Exemple de soumission d'activité

ÉTS - LOG430 - Architecture logicielle - Hiver 2026

Étudiant(e) : Adel Belhadj

Questions

(Il est obligatoire d'ajouter du code, des captures d'écran ou des sorties de terminal pour illustrer chacune de vos réponses.)

1. Si l'un des tests échoue à cause d'un bug, comment pytest signale-t-il l'erreur et aide-t-il à la localiser ? Rédigez un test qui provoque volontairement une erreur, puis montrez la sortie du terminal obtenue.

Quand un test échoue, pytest le signale immédiatement en affichant un statut FAILED et en indiqunt précisément où se trouve le problème. Concrètement, il affiche le nom du fichier et la ligne du test qui a échoué, puis le bloc FAILURES qui contient l'expression assert fautive, et les valeurs évaluées. Si l'échec provient plûtot d'une exception, pytest affiche l'exception avec une traceback qui remonte la pile de calls jusqu'à la ligne exacte qui en est responsable. Grâce à ces éléments, on peut localiser rapidement un bug, on sait quel test échoue et à quel endroit, et quelle valeur réelle a été obtenue par rapport à celle attendue.

Pour illustrer cela, j'ai créé un test qui est volontairement faux dans test_calculator.py, puis j'ai exécuté pytest. Voici le code du test:

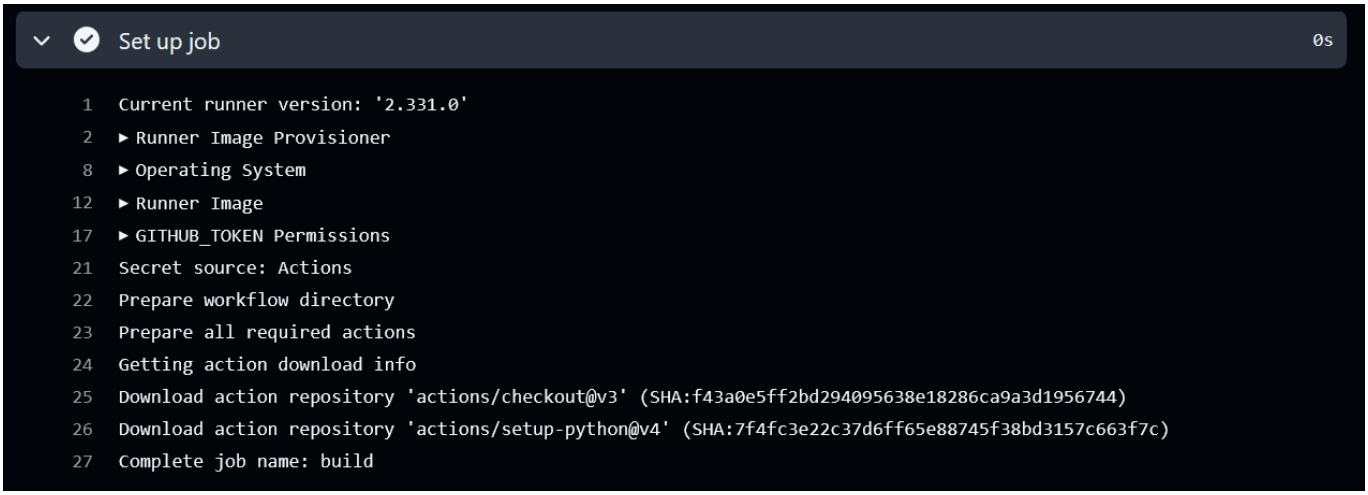
```
def test_addition_fails():
    my_calculator = Calculator()
    assert my_calculator.addition(2, 2) == 5
```

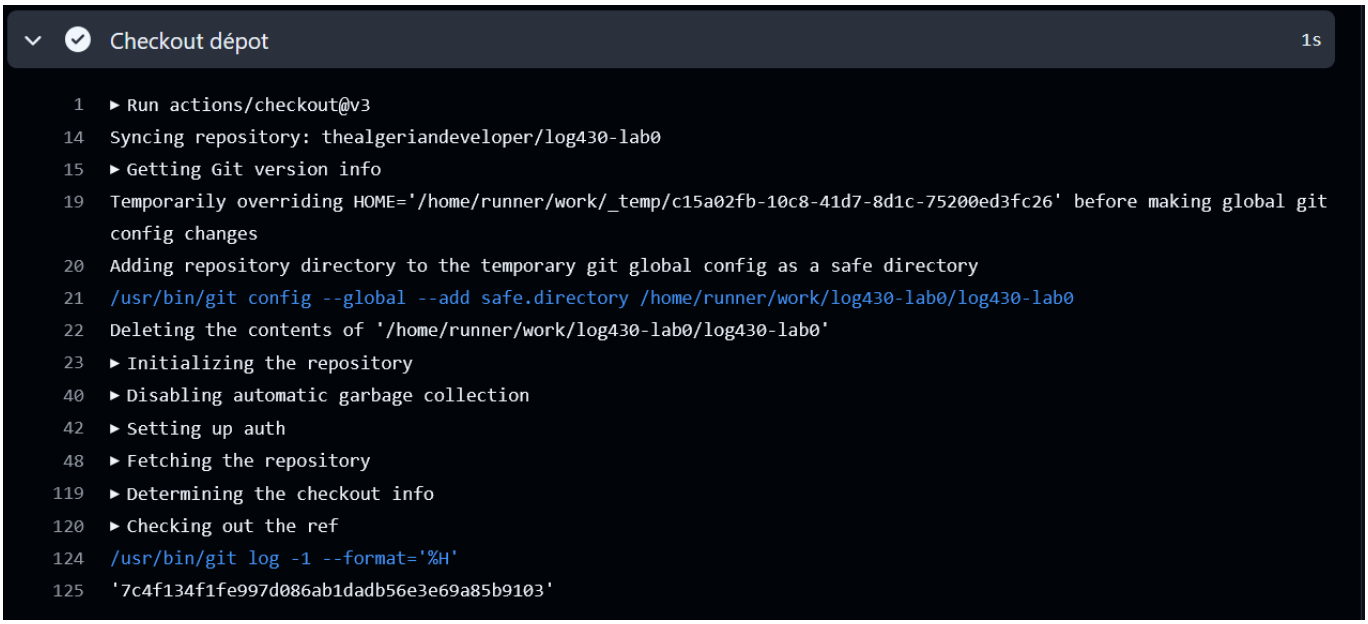
Voici une capture d'écran du output du terminal une fois qu'on a lancé la commande pytest:

```
================================= test session starts =================================
platform win32 -- Python 3.12.3, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\adel_\Desktop\all\Ecole\Juice (nouvelle edition)\LOG430\Lab\lab0\log430-lab0
configfile: pyproject.toml
plugins: anyio-4.11.0, Faker-36.1.1
collected 3 items

src\tests\test_calculator.py ..F                                              [100%]

====================================== FAILURES =======================================
_____________________________ test_addition_fails _____________________________

    def test_addition_fails():
        my_calculator = Calculator()
>       assert my_calculator.addition(2, 2) == 5
E       assert 4 == 5
E        +  where 4 = addition(2, 2)
E        +    where addition = <calculator.Calculator object at 0x0000025125BAE060>.addition

src\tests\test_calculator.py:20: AssertionError
================================= short test summary info =================================
FAILED src/tests/test_calculator.py::test_addition_fails - assert 4 == 5
============================== 1 failed, 2 passed in 0.18s ==============================
```

2. Que fait GitHub pendant les étapes de « setup » et « checkout » ? Veuillez inclure la sortie du terminal GitHub CI dans votre réponse.

Pendant les étapes setup et checkout d'un workflow GitHub Actions, GitHub prépare d'abord le job via une étape Set up job. Cette étape sert à initialiser l'environnement d'exécution sur le runner afin que les étapes suivantes s'exécutent dans un environnement qui est cohérent. Ensuite, l'étape checkout récupère le dépôt dans l'espace de travail du runner pour que le workflow puisse accéder au code source et exécuter des commandes comme l'installation des dépendances et l'exécution des tests. Par défaut, actions/checkout récupère uniquement le commit associé au event qui a déclenché le cworkflow, sauf si on change la configuration.





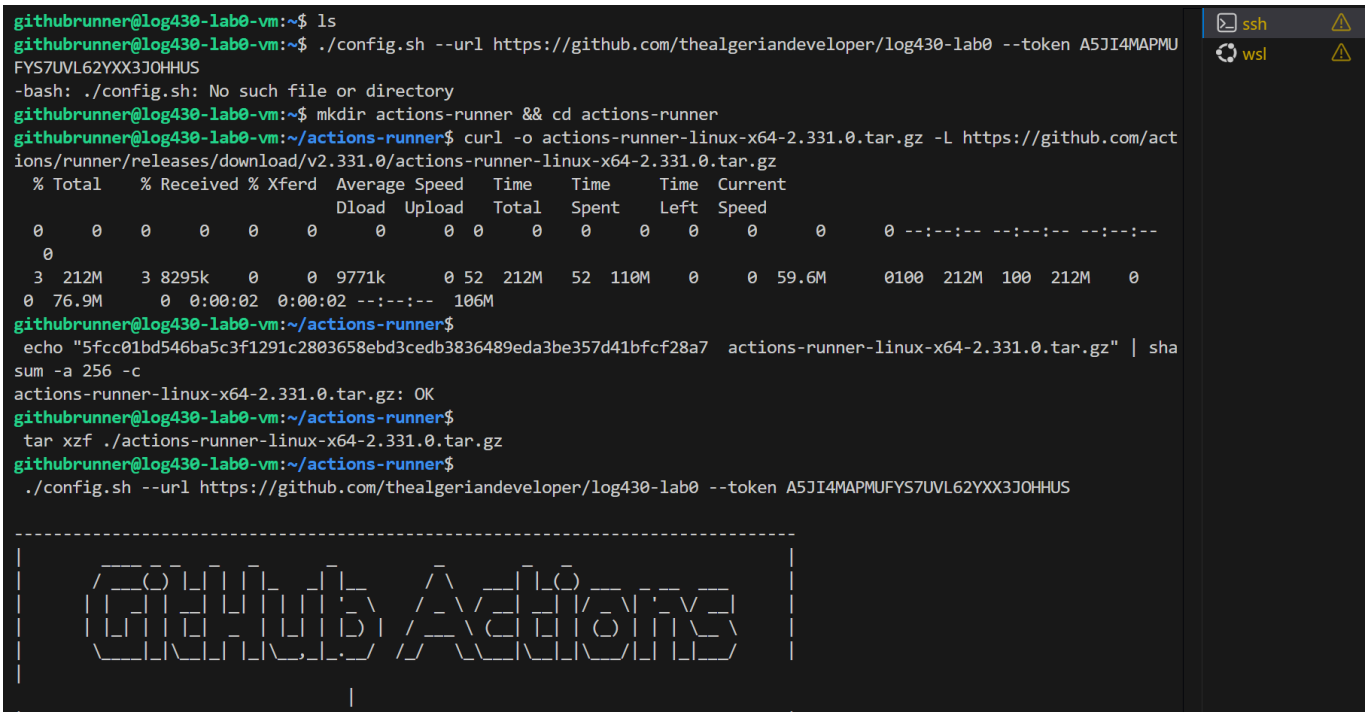
3. Quel type d'informations pouvez-vous obtenir via la commande top ? Veuillez donner quelques exemples. Veuillez inclure la sortie du terminal dans votre réponse.

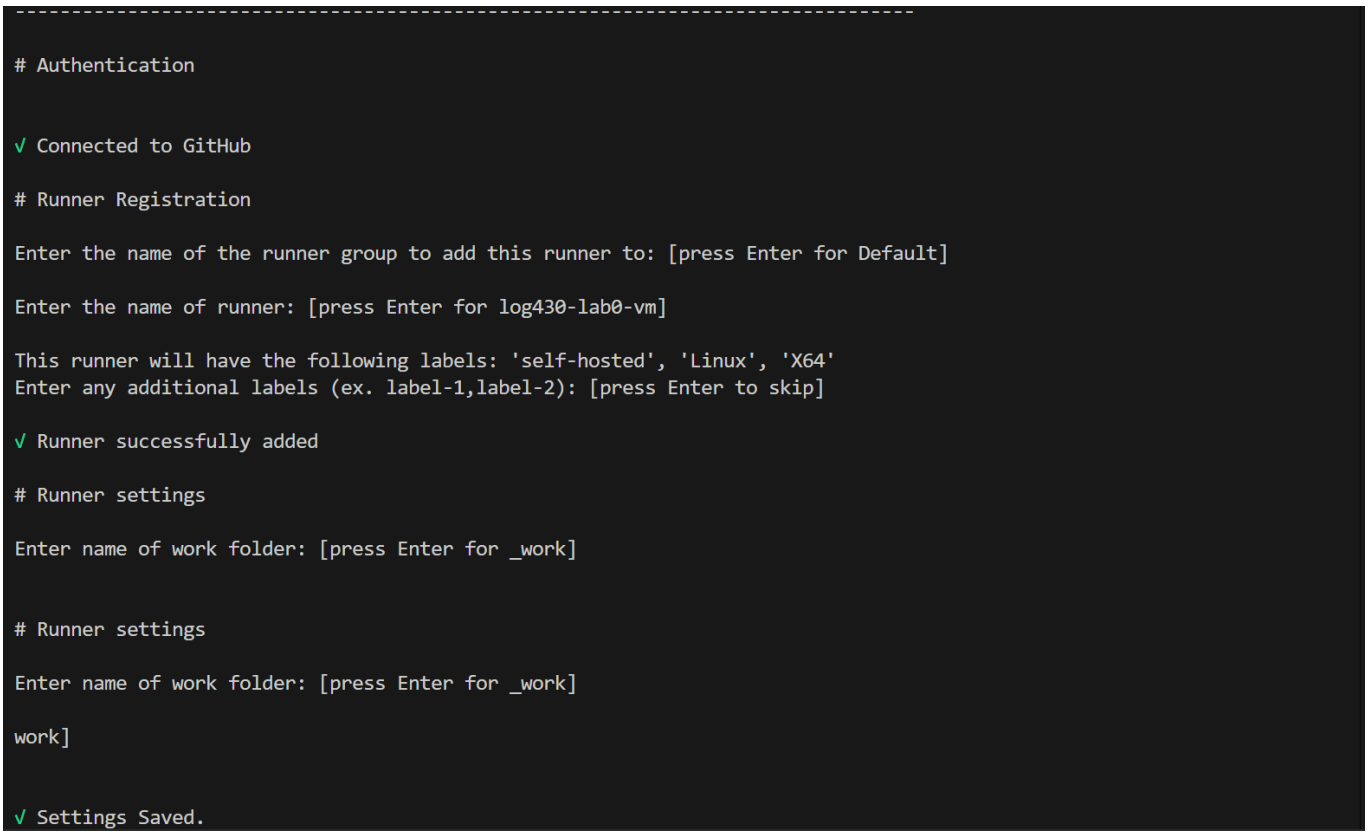
La commande top sert à surveiller en temps réel l'état de la VM et la consommation de ses ressources. Elle donne d'abord un résumé global du système puis uns liste des processus triés. On peut y retrouver le load average, le uptime, le nombre total de processus et leur état, l'utilisation du CPU en pourcentage, l'utilisation de la mémoire vive, l'utilisation de la mémoire swap et des informations reliées à chaque processus comme le PID.

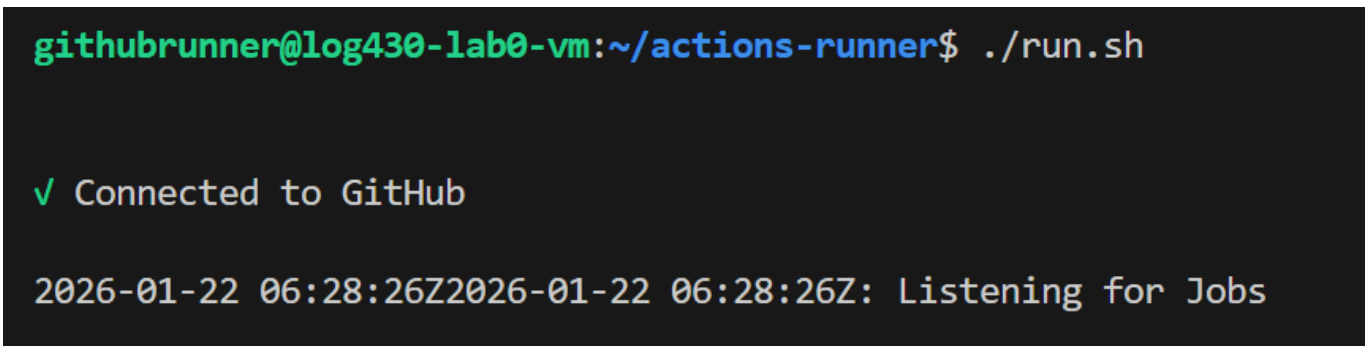
```
root@log430-lab0-vm:~# top
top - 06:12:26 up  1:01,  1 user,  load average: 0.00, 0.10, 0.07
Tasks:  79 total,   1 running,  78 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.3 st
MiB Mem :   3649.5 total,   2440.3 free,    146.0 used,   1063.2 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   3436.6 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
      1 root      20   0  167360  12820   8400 S   0.0   0.3   0:04.74 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 slub_flushwq
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
      8 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-events_+
     10 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
     11 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_trace
     12 root      20   0       0      0      0 S   0.0   0.0   0:00.37 ksoftirqd/0
     13 root      20   0       0      0      0 I   0.0   0.0   0:00.59 rcu_sched
     14 root      rt   0       0      0      0 S   0.0   0.0   0:00.00 migration/0
     15 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/0
     16 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kdevtmpfs
     17 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 inet_frag_wq
     18 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kauditd
     19 root      20   0       0      0      0 S   0.0   0.0   0:00.00 oom_reaper
     20 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 writeback
     42 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kblockd
     43 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 blkcg_punt_bio
     44 root      20   0       0      0      0 I   0.0   0.0   0:00.01 kworker/0:1-cgroup_d+
     45 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 tpm_dev_wq
     46 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 ata_sff
```

Déploiement

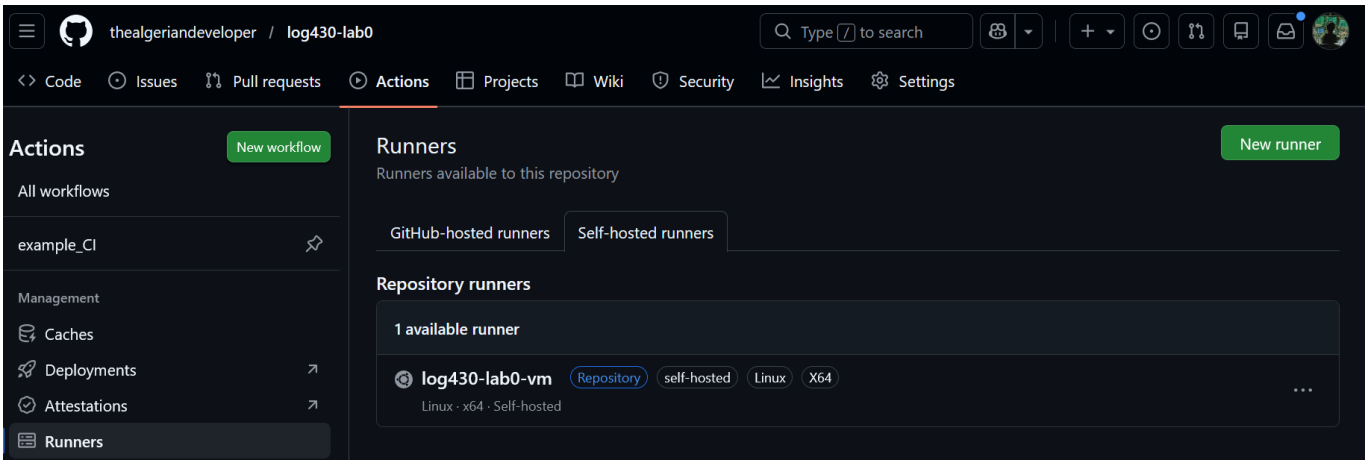
Dans ce laboratoire, le déploiement continu a été automatisé à l'aide d'un GitHub Runner self-hosted installé directement sur la VM. L'idée est que le job GitHub Actions ne s'exécute pas sur un runner cloud, mais sur la VM elle-même, ce qui permet de déployer sans ouvrir de ports supplémentaires et sans dépendre d'un mécanisme de webhook externe. Après l'installation des dépendances du runner, un utilisateur a été créé pour respecter la contrainte du runner (ne pas être exécuté en sudo), puis le runner a été enregistré sur le dépôt GitHub avec le script config.sh en fournissant l'URL du repo et un token d'inscription.







Une fois configuré, le runner a été démarré et confirmé être Online dans l'interface GitHub:



Après cela, un workflow CD a été ajouté au repo dans le fichier .github/workflows/cd.yml et configuré pour se déclencher automatiquement à chaque push un peu comme le CI. Ce workflow utilise runs-on: self-hosted, ce qui force GitHub Actions à exécuter le job sur la VM. Le job commence par récupérer le code du repo avec l'étape checkout, puis exécute des commandes de validation.