

Généralités :

- se connecter en adminetu sous Debian
- récupérez votre dossier de travail du TP2, penser à le sauvegarder en fin de séance
- de préférence, utiliser VSCode avec l'extension YAML en ouvrant votre répertoire de travail
- on a besoin de pipenv pour travailler :

```
sudo apt update && sudo apt -y install python3-pip pipenv  
pipenv install && pipenv shell
```

1. GIT GUD

1.1. CONFIGURATION DE GIT

Si git n'est pas installé, installez-le à l'aide d'apt.

Avant de pouvoir utiliser git depuis votre VM ou, plus globalement, depuis n'importe quel poste de travail, il faut configurer, a minima, deux paramètres globaux, à savoir notre nom et votre adresse mail. Il s'agit ici d'indiquer à git l'identité avec laquelle vous signerez les commits.

Utilisez ces deux commandes pour réaliser cette configuration. Remplacer les valeurs entre chevrons par votre nom/prénom et votre adresse mail (il ne s'agit qu'une valeur déclarative, rien d'obligatoire)

```
git config --global user.name "<Prénom> <Nom>"  
git config --global user.email "<adresse mail>"
```

1.2. PREMIERS COMMITS

Testez la commande `git status` dans votre répertoire de travail : normalement vous aurez un message d'erreur car vous n'êtes pas dans un dépôt git. Il faut donc effectuer :

```
git init  
git status
```

Que voyez-vous ? En particulier, qu'est-il indiqué concernant les fichiers ? Sont-ils suivis par git ?

1.2.1. .GITIGNORE

Avant de commencer à ajouter des fichiers dans l'espace de staging, nous allons prendre quelques précautions !

Dans la liste des fichiers que git vous indique comme untracked, vous verrez, par exemple un dossier `.vagrant` (si vous avez déjà lancé la commande `vagrant`) qui est le dossier où Vagrant stocke des informations spécifiques aux maquettes que vous exécutez localement.

Hors, il y a certains fichiers ou dossiers, à l'image du dossier `.vagrant`, que l'on ne souhaite jamais ajouter à notre espace de staging. Par exemple, des fichiers temporaires, des fichiers contenant des identifiants et autres secrets, des fichiers de compilation, etc.

Afin d'éviter par erreur ces fichiers, il est possible de spécifier à `git` que certains éléments de l'arborescence sont bannis de l'index et ne pourront jamais être stagés. Cela se fait simplement en ajoutant un fichier nommé `.gitignore` (notez bien le point au début du fichier)

Dans ce fichier, on précise la liste des noms de fichier qui ne pourront jamais être ajoutés à l'espace de staging.

Créez un fichier `.gitignore` contenant les lignes suivantes :

```
.vagrant/  
.DS_Store  
.*
```

Les deux dernières lignes du fichier sont plutôt à usage des utilisateurs de MacOS (*quelle horreur*), celui-ci ayant tendance à ajouter des fichiers de ce type dans les dossiers.

NB : comme vous pouvez le voir, on peut utiliser les jokers habituels pour définir des noms de fichiers. Ici, la 3e ligne indique tous les fichiers commençant par « `._` »

Maintenant, refaites un `git status`. Le dossier `.vagrant` apparaît-il toujours ?

1.2.2. AJOUT DES FICHIERS À L'INDEX ET PREMIER COMMIT

Pour « stager » tous les fichiers, vous pouvez utiliser :

```
git add -A .
```

Par la suite, pour ajouter les fichiers dont vous avez besoin ou que vous avez modifié vous pourrez utiliser la commande :

```
git add <nom_du_fichier>
```

Maintenant, relancez la commande `git status` et vérifiez bien que les fichiers ajoutés sont bien prêts à être commit ! Enfin, on peut effectuer un commit avec la commande suivante :

```
git commit -m "Premier commit du dépôt"
```

L'option `-m` vous permet d'indiquer un message descriptif court pour indiquer la raison du commit : correction de bug, modification d'un fichier, ajout d'une fonctionnalité, etc. C'est très important de garder un message explicite pour le suivi de votre code dans le temps !

Vous pouvez consulter l'historique des commits avec la commande `git log` (q pour quitter par la suite)

1.3. FAIRE ÉVOLUER SON PROJET AVEC GIT

1.3.1. MODIFIER UN FICHIER

Ouvrez votre fichier `Vagrantfile` et ajoutez un commentaire en début de fichier, par exemple :

```
# Vagrantfile pour le TP R405. Configure les VM Load-Balancer, DNS et Web pour la  
génération par Vagrant.
```

Refaites un `git status` : que se passe-t-il ? En vous appuyant sur ce que vous avez vu avant, ajoutez les modifications dans votre espace `staging` et faites un `commit` des changements. Vérifiez que votre `commit` est bien présent dans l'historique !

1.3.2. VISUALISER L'ÉVOLUTION DU PROJET

L'un des intérêts de `git` est de pouvoir voir, l'évolution du code par rapport à un autre `commit` et de pouvoir revenir dans le temps. Utilisez la commande suivante :

```
git diff HEAD~1
```

Que vous montre cette commande ?

`HEAD` est un pointeur vers le dernier `commit` en date. L'ajout de « 1 » permet d'indiquer le `commit` -1 avant `HEAD`. On peut ainsi référencer les `commits` antérieurs à l'endroit où nous nous trouvons dans l'arbre d'historique du projet. Si vous regardez attentivement la sortie de la commande `git log`, vous verrez qu'il vous est indiqué sur quel `commit` pointe `HEAD`. Sinon, on peut directement référencer le `commit` que l'on souhaite voir en spécifiant son identifiant `SHA1` dans la commande ci-dessus.

1.3.3. REVENIR EN ARRIÈRE

Maintenant utilisons `git` pour annuler notre dernière modification dans le fichier `Vagrantfile`. Autrement dit, annulons le dernier `commit` !

Pour cela, nous pouvons utiliser la commande suivante : `git revert HEAD`

Cette commande indique à `git` de créer un `commit` faisant exactement l'opposé du `commit` indiqué en argument, ici `HEAD`, autrement dit votre dernier `commit`. Comme nous créons un nouveau `commit`, `git` va nous demander de spécifier le message de `commit`, pour cela une fenêtre `nano` s'est ouverte avec un message par défaut qui vous est proposé.

Ne modifiez pas le message mais sortez directement en enregistrant et quittant `nano` (`CTRL+X`) ou `vim` (`escape`, `:wq`, entrée).

Refaites un `git log`, que constatez-vous ?

1.4. TRAVAILLER AVEC UN DÉPÔT DISTANT

1.4.1. INSCRIPTION SUR GITHUB


Si vous n'avez pas encore de compte, inscrivez-vous sur [GitHub](https://github.com), avec votre adresse mail Unicaen (ou votre adresse perso si vous souhaitez garder votre compte dans le futur !)

Vous devez ensuite créer un nouveau dépôt (repository en anglais). Indiquez un nom de dépôt explicite, vous pouvez le garder en privé et c'est à peu près tout !

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)


Required fields are marked with an asterisk (*).


Owner *  brisacier / **Repository name ***

✔ vega_myssil is available.

Great repository names are short and memorable. Need inspiration? How about **cautious-octo-barnacle** ?

Description (optional)

☐  **Public**
Anyone on the internet can see this repository. You choose who can commit.

☒  **Private**
You choose who can see and commit to this repository.

Initialize this repository with:


☐ Add a README file
This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

 You are creating a private repository in your personal account.

[Create repository](#)

Fig. 1. – Exemple de création de dépôt (ça ne laisse pas indifférent)

Pour l'authentification, GitHub a déprécié la connexion par mot de passe simple et préconise maintenant l'utilisation de token de sécurité. Voir la documentation suivante, section « legacy token » : <https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens#creating-a-personal-access-token-classic>

New personal access token (classic)

Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Note

Clé d'accès temporaire à la vega-myssil

What's this token for?

Expiration *

30 days ▾ The token will expire on Fri, Mar 7 2025

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events

Fig. 2. – Exemple de création de token

Il faut garder précieusement votre token, il vous servira pour l'authentification sur GitHub. Si vous le perdez, il faudra en régénérer un !

1.4.2. SYNCHRONISER SON DÉPÔT LOCAL AVEC LE DISTANT (GIT : ÇA POUSSE !)

Une fois que vous avez fait cela, votre dépôt sur GitHub est vide. Il faut maintenant configurer votre dépôt local pour lui indiquer de se synchroniser avec GitHub. Pour cela, on utilise la commande `git remote` :

```
git remote add origin <URL de votre dépôt GitHub>
```

L'URL de votre dépôt est de la forme : `https://github.com/nom_utilisateur_git/nom_du_depot.git`

Avec cette commande, nous indiquons à votre git local un serveur distant que nous appelons `origin`. Ce nom est arbitraire et on pourrait l'appeler différemment. Mais il est dans les pratiques courantes d'appeler le serveur distant `origin`.

Une fois le serveur `origin` déclaré, il nous faut pousser le code vers GitHub :

```
git push -u origin master
```

Le nom d'utilisateur est votre login GitHub et le mot de passe est votre token généré précédemment ! Avec cette commande vous liez votre branche `master` local à une nouvelle branche `master` sur le serveur `origin`.

Allez voir l'interface web de votre dépôt GitHub : vous devriez voir vos différents fichiers et l'historique de vos commits !

Maintenant, à chaque fois que vous réaliserez un `commit` local, vous devrez utiliser la commande `git push` (sans argument supplémentaire) pour télécharger ce `commit` vers le serveur GitHub. L'option `-u`, qui est le raccourci de `--set-upstream`, permet d'indiquer la branche distante par défaut. Ainsi, vous

n'aurez pas besoin de préciser à chaque `git push` la destination. (on pourrait vouloir synchroniser avec différents serveurs)

1.4.3. RÉCUPÉRER UN PROJET DÉJÀ EXISTANT

Maintenant que vous savez synchroniser votre dépôt local avec un dépôt distant, on va voir le cas inverse. Imaginez que vous arrivez en salle de TP et que vous voulez récupérer rapidement votre dépôt GitHub en local sans passer par l'interface Web !

Pour joindre un projet, il faut d'abord cloner le dépôt distant localement. Cette opération permet de récupérer l'ensemble d'un dépôt existant vers un dossier local.

Allez sur le Bureau (ou un autre dossier que votre dossier de travail) puis clonez votre dépôt GitHub :

```
cd /home/adminetu/Desktop
git clone https://github.com/nom_user/mon_depot.git
cd mon_depot
```

NB : `git clone` crée un dossier du nom du projet où télécharger localement le dépôt. Encore une fois, le mot de passe est votre token généré précédemment.

Puis exécutez la commande `git log`. Que voyez-vous ? Avez-vous récupéré l'ensemble de l'historique du projet ?

1.4.4. COMPLÉMENTS

Si vous êtes « en retard » sur votre dépôt local par rapport au dépôt distant, on peut utiliser la commande `git pull` pour télécharger la version distante (en gros c'est l'inverse de `git push`). C'est surtout utile lorsqu'on travaille à plusieurs sur le même dépôt.

Il est aussi possible de créer des « branches » pour avoir différentes versions du code en concurrence, et de finalement fusionner ces branches. Ce ne sera pas vu dans ce cours pour aller au plus simple !

Maintenant, vous savez créer un dépôt local, lancer des `commit` et synchroniser votre dépôt local avec un dépôt distant GitHub. Donc pour la suite des TP (à partir du TP4), pourquoi ne pas créer un dépôt GitHub pour stocker tout votre code Vagrant et Ansible ?

2. PRIORITÉ DES VARIABLES ANSIBLE (PRECEDENCE)

Documentation utile :

- https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_variables.html#understanding-variable-precedence
- https://docs.ansible.com/ansible/latest/collections/ansible/builtin/debug_module.html

Ici, je pars du principe que vous avez votre environnement de TP opérationnel : Vagrant installé, et environnement virtuel Python avec Ansible installé. Si ce n'est pas le cas, il suffit de relancer quelques commandes ! Assurez-vous que vos VM sont bien accessibles en SSH avec Vagrant, et que le fichier d'inventaire est à jour pour Ansible.

2.1. MODULE DEBUG

Dans le fichier `inventory/group_vars/all.yml`, ajoutez une variable appelée `vartest` dont le contenu est « variable définie pour tous les hôtes ». Exécutez ensuite la commande suivante :

```
ansible -m ansible.builtin.debug -a 'msg="Contenu de la variable vartest:
{{ vartest }}"' all
```

Qu'est-ce que cette commande a effectué ?

2.2. VARIABLES DE GROUPE

À l'aide de la documentation ci-dessus et de ce qui a été vu en CM, créez un fichier de variables pour le groupe web et un autre pour le groupe lb. Dans ces fichiers, définissez de nouveau la variable vartest avec respectivement les valeurs « variable définie pour les hôtes du groupe web » et « variable définie pour les hôtes du groupe lb ».

Exécutez de nouveau la commande ansible-debug de la partie précédente. Que voyez-vous ? Pourquoi ?

2.3. VARIABLES D'HÔTES

Maintenant, créez un fichier de variables pour l'hôte web-1. Dedans, définissez à nouveau la variable vartest avec la valeur « variable définie pour l'hôte web-1 uniquement ».

Exécutez de nouveau la commande ansible-debug précédente. Que voyez-vous ? Pourquoi ?

Pour conclure, comment fonctionne la notion de Precedence sur Ansible ?