USINE LOGICIELLE

setup vagrant

* lien de téléchargement: https://www.vagrantup.com/downloads
* créer un dossier pour accueillir un fichier de confguration vagrant
* contenu "classique" d'un fichier de configuration **Vagrantfile**

```
Vagrant.configure(2) do |config|

  int= "interface réseau"
  range = "adresse ip"
  cidr = "24"

  [
    ["nom de la vm dans virtualbox", "#{range}", "RAM", 'nombre de coeurs/processeurs', "nom de l'image vagrant"],
  ].each do |vmname,ip,mem,cpu,os|
    config.vm.define "#{vmname}" do |machine|

      machine.vm.provider "virtualbox" do |v|
        v.memory = "#{mem}"
        v.cpus = "#{cpu}"
        v.name = "#{vmname}"
        v.customize ["modifyvm", :id, "--ioapic", "on"]
      end
      machine.vm.box = "#{os}"
      machine.vm.hostname = "#{vmname}"
      machine.vm.network "public_network",bridge: "#{int}",
        ip: "#{ip}",
        netmask: "#{cidr}"
      machine.ssh.insert_key = false
    end
  end
end
```

* la variable **int** contient l'interface réseau reliant la machine au web
   * Windows: gestionnaire de périphs -> cartes réseay

* la variable **range** contient une ip disponible du sous réseau local (!= ip de l'hôte !)
* la variable **cidr** contient une valeur décrivant le masque de sous réseau (16 ou 24)

* paramètres principaux de la vm:
    * le nom et le hostname de la vm: **gitlab.myusine.fr**

    * la RAM allouée: 6144
    * le nombre de proc / coeurs : 2+
    * nom de la box vagrant : **hashicorp/bionic64**

* boxes publiques vagrant: https://app.vagrantup.com/boxes/search

* lancer la vm: se déplacer dans le dossier contenant le vagrantfile
   * dans powershell: `cd c:\...`
   * `vagrant up`

installation de gitlab sur la vm vagrant

* se connecter sur la vm
    * dans le dossier contenant le vagrant file: `vagrant ssh`
    * stopper la vm : `vagrant halt`
    * détruire la vm : `vagrant destroy`

* suivre l'installation via apt

* commandes de base:
    * ls [path]: afficher le contenu d'un dossier (option -al pour accès aux métadonnées)
    * cd pour changer de répertoire de travail
=
* édition du fichier de configuration de gitlab: `sudo nano /etc/gitlab/gitlab.rb`

* édition du dns local: c:\Windows\System32\drivers\etc\hosts
    * [ip de la vm] gitlab.myusine.fr

* dans la vm : `sudo gitlab-ctl status`

création d'un projet gitlab

* sur gitlab.myusine.fr, choisir et confirmer un mot de passe root "roottoor"

* se connecter en root / roottoor

* créer un projet "from blank"

* créer une paire de clés privée / publique

connection dépôt local -> gitlab

* créer un dossier de travail

* git init

* créer un fichier README.md

* git add . && git commit -m "first commit"

* git remote add origin git@gitlab.myusine.fr:root/myusine.git

* vérifier avec git remote -v

* créer un fichier **config** dans le dossier **.ssh** du dossier utilisateur

* ajouter le code qui force l'utilisation de la clé privée

```
Host gitlab.myusine.fr
 Hostname gitlab.myusine.fr
 IdentityFile "/c/Users/"nom utilisateur"/.ssh/gitlab"
```

* git push origin master

installation et configuration gitlab-runner

* installation de docker pour pouvoir lancer des conteneur à travers gitlab-runner

* les commandes docker s'exécutent en root, ou avec un utilisateur membre du group docker créé à l'installation
    * on va ajouter l'utilisateur courant au groupe docker: `sudo usermod -aG docker vagrant`
    * l'ajout est pris en compte après reconnexion (exit + vagrant ssh)
    * confirmation avec la commande `id`

* installation de gitlab-runner

* enregistrement d'un runner spécifique au projet gitlab via l'url de gitlab et un token

* utilisation de l'exécuteur **docker** : les agents seront des conteneurs lancés sur la machine hébergeant gitlab-runner

* configuration du runner dans `/etc/gitlab-runner/config.toml`

* ajout d'une résolution dns dans le conteneur
    * enregistrer dans l'éditeur nano : **ctrl + o et Entrée**
    * fermer l'éditeur nano : **ctrl + x**

création du fichier **.gitlab-ci.yml**

```
build:
  script:
    - echo "Build"


test:
  script:
    - echo "Test"


deploy:
  script:
    - echo "Deploy"
```

* ajout, commit et push du nouveau fichier sur gitlab

quelques manipulations de yaml: bloc litéral et replié, alias et clés désactivées

```
*# les clés de premier niveau commençant*
*# par "." ne sont pas exécutées par gitlab*
.where_am_i: &where pwd


build:
  script:
    - |
      echo "Build"
      pwd
      ls -al

test:
  script:
    - >
      find /builds/root/myusine2/
      -type f
      -name "README*"
      -not -path "./lib/*"
    - *where
```

clés fondamentales des jobs

```
# image par défaut des jobs du pipeline
# surcharge l'image par défaut du runner
image: ubuntu:bionic

build:
  # image pour le job
  # surcharge les précédentes images
  image: python:latest
  script:
    # affichage d'une variable d'environnement
     # prédéfinies dans gitlab
     - echo "$CI_PROJECT_DIR"
    - python3 -c "print('Build')"

test:
  # préparation à l'exécution
  before_script:
    - |
      apt-get update
      apt-get install -y python3 python3-pip
  # exécution
  script:
    - python3 -c "print('Test')"
```

synchronicité des jobs

* dans la suite on va utiliser l'image **python:rc-slim-buster**

* mise en place des stages

* à l'intérieur d'un stage, exécution parallèle

* le runner lance les conteneurs par défaut dans un seul thread
    * cela reste séquentiel, mais on ne connait pas l'ordre exécution

* pour mettre en place un vrai prallélisme, il faut configurer l'option **concurrent** dans la conf de gitlab-runner: `sudo nano /etc/gitlab-runner/config.toml`

```
stages:
  - builds
  - tests
```

```
build:
  # cahque job est associé
  # à un stage
  stage: builds
  script:
    - python3 -c "print('Build')"

test:
  stage: tests
  script:
    - python3 -c "print('Test')"

test2:
  stage: tests
  script:
    - python3 -c "print('Test2')"
```

* utilisation des clé **needs:**

```
build:
  stage: builds
  script:
    - python3 -c "print('Build')"

build2:
  stage: builds
  script:
    - sleep
    - python3 -c "print('Build2')"

test:
  stage: tes
  # liste sous forme de JSON
  # ce job démarre dès que "build" a terminé
  needs: [build]
  script:
    - python3 -c "print('Test')"
```

* pipelines parent / enfants

```
```

```
stages:
  - triggers

trigger1:
  stage: triggers
  trigger:
    include: module1/.gitlab-ci.yml

trigger2:
  stage: triggers
  trigger:
    include: module2/.gitlab-ci.yml
```

conditionnalité : when/only

* cf sources

conditionnalité : rules

* cf sources

merge requests dans gitlab

* similaire aux pull requests dans github, fusionne deux branches sur le dépôt de référence
* le code fusionné est d'abord gardé dans une branche de fusion faisant l'objet d'un pipeline
* à l'issue du pipeline de fusion (si autorisé) des mails sont envoyés à des réviseurs ("Reviewers")
* un consensus doit s'opérer dans l'équipe après analyse pour valider la merge request
* on peut alors effectivement fusionner les branches

création d'un environnement virtuel pour un projet python

1. installation de python sur windows
   * python.org -> downloads -> windows -> windows installer 64 bits
   * https://www.python.org/ftp/python/3.9.4/python-3.9.4-amd64.exe

2. création de l'environnement virtuel dans le dossier projet
   * `python -m venv "destination"` avec destination = venv
   * cela injecte dans le dossier venv un dossier Scripts avec un binaire python et pip
   * cela injecte un dossier Lib dans lequel les dépendances projets seront téléchargées

3. activation de l'environnement virtuel dans le dossier projet
   * `.\venv\Scripts\(a|A)ctivate.(bat|ps1)`

4. désactivation
    * `deactivate`

installer les dépendances projet dans l'environnement virtuel

* le projet a comme dépendance le paquet bottle
* installation dans l'environnement virtuel: `pip install bottle`
* le dépôt officiel de paquet python se trouve sur pypi.org
* afficher les dépendances installées: `pip freeze`

écriture d'un premier job lié au projet local:

* fichier gitlab de base

```
image: python:rc-slim-buster

workflow:
  rules:
    # OU logique: déclenchement sur un push ou une merge request
     - if: $CI_PIPELINE_SOURCE == "push"
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"

create_env:
  script:
    - echo "create venv and put it in cache"
```

1. générer les dépendances projets dans un fichier requirements.txt
2. mettre à jour le dépôt gitlab avec les nouvelles sources
3. créer un stage "init" et y associer le job create_env

4. dans le job create_env: créer et activer l'environnement virtuel dans un dossier venv
    * sous linux, utiliser python3 et pip3
5. dans le job create_env: installer les dépendances à partir du fichier requirements.txt
6. utiliser la clé cache pour placer le dossier venv en cache
7. tester le chargement du cache dans un job de test

* exemple de job
```
stages:
  - init
  - test

create_env:
  stage: init
```

```
  before_script:
    - |
      apt-get update
      apt-get install -y python3-venv
  script:
    - |
      python3 -m venv venv
      source venv/bin/activate
      pip3 install -r requirements.txt
  # création du cache en mode push
  cache:
    key: venv
    untracked: true
    paths:
      - venv/
    policy: push

test:
  stage: test
  script:
    - test -e venv
  # chargement du cache en mode pull
  # pas besoin de renseigner paths
  cache:
    key: venv
    untracked: true
    policy: pull
```

* BONUS: n'exécuter le job que si modification sur requirements.txt
* BONUS: placer une procédure manuelle de regénération du cache

```
rules:
  - changes:
      - requirements.txt
  - if: $TRIGGER_CACHE == "on"
```

tests unitaires automatisés

* on utilise le module unittest
* on déclenche un test unitaire en ligne de commande avec `python -m unittest`
* on sélectionne un ou des tests en particulier grâce au chemin python
    * `python -m unittest bank.tests.test_client`
* on peut lancer une découverte de tests avec `python -m unittest discover`

* on peur lancer une suite de tests: ex `python -m unittest run_tests.py`

```
units:
  stage: tests
  before_script:
    - source venv/bin/activate
  script:
    - python3 -m unittest run_tests.py -v
  cache:
    key: venv
    untracked: true
    policy: pull
```

remontée du rapport dans Gitlab

* Gitlab peut intégrer dans son interface le rapport de test unitaire s'il est fourni en tant qu'artefact au format XML-Junit
* unittest ne sait pas exporter le rapport en xml nativement
* on doit installer un package python externe: `pip install unittest-xml-reporting`
* on va séparer les dépendances du projet des dépendances CI/CD en exportant ces dernières dans un fichier spécifique **requirements-ci.txt**
* TIP: `pip freeze | ?{$_ -notmatch "bottle"}` > requirements-ci.txt
* on fait en sorte d'exporter un rapport xml dans un fichier (au niveau de la classe XMLTestRunner du module coverage_tests.py)
* on écrit les modifications du job units qui font
   1. remonter les rapports xml dans l'interface gitlab
   2. qui active le job create_env sur une modification de requirements ou requirements-ci.txt

```
units:
...
script:
  # génération des xml dans le dossier reports
  - python3 coverage_tests.py
artifacts:
  reports:
    junit: "reports/TEST-*.xml"
```

couverture de code

* on va installer le package coverage via pip dans l'environnement virtuel
* on lance les tests avec la commande `coverage run coverage_tests.py`
* on affiche le rapport de couverture de code : `coverage report -m`

* on constate que la couverture concerne des sources inutiles (__init__, les tests, les dépendances)
* on va recentrer la couverture sur le code utile les classes métiers
    * https://coverage.readthedocs.io/en/coverage-5.5/cmd.html pour chercher les options d'inclusion et d'exclusions de fichiers pris en compte
   * `coverage run --include="bank/*.py" --omit="**/__init__.py,**/tests/*.py" coverage_tests.py`

* on adapte le job units dans gitlab
```
script:
  # génération des xml dans le dossier reports
  - >
    coverage run
    --include="bank/*.py"
    --omit="**/__init__.py,**/tests/*.py"
    coverage_tests.py
  - coverage report -m
```
* on regénère le requirements-ci.txt

* on peut faire remonter le résultat de couverture dans le tableau des jobs dans gitlab

- * se rendre des la section **pipeline généraux** du sous menu **CI/CD** des réglages du projets
- * ajouter `^TOTAL.+?(\d+\%)$` dans l'input **coverage parsing**

Evolution de la couverture de code dans les merge requests

* on cherche à comparer la couverture de code entre une nouvelle branche de fonctionnalité et la branche master

1. on procède à un cycle de TDD
   * modification / ajout de tests
   * modification / ajout de fonctionnalité
   * pousser la branche pour exécuter l'intégration continue

2. pour accéder au widget de couverture de code dans les merge requests, il faut fournir un artefact au format cobertura
   * génération du rapport de couverture XML: `coverage xml -o reports/coverage.xml`
   * ajouter une clé **artifacts:reports:cobertura**

3. on lance une Merge Request
   * menu **Merge Request** du projet
   * détection du push sur la beanche de fonctionalité **Create Merge Request**
   * configuration de la MR (titre, description, responsable, réviseurs)
   * submit Merge Request
   * un **pipeline de merge request** est exécuté

   * l'évolution de la couverture apparaît dans l'interface
   * les lignes couvertes et non couvertes du code testé appraissent dans l'inglet **changes** qui analyse les différences entre commits

4. si la MR est acceptée
   * on fusionne: bouton **Merge**
   * on met à jour le dépôt local: pull sur master et suppression de la branche de fonctionnalité

tests End to End (E2E) avec Selenium

* setup avec firefox local
   1. rajouter le code suivant dans le fichier e2e_firefox.py
```
import unittest
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

class PythonOrgSearch(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def test_search_in_python_org(self):
        driver = self.driver
        driver.get("http://www.python.org")
        self.assertIn("Python", driver.title)
        elem = driver.find_element_by_name("q")
        elem.send_keys("pycon")
        elem.send_keys(Keys.RETURN)
        assert "No results found." not in driver.page_source



    def tearDown(self):
        self.driver.close()

if __name__ == "__main__":
    unittest.main()
```
2. `pip install selenium` dans l'environnement virtuel
3. télécharger geckodriver pour windows 64 bit: https://github.com/mozilla/geckodriver/releases
4. placer geckodriver.exe dans le PATH Windows
5. `py e2e_firefox.py`

* setup serveur Sélenium en local
   1. commenter les lignes 11, 13, et 14 du code e2e_remote.py
   2. télécharger un serveur selenium en local, le lancer via `java -jar selenium-....jar`

   3. remplacer la ligne 14 par `command_executor="http://localhost:4444/wd/hub",`
   4. installer le package selenium dans l'environnement virtuel `pip install selenium`
   5. télécharger le geckodriver windows 64 bit à jour https://github.com/mozilla/geckodriver/releases
   6. placer le **geckodriver.exe** dans le PATH windows
   7. `py e2e_remote.py

* automatiser le test E2E
   1. dans un contexte conteneurisé, pas d'interface graphiqe disponible => navigateur en mode "HEADLESS"
   2. soit on dispose d'un serveur selenium d'entreprise et on configure le client avec
   3. soit on peut se donner un service réseau sélenium qu'on va adjoindre à l'agent du job e2e:
   ```
   services:
     - name: selenium/standalone-firefox:latest
       alias: gitlab_selenium_server
   ````
   4. se connecter avec **http://gitlab_selenium_server:4444/wd/hub** dans le client selenium
   5. ajouter la gestion du geckodriver sur le conteneur dans un before_script

```
e2e:
  stage: tests
  services:
    - name: selenium/standalone-firefox:latest
      alias: gitlab_selenium_server
  before_script:
    - cp geckodriver /usr/local/bin
    - chmod +x /usr/local/bin/geckodriver
    - source venv/bin/activate
  script:
    - python3 e2e_remote.py
  cache:
    key: venv
    untracked: true
    policy: pull
```

troubleshooting cache et multithreading

* le cache gitlab est enregistré dans un volume docker qui dépend du runner, du projet gitlab et du numéro de thread du job qui le génère: `docker volume list`
* si plusieurs jobs exécutés en parallèle dépendent du même cache, seul celui exécuté par le même thread recevra le cache
* solution: la clé **parrallel** permet de génrer un cache dans plusieurs threads simultanément

Qualité: linter python pylint

* `pip install pylint`
* `pip freeze | ?{$_ -notmatch "bottle"}` > requirements-ci.txt

* `pylint --generate-rcfile > .pylintrc`pour générer la conf pylint (attention encodage !)
* utiliser les règles **enable** et **disable** pour affiner les règles pertinentes d'analyses

* aplliquer pylint à l'ensemble des sources d'un projet, hors environnement virtuel :
    * sous linux: `find . -type f -name "*.py" ! -path "./venv/**/*.py" | xargs pylint`

* job d'analyse syntaxique

```
syntax:
  stage: syntax
  before_script:
    - *activate
  script:
    # xargs transorme le flux de sortie de find en paramètres d'entrée de pylint
     - find . -type f -name "*.py" ! -path "./venv/**/*.py" | xargs pylint -E
  cache: *pull_cache
```

remontée du score qualité dans un badge gitlab

* on peut créer des badges dans gitlab: settings -> General -> Badges
    * il faut renseigner le nom, une url de lien cliquable et l'url du badge

* pour créer un badge custom, il faut:
    * un package de génération de badge: `pip install anybadge`
    * `anybadge --file="filename.svg" --label="label" --value="value" val1=red val2=orange ...`

* la valeur du badge doit être le score qualité, remontée depuis la sortie de pylint
   * `find . -type f -name "*.py" ! -path "./venv/**/*.py" | xargs pylint`
   * on duplique la sortie de pylint sur la console et dans un fichier pylint.txt avec la commande **tee**
   * on exécute la commande **sed** en mode substitution pour retourner le score de pylint
   * on remplace la dernière ligne par le score
   * `sed -n 's/^Your code has been rated at \([-0-9.]*\)\/.*/\1/p' pylint.txt`
   * on place se résultat dans une variable: score=$(sed ...)
   * on complète le badge: `anybadge --file=pylint.svg --label=pylint --value=$score 2=red 4=orange 8=yellow 10=green`

* le job qui en découle
    * le job style ne doit pas être bloquant, on l'exécute en même temps que les tests
    * le résultat de pylint est considéré comme un code de retour non nul ==> échec
    * on utilisera l'option -**-exit-zero** de pylint pour forcer le code de retour à 0

```
style:
  stage: tests
  before_script:
    - *activate
  script:
    - >
      find . -type f -name "*.py" ! -path "./venv/**/*.py" |
      xargs pylint --exit-zero | tee pylint.txt
    - score=$(sed -n 's/^Your code has been rated at \([-0-9.]*\)\/.*/\1/p' pylint.txt)
    - anybadge --file=pylint.svg --label=pylint --value=$score 2=red 4=orange 8=yellow 10=green
  artifacts:
    expire_in: 1 hour
    paths:
      - pylint.svg
```

* on créé le badge dans gitlab:
    * nom pylint
    * URL: `https://gitlab.myusine.fr/%{project_path}/-/jobs/artifacts/%{default_branch}/raw/pylint.svg?job=style`
    * lien: `https://gitlab.myusine.fr/%{project_path}/-/commit/%{commit_sha}`

évolution de la qualité dans les Merge Requests

1. faire remontér le rapport qualité au format **codequality** dans gitlab
    * `pip install pylint-gitlab`
    * pip freeze ...
    * rajouter une commande pylint avec format de sortie json
    * `find . -type f -name "*.py" -not -path "./venv/**/*.py" | xargs pylint --output-format=pylint_gitlab.GitlabCodeClimateReporter --exit-zero > codeclimate.json`
    * prélever l'artefact codeclimate.json en tant que rapport codequality

```
style:
    ...

 - >
    find . -type f -name "*.py" -not -path "./venv/**/*.py" |
    xargs pylint --output-format=pylint_gitlab.GitlabCodeClimateReporter --exit-zero
    > codeclimate.json
artifacts:
  expire_in: 1 hour
  paths:
```

```
    - pylint.svg
    - codeclimate.json
  reports:
    codequality: codeclimate.json
```

2. améliorer le code dans une branche et pousser et fusionner
    * correction des trailing-newlines

3. créer une Merge Request dans gitlab
    * le rapport d'évolution de la qualité apparaît après le pipeline de MR

serveur SonarQube

1. lancer un conteneur sonarqube dans la vm
    * `docker container run --name sonar --restart unless-stopped -d -p 9000:9000 sonarqube:lts`
    * `docker ps` pour confirmer le lancement du conteneur

2. accès au serveur sonarqube en ligne via ip_de_la_vm:9000

* création du projet (nom, token, language, OS)
* création d'un profile qualité python à partir du profil par défaut
* création d'une gate (seuil de valisation) d'après le % de lignes dupliquées

3. écriture du job gitlab

* utilisation de l'image dédiée sonar-scanner
* ajout d'options **sonar.inclusions** et **sonar.exclusions** pour affiner le scope de l'analyse
* ajout du rapport de coverage au format cobertura XML :
    * on a besoin du rapport généré dans le job **units**, comme artefact prélevé en plus du rapport cobertura
    * option **sonar.python.coverage.reportPaths**
* on peut également ajouter le rapport de test unitaire pour déterminer une gate supplémentaire
* on doit cacher les informations critiques (mot de passe) dans des variables en dehors du fichier de configuration (Settings -> CI/CD -> Variables)

```
sonar:
  stage: quality
  # image docker dont on modifie la clause ENTRYPOINT
  image:
    name: sonarsource/sonar-scanner-cli:latest
    entrypoint: [ "" ]
  script:
    - >
      sonar-scanner
      -Dsonar.projectKey=myusine
```

```
      -Dsonar.sources=.
      -Dsonar.inclusions=**/*.py
      -Dsonar.exclusions=**/*.css
      -Dsonar.python.coverage.reportPaths=reports/coverage.xml
      -Dsonar.coverage.exclusions=**/__init__.py,**/tests/*.py,*.py
      -Dsonar.host.url=http://172.17.0.1:9000
      -Dsonar.login=$SONAR_TOKEN
  dependencies: [units]
```

déploiement avec Ansible

* SETUP de la cible Ansible
    1. Création d'un compte utilisateur **ansible** sur la vm, avec la commande `adduser`
        * `sudo adduser ansible` et renseigner mdp
    2. configurer le serveur ssh de la vm pour accepter les connections par clés publiques

        - https://www.linuxtricks.fr/wiki/ssh-installer-et-configurer-un-serveur-ssh ici il y a des éléments de réponses
        - * `sudo nano /etc/ssh/sshd_config`
        - * décommenter la ligne `#PubKeyAuthentication yes`
        - * `sudo service sshd reload`

    3. connecté en tant que ansible (`su`), générer une paire de clé publique/ privée, nommée ansible, à placer dans le dossier .ssh, sans passphrase
        * `su ansible -`
        * `ssh-keygen` chemin: /home/ansible/.ssh/ansible et pas de passphrase
    4. créer un fichier **authorized_keys**, dans le dossier .ssh de l'utilisateur ansible, avec les droits 600 et y copier le contenu de la clé publique (chmod)
        * **authorized_keys** stocke les clés publiques d'utilisateurs distant se connectant au compte anible de la vm avec une clé privée correspondant
        * dans le dossier .ssh: `mv ansible.pub authorized_keys` pour renommer la clé publique
        * `chmod 600 authorized_keys`
    5. se reconnecter en vagrant et rapatrier la clé privée dans le compte vagrant
        * `exit` pour revenir sur la session vagrant
        * `sudo mv /home/ansible/.ssh/ansible .`

* test de la connection ssh depuis un conteneur
    1. on va copier la clé privée, et les fichiers ansible du projet dans un dossier ansible_volume
        * `mkdir ansible_volume` fichiers: ansible, ansible.cfg, inventory, bootstrap.yml
        * `mv ansible ansible_volume`
        * `touch ansible.cfg inventory bootstrap.yml`
        * `mv ansible.cfg inventory bootstrap.yml ansible_volume`
        * copier coller de windows dans les fichiers édités avec nano
    2. on va lancer un conteneur ansible avec un volume pour tester la connexion et ansible
        * `docker container run -d --name ansible --restart unless-stopped -v /home/vagrant/ansible_volume:/root ansible/ansible:default`

    * vérifier que le conteneur tourne avec `docker ps` et le champ STATUS à "Up xxx"
    * se connecter sur le conteneur en exécutant un shell dessus: `docker exec -it ansible /bin/bash` (Ctrl +p+q pour sortir du conteneur)
    * se rendre dans /root pour trouver les fichiers du volume
    * changement de propriétaire dans le dossier /root: `chown -R root:root /root`
    * le dossier contenant la clé privée doit avoir des droits en 700: `chmod 700 /root`
    * la clé privée doit avoir des droits en 400: `chmod 400 /root/ansible`
    * test de connexion : `ssh -i ansible ansible@172.17.0.1`
      * -i pour ajouter la clé privée
      * 172.17.0.1 est l'adresse de la vm depuis le sous réseau docker

* test des commandes ansible
  * revenir sur le conteneur avec `exit`
  * il faut installer ansible sur le conteneur, via pip3: `pip3 install ansible`
  * commande ansible pour tester la connection:
    * `ansible -i inventory -u ansible --private-key ansible -m ping staging`
  * les paramètres déjà configurés dans **ansible.cfg** permettent de retirer des options:
    * `ansible -m ping staging`
  * recolter des informations sur la cible (config os, matérielle, réseau...)
    * ansible -m setup staging`

* lancement d'un playbook ansible
  * `ansible-playbook bootstrap.yml`
* troubleshooting `fatal: [staging]: FAILED! => {"msg": "Missing sudo password"}`
  * l'utilisateur distant doit avoir accès au sudo: `sudo usermod -aG sudo ansible`
* troubleshooting du mot de passe demandé
  * on créé un fichier /etc/sudoers.d/ansible dans lequel on configure le sudo avec l'option NOPASSWD
  * en manuel, on utilise l'option **--ask-become-pass** pour renseigner dans un prompt le mot de passe dès le lancement de la commande
  * en automatique on utilise l'option **ansible_become_pass=[mdp]** en tant que variable d'environnement de la commande

* écriture du job dans gitlab ci
  1. écrire le contenu de la clé privéd de connection à staging dans une variable d'environnement cachée
  2. insérer le contenu de la variable dans un fichier sur le conteneur du job
     * `echo "$STAGING_PKEY" > ansible
  3. changer les droits de la clé privée (400) et du répertoire contenant (700)
  4. utiliser l'image **ansible/ansible:default**
  5. installer ansible sur l'image: `pip3 install ansible`
  6. lancer la commande `ansible-playbook playbook.yml`
  7. transmettre le mot de passe sudo de l'utilisateur ansible de la cible avec la variable d'environnement affectée par une valeur en variable cachée:
     * `ansible-playbook -e "ansible_become_pass=$STAGING_PSSWD" playbook.yml
  8. on utilise la clé **environment:** pour donner accès à la version déployée dans l'interface Operations -> Environments

BONUS: registre docker

* customiser ses images de conteneurs avec des fichiers Dockerfile pour limiter les installations dans les jobs
* configurer le container registry de gitlab pour charger ces images bcp plus rapidement