèmes différents ;
 - Je peux contrôler l'allocation des ressources (mémoire, disques, CPU, ...).

[Je peux même émuler d'autres processeurs, si j'ai ce besoin. C'est toujours de la virtualisation, mais dans une acception un poil différente.]



Virtualisation

- « J'ai un ordinateur (physique ... mais pas forcément), et je le transforme en plusieurs ordinateurs. »
 - Si la virtualisation est complète, rien n'empêche effectivement de faire de la virtualisation sur un ordinateur lui-même virtuel.
 - Et c'est vieux, comme idée, 1960 comme évoqué plus haut (VM et VM/CMS). En gros, on prend un gros ordinateur, qu'on découpe par département, avec des quotas, et les départements peuvent redécouper et attribuer à leurs diverses entités, et ainsi de suite, c'est comme des poupées russes 😊
 - On appelle ça de l'imbrication ('nesting', pour faire bien en société).



Virtualisation

- Les machines virtuelles, c'est bien...
 - Une possible meilleure utilisation des ressources ;
 - Une plus grande souplesse ;
 - Des coûts plus faibles ;
 - Et, si les circonstances le permettent, on peut faire des trucs bien tout plein (déduplication, la seule vraie chose qu'on ne peut pas émuler avec des serveurs physiques, ou, du moins, qui dans ce cas est une pénalité et non un avantage).



Virtualisation

- Les machines virtuelles, c'est bien... mais parfois c'est trop.
 - On n'a pas toujours besoin d'exécuter des systèmes différents ;
 - Et, parfois, exécuter des systèmes différents, ou les mêmes, en fait, cela peut avoir un coût : licences, ce genre de chose, et on peut souhaiter l'éviter.
- En d'autres termes, on peut rechercher une isolation de plusieurs « instances » sur un même système (physique ou virtuel), sans le surcoût engendré par la virtualisation « classique ».
- (Et, jusqu'aux années 1990, il y avait les brevets sur la virtualisation 😊)



Virtualisation

- Les machines virtuelles, c'est bien... mais parfois c'est trop. D'où la recherche de solutions plus légères et souples, encore : les conteneurs.
 - Qui utilisent le système hôte (Windows, linux, ...), sans nécessiter de système d'exploitation séparé ;
 - Fondées sur l'isolation des ressources (processeur, mémoire, E/S, ...).
- Avec Docker (2013), mais les conteneurs sont plus anciens (Solaris zones, BSD jails, ... 2000 en gros, hors chroot, entre 1979 et 1982).



~~Virtualisation~~Conteneurs

- Mais, au fait, c'est quoi, un conteneur ?
 - Un processus dans un conteneur = un processus dans le système hôte (droits différents, seulement) ;
 - Imbrication possible (mais pas vraiment avec Docker en tant que tel, pas « multi-tenant » à ce jour, mais cela viendra peut-être) ;
 - Une notion de couches/calques ('layers' pour briller en société), pour ce qui est du stockage ;
 - Une philosophie un peu différente de celle typiquement associée à l'usage des machines virtuelles (micro services et, surtout, « *stateless* »).



Machines virtuelles vs. conteneurs

- C'est quoi les différences ?
 - (En dehors de la philosophie, évoquée précédemment.)
 - La sécurité (meilleure isolation des VM) ;
 - Les performances de démarrage (minutes pour les VM vs. secondes pour les conteneurs), qui modifient profondément l'usage ;
 - Le caractère jetable (on ne modifie pas en général un conteneur, on le jette et on le remplace par un autre, car cela ne « coûte » rien) ;
 - La gestion des dépendances (une application ou service unique sur un conteneur, vs. plusieurs applications sur une même VM).



Et donc ?



Et donc

- Et donc, donc, pourquoi cette longue présentation ?
 - Pour déployer en continu, il faut bien déployer quelque part.
 - Machine physique, machine virtuelle et conteneurs sont ce sur quoi on peut déployer (en continu ou pas).
 - On peut le faire « à la main » ou de manière automatisée.
 - Nous, on va faire les deux 😊



Reprenons



À la main

- À la main pour commencer, et pour voir les inconvénients :
 1. Sur une machine physique ;
 2. Sur une machine virtuelle ;
 3. Sur un conteneur.

[Et oui, on va faire les trois, joie.]



Quoi ?

- Dans un précédent cours les outils d'intégration continue (Jenkins, ...) ont été présentés. Nous n'allons donc pas ici les reprendre (mais nous allons y revenir, pour proposer d'autres utilisations).
- Ce que nous allons installer dans les prochains jours, c'est Squash-TM, avec une base de données MySQL ou PostgreSQL.
- C'est cette même application que nous allons installer dans les divers environnements (physiques, virtuels et conteneurs).



C'est où ?

- Pour Squash-TM, c'est par là :
 - <http://squashtest.org/fr>
- Pour MySQL, c'est par là :
 - <https://dev.mysql.com/>
- Et pour PostgreSQL, c'est par là :
 - <https://www.postgresql.org/>



Déploiement

Sur une machine physique



C'est à vous !

- Objectif :
 - Installer Squash-TM dans sa version la plus récente sur votre ordinateur, en utilisant comme base de données MySQL (dans une version compatible 😊).
- Moyens :
 - Un ordinateur et une connexion internet.
- Attendus :
 - Se connecter à Squash-TM, créer un nouveau dossier. Redémarrer l'ordinateur, se reconnecter, le nouveau dossier doit toujours être là.





Machines virtuelles

Les premiers pas



Préambule

- C'est presque pareil, mais il y a un petit préambule 😊
- Car, oui, comment je fais pour avoir un serveur virtuel ?



Infrastructure virtuelle

- Plusieurs solutions...
 - Utiliser un fournisseur de machines virtuelles (AWS, Azure, pour citer les principaux, mais il y a plein d'autre choix) ;
 - Utiliser une infrastructure dédiée propre (ou mutualisée), via un service interne ou un infogérant ou ... ;
 - Utiliser son poste de travail.



Infrastructure virtuelle

- Principe
 - Un « hyperviseur », qui va transformer le système d'exploitation en « système hôte » :
 - VMWare,
 - HyperV,
 - VirtualBox,
 - ProxMox,
 - KVM,
 - ... ;
 - L'hyperviseur propose une interface qui permet de créer des machines virtuelles (ou de lancer/... des machines existantes), les « systèmes invités ».



Hyperviseur

- En plus des fonctions basiques évoquées plus haut, et même si cela dépasse le cadre de la présente formation, les hyperviseurs évolués permettent aussi des choses amusantes comme :
 - Le déplacement « à chaud » d'une machine virtuelle d'un hôte à un autre ;
 - La prise d'instantanés ('snapshots') qui représentent l'état une machine virtuelle à un instant donné, et auquel on peut revenir à loisir.



Installation d'un hyperviseur

- Comme nos postes de travail sont sous Debian, nous allons utiliser ... Gnome Boxes (« Machines » en français).

(Si vous voulez faire chez vous, sous Windows, vous pouvez utiliser Hyper-V ou, sans Windows 10 Pro, VirtualBox, qui s'installe simplement. Ou installer Hyper-V ou VMWare qui existent en version gratuite mais qui alors se mettront en système principal, ce qui peut avoir des effets indésirables selon l'usage que vous faites de votre PC, en particulier pour les jeux.)



C'est à vous !

- Objectif :
 - Installer Gnome Boxes comme hyperviseur sur votre ordinateur.
- Moyens :
 - Un ordinateur supportant la virtualisation, et une connexion internet
- Attendus :
 - Lancer la console Gnome Boxes. Elle apparaît et est fonctionnelle.



Système invité

- Une fois l'hyperviseur installé, nous pouvons créer notre machine virtuelle (ou nos machines virtuelles...).
- Nous pourrions installer Windows sur ces machines invités, mais ce ne serait pas amusant. Et en plus il faudrait une licence. Nous allons donc installer Ubuntu.
- Cette installation va créer une image (pour simplifier, un fichier qui va contenir un disque dur virtuel, celui de la machine virtuelle, et qui contiendra Ubuntu et ses diverses applications et données).



De l'aide (pour le réseau 😊)

- L'image ISO est par là :
 - <\\192.168.2.11\autom1>



C'est à vous !

- Objectif :
 - Initialiser une VM Windows 10 (ou Ubuntu 16.04, comme vous voulez).
- Moyens :
 - Un ordinateur, avec un hyperviseur, et une connexion internet.
- Attendus :
 - Lancer la VM, s'y connecter. Ça marche.



Déploiement

Sur une machine virtuelle



C'est à vous !

- Objectif :
 - Installer Squash-TM dans sa version la plus récente sur la première VM, en utilisant comme base de données MySQL (dans une version compatible 😊).
- Moyens :
 - Un ordinateur, avec une VM lancée, et une connexion internet.
- Attendus :
 - Se connecter, créer un nouveau dossier. Redémarrer la VM, se reconnecter, le nouveau dossier doit toujours être là.



Déploiement

Sur deux machines virtuelles



Infrastructure virtuelle

- L'un des avantages d'une infrastructure virtuelle, c'est qu'on peut facilement créer des machines virtuelles.
- Jusqu'à présent, nous avons installé sur une unique machine, physique d'abord puis virtuelle.
- C'est bien, mais un peu dommage, et pas extraordinaire en terme d'architecture.
- Nous allons donc recommencer en utilisant le serveur de base de données sur notre hôte, et reconfigurer Squash-TM pour qu'il l'utilise.



Infrastructure virtuelle

- Nous avons jusqu'à présent parlé de machines virtuelles, mais, en fait, il est question d'infrastructure virtuelle, pas juste de machines virtuelles.
- En particulier, comme nos machines sont virtuelles, le réseau qui les lie est aussi possiblement en partie virtuelle (et, dans notre cas, il n'est que virtuel, les machines virtuelles étant hébergées sur le même serveur physique, notre ordinateur).



Infrastructure virtuelle

- Nos deux machines doivent communiquer entre elles.
- Pour cela, elles doivent connaître leurs adresses (ou, dans notre exemple, Squash-TM doit connaître l'adresse du serveur de base de données).
- Une solution « *quick&dirty* » est d'utiliser les adresses IP des machines. Mais ces adresses peuvent changer, après arrêt relance par exemple.
- Le mieux est de donner un nom aux diverses machines virtuelles, et de partager ces noms. Le système se débrouillera pour résoudre les adresses IP correspondantes.



Infrastructure virtuelle

- Avec Gnome Boxes, avec la configuration par défaut des machines virtuelles, les VM créées sont isolées les unes des autres.
- Elles sont également isolées du système hôte.
- De vrais ordinateurs, presque.
- L'exposition (ce qui est visible de l'extérieur) se fait explicitement.

On joue !





Infrastructure virtuelle

- Ce que nous venons de voir est un petit bout de ce qui est appelé « *Software Defined Network* » ou « SDN » pour les intimes, hors périmètre de la présente formation, mais qui est une des révolutions majeures apportée par la virtualisation : plus besoin de commutateur ou de routeur « physiques », ni de jongler avec des câbles physiques, tout peut être fait par logiciel, reconfiguré à la volée.
- (Il faut bien sûr câbler entre eux les serveurs physiques, mais, une seule fois, à l'installation.)



C'est à vous !

- Objectif :
 - Installer Squash-TM dans sa version la plus récente sur la première VM, en utilisant comme base de données PostgreSQL installé sur la seconde VM (dans une version compatible 😊).
- Moyens :
 - Un ordinateur, avec deux VM lancées, et une connexion internet.
- Attendus :
 - Se connecter à Squash-TM, créer un nouveau dossier. Redémarrer la première VM, se reconnecter, le nouveau dossier doit toujours être là. Éteindre la seconde VM, vérifier que ça ne marche plus 😊





Conteneurs



Préambule

- C'est presque pas du tout pareil, mais il y a un petit préambule 😊
- Car, oui, comment je fais pour avoir des conteneurs ?



Conteneurs

- Au contraire des infrastructures virtuelles, nul besoin d'installer un hyperviseur pour avoir des conteneurs, linux et Windows proposant dans leurs versions pas trop antiques des conteneurs en standard (ou modulo l'installation de quelques packages).
- Cependant, un standard s'est développé pour manipuler les conteneurs de manière uniforme, et nous allons l'utiliser.



Docker

- Docker est une implémentation de ce standard, et est disponible tant sur linux que sur Windows (et autres systèmes pas trop « exotiques »).
- C'est par là : <https://www.docker.com/>
- Nous allons l'installer sur une des machines virtuelles Ubuntu (c'est plus facile pour voir certaines choses, mais les commandes sont identiques sur Windows, standard oblige).



C'est à vous !

- Objectif :

- Installer Docker

- Moyens :

- Un ordinateur et une connexion internet.

- Attendus :

- Lancer « `docker run hello-world` » en ligne de commande. Ça marche.



Principes

- Le principe général est similaire à ce que nous avons rencontré avec les machines virtuelles. Il y a des images, qui représentent la configuration et l'état d'un conteneur, que nous pourrions « instancier » (lancer).
- Mais dans le détail c'est très significativement différent...



Principes

- Les images ne sont pas créées en partant de rien, et en installant un système d'exploitation, puis des applicatifs, puis ...
- Les images sont créées à partir d'un modèle, et d'un fichier décrivant les diverses actions à mener (installation de logiciels, configuration, ...).
- Le modèle est constitué de la même manière ! C'est en fait lui aussi une image, créée à partir d'un modèle, et d'un fichier décrivant les diverses actions à mener. Et ainsi de suite (jusqu'à `scratch` 😊).



Principes

- Il existe un référentiel d'images, le « docker hub », qui propose des milliers d'images qu'on peut réutiliser :

- <https://hub.docker.com/explore/>

[On peut bien sûr avoir son propre référentiel, et il n'est pas obligé de passer par ce référentiel pour partager ses images.]



Dockerfile

- Le fichier de configuration s'appelle « Dockerfile » (avec la majuscule, et sans point, par convention).
- Il commence toujours par une instruction FROM (le modèle), suivie des diverses opérations nécessaires à la construction de l'image (ENV, RUN, COPY, EXPOSE, CMD, ...).
- Une fois le fichier constitué, on construit l'image :

```
docker build -t nom_image repertoire # si repertoire contient le dockerfile
```



Dockerfile

- Un exemple d'un dockerfile créant une image avec php5 (!?#! 😊) :

```
FROM ubuntu:latest
```

```
ENV DEBIAN_FRONTEND noninteractive
```

```
RUN apt-get update
```

```
RUN apt-get -y install php5-fpm
```

```
RUN sed -i 's/listen = \/\var\/run\/php5-fpm.sock/listen = 9000/'  
/etc/php5/fpm/pool.d/www.conf
```

```
EXPOSE 9000
```

```
CMD ["/usr/sbin/php5-fpm", "-F"]
```



Dockerfile

- Le dockerfile est un script qui va être exécuté par la commande « `docker build` ». Les actions doivent être non-interactives, sinon cela ne fonctionnera pas comme attendu 😊
- D'où le « `-y` » dans les commandes « `apt-get` » par exemple.



Commandes de base

- Quelques commandes simples :

```
docker images # voir la liste des images présentes sur le système
```

```
docker pull nom_image # récupérer l'image du docker hub
```

```
docker run nom_image # instancier un conteneur à partir de l'image
```

```
docker ps # voir les instances en cours d'exécution
```

```
docker kill id_instance # « tuer » l'instance (« docker ps » pour voir l'ID)
```

- Des commandes plus évoluées, pratiques :

```
docker run --name mon_nom -d nom_image # instancier en tant que démon un conteneur
```

```
docker run --rm -it nom_image /bin/bash # instancier un conteneur et s'y connecter
```


On joue !





Déploiement

Sur un conteneur



À vous (presque 😊) !

(Indice : pour votre dockerfile, c'est presque les mêmes actions que celles que vous avez fait précédemment...)



À vous !

- Objectif :
 - Installer Squash-TM dans sa version la plus récente sur un conteneur, en utilisant comme base de données H2 (le système par défaut).
- Moyens :
 - Un ordinateur, avec docker installé, et une connexion internet.
- Attendus :
 - Lancer le conteneur (« `docker run mon_image` »). Se connecter à Squash-TM, créer un nouveau dossier. Ça marche.



Déploiement

Sur deux conteneurs



Infrastructure virtuelle

- Plus encore qu'avec les machines virtuelles, les conteneurs ont vocation à être créés dynamiquement et à discuter entre eux.
- Se baser sur des adresses IP ou sur des noms de domaines n'est pas vraiment envisageable (même si on peut le faire pour des essais), les conteneurs pouvant avoir des durées de vies courtes.
- En « docker » pur, une bonne pratique est de donner des noms aux instances (paramètre « `--name` ») et, pour les instances clientes, d'utiliser le paramètre « `--link` ».



À vous (presque 😊) !

(S'il vous plaît, n'installez pas vous-même votre instance de base de données, utilisez une image déjà faite 😊)

(bon, utiliser une image déjà faite pour Squash-TM n'est pas forcément une bonne idée, ou du moins pas l'objectif de l'exercice 😊)



À vous !

- Objectif :
 - Installer Squash-TM dans sa version la plus récente sur un conteneur, en utilisant comme base de données un autre conteneur avec MySQL.
- Moyens :
 - Un ordinateur, avec docker installé, et une connexion internet.
- Attendus :
 - Lancer les conteneurs. Se connecter à Squash-TM, créer un nouveau dossier. Ça marche. Arrêter le conteneur Squash-TM, le relancer, se reconnecter.



Persistence

- Bon, il y a un petit détail que nous n'avons pas vraiment abordé avec les conteneurs...
- Un tout petit détail.
- Si j'arrête mes conteneurs et que je relance, ben, je n'ai plus mes données précieusement saisies. Oops.
- Car, oui, les conteneurs sont jetables.



Persistence

- Du coup, comment je fais ?
 - Je fais un *mapping* vers des répertoires locaux ;
 - J'utilise des volumes ;
 - J'utilise un système de persistance tiers.



Persistence

- Par exemple, pour le conteneur Squash-TM, je n'ai pas besoin de stockage spécifique (sauf pour les logs, le cas échéant, mais je peux faire appel à un système de concentration des logs, comme RSYSLOG).
- Par contre, pour le conteneur MySQL ou PostgreSQL, c'est mieux si je conserve les données.
- L'idée est de « persister » ce qui est nécessaire et rien d'autre.



Persistance

- Comment faire ?

- Pour un mapping vers des répertoires locaux :

```
docker run ... --mount type=bind,source=/tmp,target=/app
```

- Cela veut dire quoi ? Que le répertoire « /app » du conteneur recouvre le répertoire « /tmp » du système. En d'autres termes, les fichiers dans « /tmp » sont visibles du conteneur, qui peut les manipuler (lecture, modification), et en créer de nouveaux (et des sous-répertoires, et tout).



Persistance

- Comment faire ?

- Pour un volume :

```
docker volume create mon-volume
```

```
docker run ... --mount source=mon-volume,target=/app
```

- Cela veut dire quoi ? Que le répertoire « /app » du conteneur sera préservé dans un volume « mon-volume ». Le volume est initialement vide, mais, tant qu'il n'est pas supprimé il conserve son contenu.



Persistance

- Je peux restreindre les mappings (lecture seule, ...).

- Pour un volume :

```
docker run ... --mount source=mon-volume,target=/app,readonly
```

- Pour un mapping vers des répertoires locaux :

```
docker run ... --mount type=bind,source=/tmp,target=/app,readonly
```

- Je peux partager des volumes entre conteneurs.



Persistance

- Je peux créer les volumes en avance, comme nous venons de le voir, mais nous ne sommes pas obligés de procéder ainsi. Nous pouvons passer un nom de volume pas encore créé. Il sera alors créé, et **initialisé** avec le contenu original du répertoire sur lequel il est attaché.
- Ainsi, si le conteneur contenait un fichier « abc » dans le répertoire « /app », et qu'on lance la commande suivante (sans avoir créé le volume au préalable) :

```
docker run ... --mount source=mon-volume,target=/app
```
- Le volume sera créé, et contiendra un fichier « abc » (l'initialisation n'est faite qu'une fois, à la création du volume).



Persistance

- Quelques commandes utiles pour gérer les volumes :

```
docker volume create mon-volume # créer un nouveau volume
```

```
docker volume ls # voir les volumes existants
```

```
docker volume rm mon-volume # supprimer un volume existant
```

- La documentation est par là :
 - <https://docs.docker.com/storage/volumes/>



À vous !

- Objectif :
 - Installer Squash-TM dans sa version la plus récente sur un conteneur, en utilisant comme base de données un autre conteneur avec MySQL, avec persistance des données.
- Moyens :
 - Un ordinateur, avec docker installé, et une connexion internet.
- Attendus :
 - Lancer les conteneurs. Se connecter à Squash-TM, créer un nouveau dossier. Ça marche. Arrêter les conteneurs, les relancer, se reconnecter. Le dossier est là.



Administration

- Nous avons abordé les commandes de base.
- Il peut être intéressant d'observer une instance, pour voir quels sont ses paramètres d'appel, ses mappings, ce genre de chose :

```
docker inspect nom_du_conteneur
```

- On peut faire la même chose avec les volumes :

```
docker volume inspect nom_du_volume
```



Administration

- Par défaut, avec « `docker ps` », on ne voit que les conteneurs actifs.
- Pour voir tous les conteneurs, y compris ceux qui ne sont plus actifs :

```
docker ps -a # ou docker ps -all
```

- La documentation est par là :
 - <https://docs.docker.com/engine/reference/commandline/ps/>



Administration

- Si un conteneur n'a pas été lancé avec l'option « --rm », même s'il est terminé, il reste dans un état dormant (on le voit avec « `docker ps -a` »).
- Pour le supprimer vraiment :

```
docker rm nom_du_conteneur_ou_id
```



Administration

- Pour récupérer une image, nous avons vu « `docker pull mon_image` ».
- Si on veut une version spécifique, on peut utiliser :

```
docker pull mon_image:version
```

- Pour actualiser une image :

```
docker pull mon_image # ou docker pull mon_image:latest
```

- La documentation est par là :
 - <https://docs.docker.com/engine/reference/commandline/pull/>



Administration

- Et si je n'ai pas récupéré la bonne image, ou que je n'en ai plus besoin, je peux faire le ménage :

```
docker rmi mon_image
```

- (Si elle n'est plus utilisée, bien sûr.)

On joue !





Récapitulons



Récapitulations

- Nous avons installé Squash-TM six fois. Au moins 😊
- C'est (un peu) pénible et fastidieux.
- (Pas qu'avec Squash-TM)
- On n'a pas automatisé (même si, dans le cas des conteneurs, on voit comment faire).
- Mais on devrait, parce qu'on a besoin d'automatiser (mises à jours, patch de sécurité, ...) !



Récapitulations

- Nous avons fait des choses réutilisables et partageables :
 - Nous pouvons donner notre image à un tiers ;
 - Nous pouvons partager notre image docker qui contient Squash-TM.
- Ce qui veut dire que, par exemple dans le second cas, nous pouvons déployer rapidement chez un autre client.





Reprenons (encore !)



Plus jamais ça !

- (Bon, j'exagère 😊)



Orchestration



Orchestration

- Nous avons créé nos machines virtuelles et conteneurs « à la main ». C'est très bien, mais ce n'est pas dynamique.
- Pour les machines virtuelles, comme leur cycle de vie est assez « long », elles peuvent être provisionnées à partir d'un outil comme Jenkins, lors des déploiements.
- Pour les conteneurs, des outils comme Jenkins ne sont pas si bien adaptés.



Orchestration

- De nombreuses solutions ont été essayées pour les conteneurs.
- Le gagnant à ce jour est Kubernetes (jusqu'à ce qu'un autre passe devant, cela peut évoluer rapidement 😊).
- On trouve également des solutions comme Mesos ou Docker Swarm.
- Ou docker-compose, plus rudimentaire mais intéressant pour comprendre les principes (et partie intégrante de docker, donc « standard »).



Docker-compose

- Si on n'utilisait qu'un conteneur à la fois, tout serait simple (et, bon, l'intérêt serait plus ... limité).
- Mais l'idée, c'est d'avoir plein de conteneurs « élémentaires », qui discutent entre eux.
- Et donc d'explicitier ces échanges.
- Et de pouvoir dire, in fine, quelque chose comme :

```
commande_magique up  
commande_magique down
```



Docker-compose

- La commande magique s'appelle « `docker-compose` ». Bon, vous l'aviez deviné.
- Et le moyen pour décrire les relations entre nos conteneurs, c'est un fichier texte, tout bête (bon, en YAML, joie), qui s'appelle par défaut « `docker-compose.yml` ».

(On n'est pas obligé d'utiliser ce nom, mais, si on ne l'utilise pas, on doit donner explicitement le nom lors de l'appel de la commande magique.)



Docker-compose

- Un exemple de fichier `docker-compose.yml` :

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - ./code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```



Docker-compose

- La documentation est par là :
 - <https://docs.docker.com/compose/overview/>



À vous !

- Objectif :
 - Créer un fichier `docker-compose.yml` qui va organiser (« orchestrer ») nos deux conteneurs, Squash-TM et MySQL.
- Moyens :
 - Un ordinateur, docker, deux images (MySQL et Squash-TM), une connexion Internet.
- Attendus :
 - « `docker-compose up` » lance notre infrastructure, et « `docker-compose down` » l'arrête, et on peut relancer, sans perte de données.
 - Et, pour jouer, on peut en lancer en parallèle 😊



Progrès



Récapitulations

- Nous avons automatisé !
 - Sur la partie conteneurs, tout est sous forme de code (notre application, nos dockerfiles, et même nos instructions de déploiement, notre fichier docker-compose.yml).
 - Tout peut donc être confié à un système de gestion de version, et le déploiement de notre solution est automatisé.
 - Et les mises à jour sont simples (ou, du moins, simplifiées, il reste la migration des données, plus du côté applicatif).



Récapitulations

- Mais il reste encore le cas des machines virtuelles (ou physiques).
- Et je reste aveugle.





Gestion de configuration logicielle



Gestion de configuration logicielle

- C'est quoi, la gestion de configuration logicielle ?
 - Souvenir du cours sur l'intégration continue (« CI »)...
 - CVS,
 - SVN,
 - Git,
 - ... ;
 - Oui, mais pas seulement...



Gestion de configuration logicielle

- C'est (très) bien d'avoir l'historique de ses versions logicielles.
- Mais ce n'est pas tout.
- C'est bien de savoir aussi quelle version est déployée où.
- C'est aussi très bien de pouvoir mettre à jour (patch de sécurité, par exemple), automatiquement.
- Ou plus simplement d'installer automatiquement 😊



Gestion de configuration logicielle

- Il y a bien sûr des limites.
 - C'est raisonnable au sein d'une entreprise, ou au sein d'un fournisseur de service (« SaaS » ou « Paas », pour employer des termes qui font bien) ;
 - C'est plus discutable lorsque les machines cibles sont chez d'autres clients ;
 - Ce n'est tout simplement pas possible pour des machines cibles non connectées ;
 - Cela peut avoir un coût (volume de données, sécurité, ...).



Gestion de configuration logicielle

- Deux grandes philosophie :
 - Le client (la machine « cible ») interroge un service central ('pull') ;
 - Le service central se connecte aux clients ('push').
- Il y a des avantages et des inconvénients dans les deux cas. Pas de solution miracle...
- Par contre il y a toujours historisation.

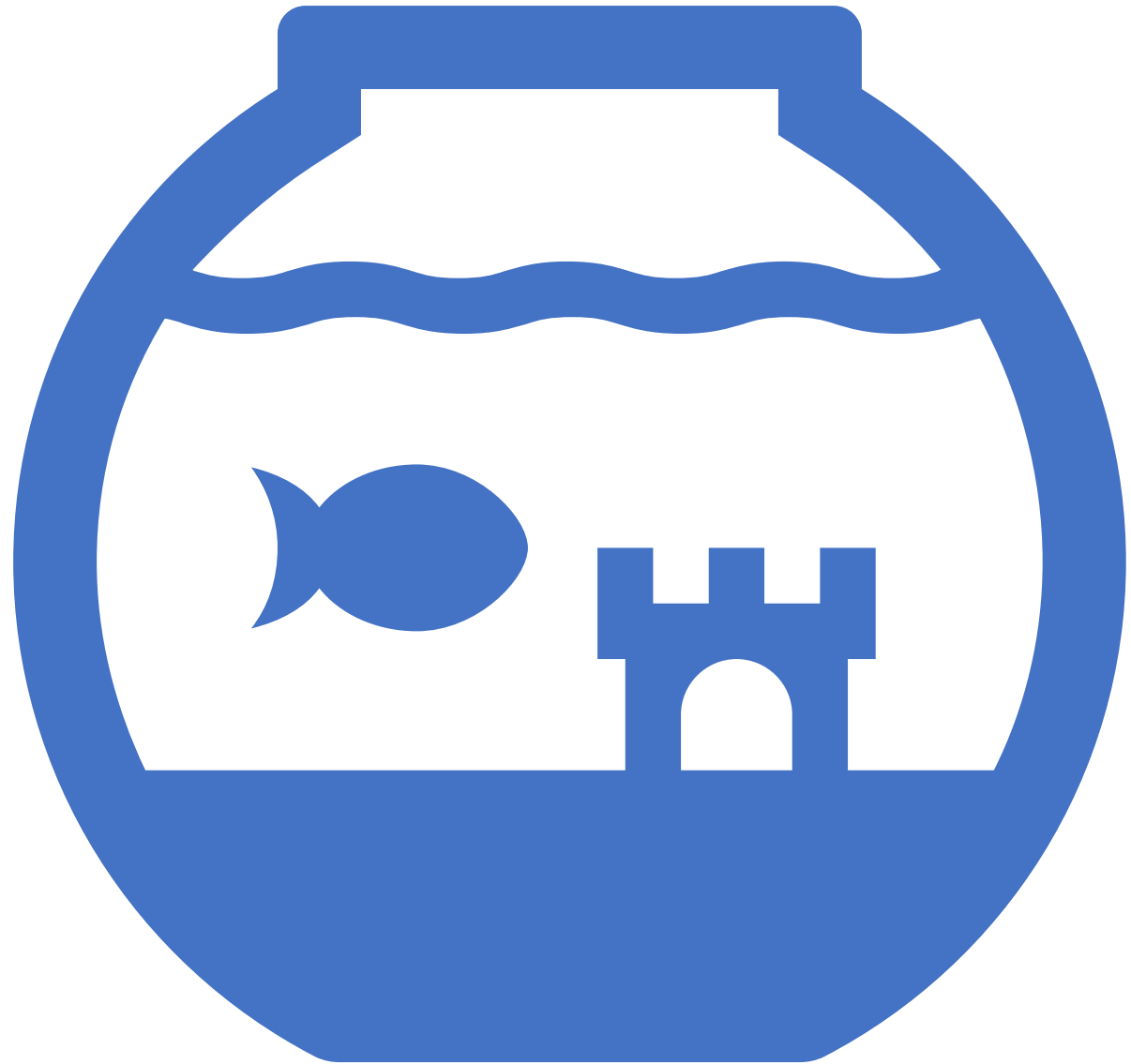


Gestion de configuration logicielle

- Quelques outils :
 - Ansible,
 - Puppet,
 - Chef,
 - DSC ('Desired State Configuration', pour PowerShell).
- Ce sont des outils orientés « machines virtuelles » et « machines physiques », pour les conteneurs la problématique est comme évoqué différente (mais toujours besoin d'historisation).



Ansible





DevOps



C'est quoi?

“DevOps is the practice of operations and development engineers participating together in the entire service lifecycle, from design through the development process to production support.

DevOps is also characterized by operations staff making use many of the same techniques as developers for their systems work.”



C'est quoi?

“DevOps is the practice of operations and development engineers participating together in the entire service lifecycle, from design through the development process to production support.

DevOps is also characterized by operations staff making use many of the same techniques as developers for their systems work.”

La première partie est purement organisationnelle...



C'est quoi?

“DevOps is the practice of operations and development engineers participating together in the entire service lifecycle, from design through the development process to production support.

DevOps is also characterized by operations staff making use many of the same techniques as developers for their systems work.”

... mais il y a peut-être quelques choses à connaitre au propos de la seconde. En particulier, c'est quoi les « techniques » des développeurs ?



Les principes

- *The Three Ways*
 - *Systems Thinking*
 - *Amplified Feedback Loops*
 - *Culture of Continual Experimentations*
- *CAMS*
 - *Culture – People > Process > Tools*
 - *Automation – Infrastructure as Code*
 - *Measurement – Measure Everything*
 - *Sharing – Collaboration/Feedback*



Les pratiques

- *Version Control for All*
- *Automated Testing*
- *Proactive Monitoring and Testing*
- *Kanban/Scrum*
- *Visible Ops/Change Management*
- *Configuration Management*
- *Incident Control System*
- *Continuous Integration/Deployment/Delivery*
- *“Put Developers on Call”*
- *Virtualization/Cloud/Containers*
- *Toolchain Approach*
- *Transparent Uptime/Incident Retrospectives*



Dev vs. Ops

- Ne pas faire les tâches des autres. Construire les outils qui vont leur permettre de faire leur travail.
- Monitorer/Logger/Mesurer et partager avec les développeurs (et le métier)
- Des postmortems après incidents, sans chercher à pointer un responsable
- *Developers Do Production Support/Empower Ops Acceptance.*



Dev



Environnement de développement

- vi c'est bien, vim c'est mieux (le 'm' c'est pour « improved » 😊)
- Notepad c'est bien, Notepad++ c'est mieux, mais SublimeText ou Code ou ... c'est encore mieux.
- La colorisation syntaxique c'est sympa, mais ce n'est pas tout.
- Linters, complétion, ...



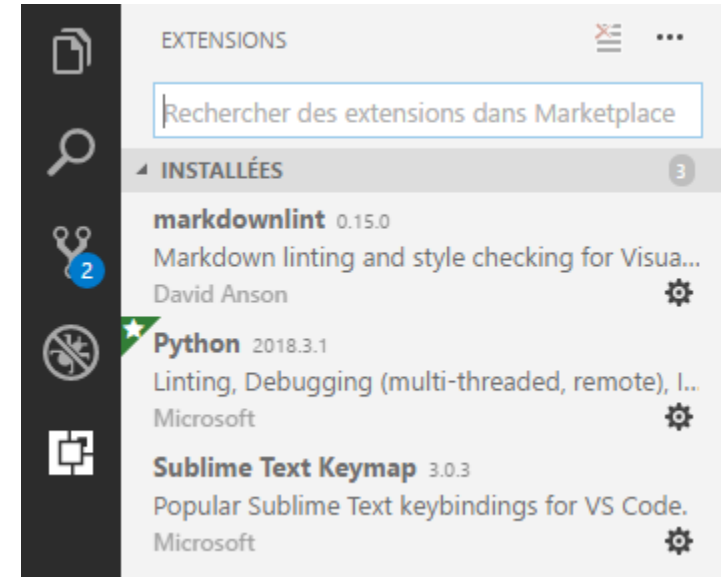
VisualStudio Code

- Religion Wars are nothing compared to Editor Wars.
- Open Source, gratuit (et libre), cross-plateforme, excellente intégration avec Git, support Python agréable, ...
- (Et c'est plutôt amusant de voir des « Microsoft haters » utiliser et recommander cet outil 😊)



VisualStudio Code

- L'extension "Python" est agréable.
 - Encore plus utile lorsque Python est installé sur le poste de travail...
 - Encore mieux avec un linter (pylint par exemple).
- Utilisez Python 3.x (disons, 3.6).
- Il peut s'avérer nécessaire de définir un proxy dans les paramètres de VisualStudio Code :



```
{  
  "git.path": "C:/Apps/PortableGit/bin/git.exe",  
  "sublimeTextKeymap.promptV3Features": true,  
  "editor.multiCursorModifier": "ctrlCmd",  
  "editor.snippetSuggestions": "top",  
  "editor.formatOnPaste": true,  
  "http.proxy": "http://10.24.158.1:8080",  
  "workbench.colorTheme": "Default Light+",  
  "python.linting.pylintUseMinimalCheckers": true,  
}
```



VisualStudio Code

- Il va aussi créer un répertoire `.vscode` à la racine de vos projets. Pensez à l'exclure dans votre gestionnaire de code source (ajoutez le répertoire dans le `.gitignore` de votre projet si vous utilisez Git, par exemple).

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]

# Credentials
credentials.json

# VisualStudio noise
.vs/
src/src.pyproj

# VisualStudio Code noise
.vscode/
```



VisualStudio Code

- So What?



```
def projects():
    """
    .st ava def sorted(iterable, cmp=None, key=None, reverse=False)
    irams ! Built-in functions, exceptions, and other objects.
    ibort(' Noteworthy: None is the 'nil' object; Ellipsis represents '...' in slices.
    :ey in sorted(pf.get_managedprojects()):
    rint(key)

    usage()
    [pylint] E1101:Instance of '_EnginePlatform' has no 'foo' member
    if c
    pf: _EnginePlatform
    pf.foo
```

```
params = len(sys.argv)
cmd = s
actions
    clients
        create_managedproject
        credentials
        get_canonical_user_id
        get_internal_user_id
        get_managedproject
        get_managedprojects
        get_push_strategy
        get_rollback_strategy
        get_user
        get_users
    if cmd
        pf
        platform
    pf.
    actions[cmd]()
```



VisualStudio Code

- Et bien plus encore...
- Vous pouvez vouloir affiner la configuration de votre linter.
<https://code.visualstudio.com/docs/languages/python>
- Vous pouvez regarder le tutoriel python.
<https://code.visualstudio.com/docs/python/python-tutorial>
- Oh, et GitLens. Regardez GitLens si vous utilisez Git.



Les bases de la programmation

- Structure (de données), structure (de données), structure (de données)
- Logs, logs, logs
- Vérification, vérification, vérification
- Commentaires (mais pas commentaires, commentaires, commentaires)
- « Small is Beautiful »
- Fonctions, fonctions, fonctions (et peut-être classes/méthodes, mais seulement si nécessaire)
- Se référer à un contexte implicite, c'est souvent mal (i.e., pas de variables globales, ...)



Infrastructure as Code



Vers l'infini et au delà

- Déployer une application ou un SI, de manière automatisé, c'est bien.
- Mais une application ou un SI, c'est possiblement plus qu'installer des logiciels sur un ou des serveurs (physiques ou virtuels, conteneurs ou pas).
- Nous avons esquissé le sujet lorsque nous avons fait communiquer deux machines virtuelles entre elles (et lorsque nous avons fait la même chose avec des conteneurs).



Infrastructure

- Une application ou un SI, c'est parfois un serveur unique, mais pas toujours.
- C'est souvent, dans les cas qui nous intéressent ici, plusieurs machines, qui discutent entre elles (deux ou trois au minimum).

(Après tout, lorsqu'on parle d'applications 3-tiers, cela signifie souvent cela, même si pour des tests on peut tout mettre sur une même machine.)



Infrastructure

- Mais ce n'est pas juste ça. Dès qu'on parle de plusieurs machines, se pose la question de comment elles discutent entre elles.
- En général, ce ne sont pas des discussions à la cantonade (« open bar », si vous préférez). Ces discussions se font via des canaux précis, documentés.
- On pourrait s'en approcher et définissant des règles de filtrage sur chaque serveur, mais ce n'est pas suffisant.



Infrastructure & sécurité

- Pas suffisant, par exemple, parce qu'une personne indésirable, si elle accédait à l'un de nos serveurs, pourrait modifier ces règles, et voir ou modifier des choses sur d'autres serveurs.



Infrastructure & sécurité

- D'où le souhait d'isoler « physiquement » nos serveurs, en les mettant des zones différentes, avec un contrôle d'accès effectué par un service tiers.
- Ou de mettre un contrôle en amont de notre application, pour ne permettre l'accès qu'à des personnes autorisées.

(On pourrait confier ce contrôle à la partie « présentation » de notre application, mais c'est prendre le risque de la complexifier, et possiblement de l'empêcher de répondre à des demandes valides.)



Infrastructure

- La sécurité n'est pas le seul élément qui entre en compte dans la détermination de l'infrastructure. Il y a aussi les coûts. Et les performances, pour compléter le triptyque bien connu.
 - Performance
 - Sécurité
 - Coûts
- On ne peut en choisir qu'au plus deux 😞



Infrastructure pour les tests et la qualification

- Il n'est pas toujours possible (ou souhaitable) de tester directement sur l'environnement (l'infrastructure) de production.
- Du coup, comment peut-on s'assurer que les environnements utilisés sont isomorphes (car, s'ils ne le sont pas, on ne teste ou qualifie pas vraiment) ?



Bref...

- L'infrastructure fait partie intégrante de l'application ou du SI.
- Elle évolue avec le temps :
 - avec la vie de l'application ;
 - de son utilisation, aussi.

(Et, du coup, elle suit les évolutions de l'application, mais pas seulement, elle peut être amenée à évoluer même si l'application n'est pas modifiée.)



Alors on fait quoi ?

- On fait comme pour une application.
 - On peut ne rien faire ;
 - On peut faire une jolie documentation qui donne les prérequis et les instructions de configuration ;
 - Ou on décrit ces éléments dans un format qui sera exécuté automatiquement.
- Avec les mêmes motivations que pour les applications.



Quel format ?

- (Parce que, oui, bien sûr, c'est la troisième option qui nous intéresse)
- Ça dépend du contexte, de l'environnement cible global de l'entreprise :
 - On-premise
 - Cloud
 - Hybride
 - ...



Quelques formats

- Pas de standard « officiel » largement utilisé
 - (OASIS TOSCA, en 2014, <http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html>)
- Des standards « de fait » :
 - AWS Cloudformation
 - Terraform
 - ...
- Domaine encore « jeune », qui évolue rapidement.



Quelles caractéristiques ?

- Du texte, structuré (JSON, YAML, du code, ...) ;
 - Qui peut être versionné, historisé, archivé (comme du code) ;
 - Dont on peut suivre les évolutions ;
 - Que l'on peut vérifier, contrôler ;
 - Que l'on peut appliquer, déployer, paramétrer.
-
- Et peut-être « agnostique » (i.e., ne dépendant pas d'une plate-forme cible spécifique).



Quelles cibles ?

- Par nature une cible qui peut être contrôlée automatiquement, via une API donc (Application Program Interface).
- Quelques exemples :
 - AWS (Amazon)
 - Azure (Microsoft)
 - GCP (Google)
 - OpenStack
 - VMWare
 - ...



Quel lanceur ?

- (En d'autres termes, comment on déploie l'infrastructure)
- Ligne de commande (je lance un script qui effectue le déploiement)
- Interface graphique (je clique sur un schéma d'infrastructure, il se déploie)
- Ordonnanceur (Jenkins, cron, ...)
- En réaction à un événement (lambda, i.e., serverless)

Terraform



C'est quoi ?

- Une (une) solution qui permet de faire de l'« infrastructure as code ».
- Open source (avec une version payante, les entreprises aiment bien les trucs payants, cela permet d'avoir du support).
- Par là : <https://www.terraform.io/>

(Oui, un truc de HashiCorp, ceux qui font Packer, Consul et Nomad.)



Ce n'est pas quoi ?

- Ce n'est pas un système de gestion de configuration logicielle (Ansible, Puppet, ...).
 - Cela ne déploie pas un applicatif, cela déploie une infrastructure sur laquelle l'applicatif sera déployé, avec par exemple Ansible, Puppet, Docker, ...
- Ce n'est pas une API.
- Ce n'est pas un clickodrome.



C'est comme quoi, alors ?

- C'est comme Cloudformation (AWS), par exemple, sauf que c'est multi-plateforme.
- Cela permet de ne pas être dépendant d'un seul fournisseur, et d'avoir une infrastructure qui en recouvre plusieurs, donc.



Pourquoi tout le monde n'utilise pas ça ?

- Parce que tout le monde n'est pas intéressé par du multi-plateforme.
- Parce qu'il peut y avoir un décalage entre les services dont on a besoin et les fonctionnalités du produit.
- Parce que la diversité, c'est bien.



Les principes

- Infrastructure as Code
- Plans d'exécution
- Graphe des ressources
- Changements automatiques



Infrastructure as Code

- Ben ouais, sinon nous n'en parlerions pas ici 😊



Plans d'exécution

- On ne construit pas toujours les environnements à partir de zéro.
- En production, par exemple, souvent, on applique des modifications, sans tout détruire.
(Oui, comme avec docker-compose, pas besoin de détruire avant de reconstruire, s'il n'y a pas de changement sur un service / ressource)
- Et voir ce qui a changé, avant de le mettre en œuvre, c'est, euh, bien.
(Parfois, sinon, on peut se rendre compte trop tard qu'on s'est trompé 😊)



Graphe des ressources

- Pour voir ce qui dépend de qui, de quoi, pour créer/modifier/... en parallèle, pour gagner du temps.
- Pour explorer et mieux comprendre son infrastructure.



Changements automatiques

- Pour éviter les erreurs humaines.
- Pour fiabiliser et accélérer les déploiements.



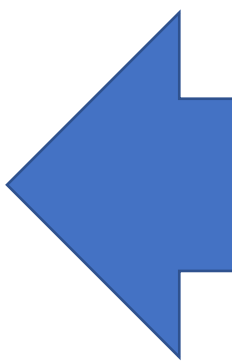
À quoi cela ressemble ?

- À ça :

```
# An AMI
variable "ami" {
    description = "the AMI to use"
}

/* A multi
   line comment. */
resource "aws_instance" "web" {
    ami          = "${var.ami}"
    count        = 2
    source_dest_check = false

    connection {
        user = "root"
    }
}
```



Il y a aussi une version JSON, qui n'est guère promue (eh oui, même en open source on aime bien les formats « propriétaires » 😞)



On joue !

(Ensemble, pour commencer 😊)



On joue !

- C'est par là pour le téléchargement :
 - <https://www.terraform.io/downloads.html>
- S'assurer que terraform est bien dans le PATH
 - export PATH=\$PATH:/path/to/dir dans ~/.bashrc ou ~/.profile (pour les dinos)
 - Control Panel -> System -> System settings -> Environment Variables pour Win



On joue !

- On teste l'installation

- terraform

- Doit donner un résultat qui n'est pas une erreur, et qui ressemble à ça :

```
$ terraform
Usage: terraform [--version] [--help] <command> [args]
```

```
The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.
```

```
Common commands:
```

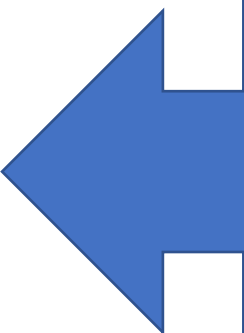
apply	Builds or changes infrastructure
console	Interactive console for Terraform interpolations
[...]	



On joue !

- Créer un répertoire, et s'y placer.
- Y créer un fichier `essai.tf` contenant la chose suivante :

```
provider "aws" {  
  access_key = "ACCESS_KEY_HERE"  
  secret_key = "SECRET_KEY_HERE"  
  region     = "us-east-1"  
}  
  
resource "aws_instance" "example" {  
  ami           = "ami-2757f631"  
  instance_type = "t2.micro"  
}
```



Pour les valeurs de `access_key` et `secret_key`, prendre celles données dans le fichier sur le répertoire partagé. **(Ne JAMAIS stocker ces infos dans un gestionnaire de code source)**



On joue !

- Initialisation
 - terraform init
- Cela initialise et configure l'environnement terraform qui sera utilisé pour les commandes ultérieures.
- En gros à faire la première fois, et lorsqu'on récupère du code d'un système de gestion de code source.



On joue !

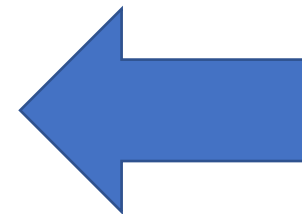
- Application des changements (tout est nouveau ici, naturellement)

- terraform apply

(Pour les versions récentes de terraform, avant il fallait faire au préalable un « terraform plan ». On peut toujours, mais ce n'est plus nécessaire depuis quelques versions.)

```
$ terraform apply
# ...
```

```
+ aws_instance.example
  ami: "ami-2757f631"
  availability_zone: "<computed>"
  ebs_block_device.#: "<computed>"
  ephemeral_block_device.#: "<computed>"
  instance_state: "<computed>"
  instance_type: "t2.micro"
```



Permet de voir ce qui est nouveau (et va être créé, donc), inchangé, et supprimé.
(« computed » veut dire que la valeur n'est pas connue, et ne le sera qu'une fois l'infra déployée)



On joue !

- S'il n'y a pas eu d'erreur dans la première phase, terraform attend confirmation et, le cas échéant, effectue le déploiement.
- Le déploiement prendra quelques minutes, terraform attendant la disponibilité effective de l'instance.

```
# ...
aws_instance.example: Creating...
  ami:                "" => "ami-2757f631"
  instance_type:      "" => "t2.micro"
  [...]

aws_instance.example: Still creating... (10s elapsed)
aws_instance.example: Creation complete

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

# ...
```



On joue !

- Une fois le déploiement effectué, on peut voir l'état actuel :
 - terraform show

```
$ terraform show
aws_instance.example:
  id = i-32cf65a8
  ami = ami-2757f631
  availability_zone = us-east-1a
  instance_state = running
  instance_type = t2.micro
  private_ip = 172.31.30.244
  public_dns = ec2-52-90-212-55.compute-1.amazonaws.com
  public_ip = 52.90.212.55
  subnet_id = subnet-1497024d
```



On joue !

- Si notre image est « autoporteuse » (faite avec Packer, par exemple, ou si on souhaite simplement le déploiement d'une AMI, sans rien de plus), c'est tout.
- Mais, souvent, on va vouloir déployer notre applicatif sur l'infrastructure nouvellement déployée (avec ansible par exemple, coucou ansible 😊)



On joue !

- Et donc on casse tout (nous sommes de grands enfants 😊).
- Bon, pas tout de suite, on va d'abord modifier notre exemple, en changeant d'AMI (on passe d'une Ubuntu 16.04 LTS à une 16.10).
- On remplace donc l'ami dans notre fichier `essai.tf`.

```
resource "aws_instance" "example" {  
    ami          = "ami-b374d5a5"  
    instance_type = "t2.micro"  
}
```



On joue !

- Et on applique nos changements...
- terraform apply

```
$ terraform apply  
# ...
```

Le « -/+ »
signifie
suppression
puis
création

```
-/+ aws_instance.example  
ami: "ami-2757f631" => "ami-b374d5a5" (forces new resource)  
availability_zone: "us-east-1a" => "<computed>"  
ebs_block_device.#: "0" => "<computed>"  
ephemeral_block_device.#: "0" => "<computed>"
```



On joue !

- S'il n'y a pas eu d'erreur dans la première phase, terraform attend confirmation, comme précédemment, et, le cas échéant, effectue le déploiement.
- Le déploiement prendra quelques minutes, vu qu'il y a suppression et recréation (terraform attendant la disponibilité effective de l'instance).



On joue !

- Et on casse tout (pour de vrai, cette fois)
 - terraform destroy
- Et comme pour la création, après confirmation, les changements (la suppression, ici) sont mis en œuvre.
- Terraform s'occupe de l'ordre dans lequel appliquer lesdits (bon, ici, avec une seule ressource, facile 😊)



On joue !

- Pour le moment, notre environnement est minimal :

- Un « fournisseur »

```
provider "aws" {  
    access_key = "ACCESS_KEY_HERE"  
    secret_key = "SECRET_KEY_HERE"  
    region     = "us-east-1"  
}
```

- Une machine virtuelle

```
resource "aws_instance" "example" {  
    ami           = "ami-2757f631"  
    instance_type = "t2.micro"  
}
```




On joue !

- Mais, souvent, dès qu'on a plus d'une ressource, des dépendances apparaissent.
- Par exemple, on peut vouloir attribuer une adresse IP « élastique » à notre VM (une IP « élastique », c'est une adresse fixe, mais qui désigne une machine virtuelle qui peut changer).
- Pour cela, il faut pouvoir dire que cette nouvelle ressource « dépend » de notre machine virtuelle.



On joue !

- Ajoutons la chose suivante dans notre fichier `essai.tf` :

```
resource "aws_eip" "ip" {  
  instance = "${aws_instance.example.id}"  
}
```



« Type de la ressource » . « nom de la ressource » . id

Pour plus d'information sur la syntaxe des références,
<https://www.terraform.io/docs/configuration/syntax.html>



On joue !

- Et on réapplique notre configuration.
- Ici, la dépendance est implicite (le champ 'instance' de l'EIP indique la machine virtuelle), donc terraform saura qu'il convient de créer le cas échéant la machine virtuelle avant d'y attacher l'EIP.

```
# ...
aws_instance.example: Creating...
  ami:                  "" => "ami-
b374d5a5"
  instance_type:       "" => "t2.micro"
  [...]
aws_instance.example: Still creating... (10s
elapsed)
aws_instance.example: Creation complete
aws_eip.ip: Creating...
```



On joue !

- Mais, dans d'autres cas, la dépendance ne peut être devinée. Il convient alors de l'indiquer explicitement :

```
resource "aws_s3_bucket" "example" {  
  # NOTE: S3 bucket names must be unique across _all_ AWS accounts, so  
  # this name must be changed before applying this example to avoid naming  
  # conflicts.  
  bucket = "terraform-getting-started-guide"  
  acl    = "private"  
}  
  
resource "aws_instance" "example" {  
  ami          = "ami-2757f631"  
  instance_type = "t2.micro"  
  
  # Tells Terraform that this EC2 instance must be created only after the  
  # S3 bucket has been created.  
  depends_on = ["aws_s3_bucket.example"]  
}
```



On joue !

- Il n'y a pas toujours de dépendance explicite ou implicite, des ressources peuvent être indépendantes.
- Dans ce cas, rien de spécial à faire, pas de « depends_on » à ajouter...



On joue !

- Déployer une infrastructure, c'est bien, mais, bon, nous, on veut installer des trucs dessus automatiquement.
- C'est vrai, quoi, si on est obligé de se connecter à la main sur la nouvelle infrastructure, c'est triste.



On joue !


- Du coup, comment on fait ?
- Terraform a une notion de « provisionneur ».
- Un provisionneur, c'est une commande (ou une suite de commande) qui se déclenchera une fois la ressource associée provisionnée.
- Allons faire nos provisions...



On joue !

- Modifions notre fichier `essai.tf`

```
resource "aws_instance" "example" {  
    ami          = "ami-b374d5a5"  
    instance_type = "t2.micro"  
  
    provisioner "local-exec" {  
        command = "echo ${aws_instance.example.public_ip} > ip_address.txt"  
    }  
}
```



« local-exec » indique une commande qui va s'exécuter sur le poste local (celui à partir duquel on a lancé terraforme).



On joue !

- Les provisionneurs ne s'exécutent qu'à la création d'une ressource. Si elle ne change pas, ils ne sont pas exécutés.
- Donc, terraform destroy et terraform apply, sinon, il ne se passera rien, snif 😞
- Dans notre exemple, rien ne change dans le résultat sur la console. Mais, si tout va bien, un fichier a été créé avec l'adresse IP de la machine virtuelle

```
$ cat ip_address.txt  
54.192.26.128
```



On joue !

- Et si tout ne se passe pas bien lors de l'exécution du provisionneur ?
- La ressource est créée, et le reste, mais terraform la marque comme étant « tainted » (polluée, contaminée, ...)
- Comme ça, pas d'ambiguïté, on sait qu'un truc ne s'est pas bien passé.



On joue !

(Les provisionneurs se déclenchent par défaut après le provisionnement d'une ressource. Il est aussi possible de spécifier des « nettoyeurs », des commandes qui seront exécutées juste avant la destruction d'une ressource, même si il est souvent préférable de définir ces actions autrement.)



On joue !

- Bon, c'est bien, mais tout est dans un fichier, c'est pas top, je veux de la modularité.
- Et, je voudrais ne pas avoir mes `secret_key` / `access_key` dans mon code non plus, ce n'est pas propre, pas sûr.



On joue !

- Dans un nouveau fichier .tf (du nom que vous voulez, mais dans le même répertoire), mettre :

```
variable "access_key" {}  
variable "secret_key" {}  
variable "region" {  
  default = "us-east-1"  
}
```


{ } est un bloc vide (qui sera défini ailleurs, donc)
{ default = « ... » } c'est pareil, sauf qu'il y a une valeur par défaut si rien n'est spécifié par ailleurs.



On joue !

- Puis, dans notre fichier `essai.tf`, on modifie la section « provider »

```
provider "aws" {  
  access_key = "${var.access_key}"  
  secret_key = "${var.secret_key}"  
  region     = "${var.region}"  
}
```



« var », c'est pour « variable »
(on aurait pu ne pas définir un fichier séparé, et mettre les sections précédentes dans notre fichier, mais c'est une bonne pratique de séparer)



On joue !

- Et après, au moment du déploiement, on peut spécifier les valeurs qui nous intéressent...
- Via la ligne de commande
- Ou via un (ou des) fichiers
- Ou via les variables d'environnement
- Ou de manière interactive



On joue !

- Via la ligne de commande :

```
$ terraform apply \  
  -var 'access_key=foo' \  
  -var 'secret_key=bar' \  
# ...
```




On joue !

- Via un fichier :
- Dans un fichier terraform.tfvars

```
access_key = "foo"  
secret_key = "bar"
```

- Les fichiers terraform.tfvars et *.auto.tfvars présents dans le répertoire courant sont automatiquement pris en compte.



On joue !

- Mais on peut aussi définir des fichiers .tfvars avec d'autres noms. Ils seront ignorés, sauf si on les spécifie via la ligne de commande.

```
$ terraform apply \  
  -var-file="secret.tfvars" \  
  -var-file="production.tfvars"
```



On joue !

- Via les variables d'environnement :
- Les variables de la forme TF_VARS_name sont reconnues et prises en compte (mais via les variables d'environnement on ne peut que définir des variables de type « chaine de caractère »).



On joue !

- De manière interactive :
- Les variables non renseignées par d'autres moyens (et sans valeur par défaut spécifiée) seront demandées par terraform lors de l'application.



On joue !

- On peut définir des variables de tout type reconnu par terraform (hors la limitation évoquée pour les variables d'environnement)
- Des listes : [« valeur 1 », « valeur 2 »]
- Des maps : { « clef » = « valeur », « clef » = « valeur » }
- Et bien sûr les simples chaines de caractère.



On joue !

- On peut aussi définir des valeurs de sortie

```
output "ip" {  
  value = "${aws_eip.ip.public_ip}"  
}
```

- Qui pourront être interrogées aisément

```
$ terraform output ip  
50.17.232.209
```



À vous !

- Objectif :
 - Déployer via Terraform une infrastructure contenant une VM à laquelle on peut se connecter via SSH.
- Moyens :
 - Un ordinateur, terraform, un compte AWS.
- Attendus :
 - « terraform apply » et hop, « ssh ubuntu@adresse_ip »

Oh non, encore ?



On joue !

- Oui, on va définir une infrastructure avec 2 VM, accessibles via SSH pour les deux et une accessible sur le port 8080.
- Sur ces deux VM, ansible va s'assurer que l'une aura SquashTM installé et l'autre une base de données.
- Le tout sera (idéalement) fonctionnel 😊

