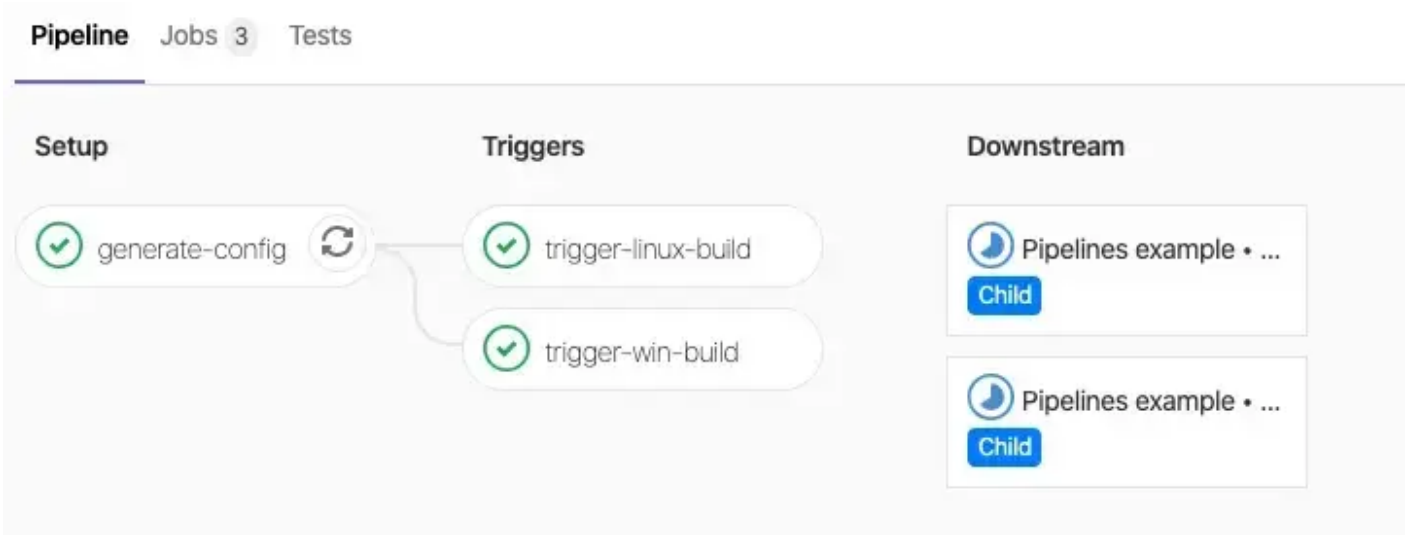
Les pipelines dynamiques

Les Pipelines Enfant Dynamiques

Heureusement, avec les **Dynamic Child Pipelines** de GitLab CI/CD, nous pouvons générer dynamiquement ces étapes, adaptant ainsi le processus de manière agile et efficiente. Cette approche permet de simplifier considérablement la configuration des pipelines et d'améliorer l'automatisation des processus CI/CD, en adaptant automatiquement les étapes de test en fonction du nombre et des spécificités des clients. Dans ce blog, nous explorerons en détail comment tirer parti des **Dynamic Child Pipelines** pour optimiser nos workflows de déploiement et de test dans des contextes multi-clients.



Je vais utiliser Ansible pour générer des étapes dynamiques. D'ailleurs, je vais utiliser 2 astuces Ansible :

- Installer la même application pour différents clients sur une seule machine
- Générer un seul fichier en utilisant un template pour tous ces clients

Mettons en place notre exemple

Nous avons l'application à installer sur la même machine pour les clients : client1, client2 et client3. L'application est installée dans le répertoire `/app/<nom-du-client>`

Bien sûr pour simplifier, je vais simplement déposer un fichier vide dans le répertoire.

Création de l'inventaire

Pour notre machine, je vais utiliser une machine provisionner pour ce tutoriel. Le fichier d'inventaire :

```
[all]
client1 ansible_host=default
client2 ansible_host=default
client3 ansible_host=default
```

Si on génère l'inventaire avec la commande Ansible avec le module ping, vous allez voir que nous aurons bien 3 appels.

```
ansible -i hosts all -m ping all
client3 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3.6"
    },
    "changed": false,
    "ping": "pong"
}
client1| SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3.6"
    },
    "changed": false,
    "ping": "pong"
}
client1 | SUCCESS => {
```

```
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3.6"
    },
    "changed": false,
    "ping": "pong"
}
```

Je vais ajouter des variables pour chaque client en créant dans le répertoire `host_vars` les fichiers `<nomduclient>.yml` qui contiendra :

```
name: <un-nom>
```

Lançons la commande ansible-inventory :

```
ansible-inventory -i hosts --graph --vars



@all:
|--@ungrouped:
| |--client1
| | |--{ansible_host = default}
| | |--{name = toto}
| |--client2
| | |--{ansible_host = default}
| | |--{name = titi}
| |--client3
| | |--{ansible_host = default}
| | |--{name = tata}
```

La variable name sera utilisé dans le fichier template.

Installons l’application

Maintenant le playbook pour installer l’application `install-app.yml` qui ne fait que créer le répertoire.

```
---
- hosts: all
gather_facts: false



tasks:
- name: create folder /app
become: true
ansible.builtin.file:
path: /app
state: directory
mode: 0755
- name: create folder
become: true
ansible.builtin.file:
path: /app/{{ inventory_hostname }}
state: directory
mode: 0644
```

```
ansible-playbook -i hosts install-app.yml
```

Écriture du test

Nous allons utiliser l’outil testinfra :

```
pip install pytest-testinfra
```

Créer un fichier test_app.py dont le contenu est le suivant :

```
def test_app_install(host):
all_variables = host.ansible.get_variables()
hostname = all_variables['inventory_hostname']
directory = "./app/%s" % hostname
```

```
assert host.file(directory).exists
assert host.file(directory).is_directory
```

On récupère les variables Ansible grâce à la fonction `host.ansible_.get_variables`. Ensuite, on construit le chemin avec la variable `inventory_hostname`. On teste s'il s'agit bien d'un répertoire et qu'il existe.

Testons-le sur un seul client avec l'option `--hosts` :

```
py.test --ansible-inventory=hosts --connection=ansible test_app.py --hosts client3
========================================= test session starts
=========================================
platform linux -- Python 3.9.7, pytest-6.2.5, py-1.10.0, pluggy-1.0.0
rootdir: /home/vagrant/Projets/perso/test-dyn
plugins: testinfra-6.4.0
collected 1 item



test_app.py . [100%]



========================================== 1 passed in 1.37s
==========================================
```

Passons maintenant à l'écriture du pipeline et en particulier de la partie générant le pipeline dynamique.

Mettons en place le pipeline dynamique

Une des possibilités offertes par cette nouvelle fonctionnalité est de pouvoir inclure un autre pipeline de similaire aux pipelines parent/enfant :

```
tests:
stage: test
trigger:
include:
- artifact: test-ci.yml
job: generate
```

```
strategy: depend
```

On peut inclure un fichier de pipeline externe généré par un job précédent. Pour rendre **dépendant les enfants** du parent, on ajoute la clé `strategy` à `depend`. Ici un job `generate` qui produit un artefact `test-ci.yml`. En voici le contenu :

```
generate:
stage: generate
script:
- ansible-playbook -i hosts generate-ci.yml
artifacts:
paths:
- test-ci.yml
```

Voici le contenu du playbook :

```
---
- hosts: all
gather_facts: false



tasks:
- name: generate dynamique ci stages
blockinfile:
mode: 0644
create: true
path: ./test-ci.yml
block: "{{ lookup('template', 'templates/test-ci.yml.j2') }}"
marker: "# {mark} Test for {{ inventory_hostname }}"
delegate_to: localhost
```

Ce playbook génère sur localhost un fichier en concaténant dans un fichier les 3 sorties produites par le lookup template dont en voici le contenu :

```
test-{{ name }}:
image: python:3.9.7-alpine3.14
stage: test
```

```
script:
- apk add py3-pip
- pip3 install ansible pytest-testinfra
- py.test --ansible-inventory=hosts --connection=ansible test_app.py --hosts {{
inventory_hostname }}
```

Lançons le playbook :

```
ansible-playbook -i hosts generate-ci.yml
```

Ce qui produit comme fichier :

```
# BEGIN Test for client2
test-titi:
stage: test
script:
- py.test --ansible-inventory=hosts --connection=ansible test_app.py --hosts client2
# END Test for client2
# BEGIN Test for client1
test-toto:
stage: test
script:
- py.test --ansible-inventory=hosts --connection=ansible test_app.py --hosts client1
# END Test for client1
# BEGIN Test for client3
test-tata:
stage: test
script:
- py.test --ansible-inventory=hosts --connection=ansible test_app.py --hosts client3
# END Test for client3
```

Tout à fait le résultat attendu.

Le fichier `.gitlab-ci.yml` au complet.

```
---
stages:
- deploy
- generate
- test



deploy:
stage: deploy
script:
- apk add --no-cache bc cargo gcc libffi-dev musl-dev openssl-dev py3-pip python3 python3-dev
rust
- pip install ansible
- ansible-playbook -i hosts install-app.yml



generate:
stage: generate
needs:
- deploy
script:
- ansible-playbook -i hosts generate-ci.yml
artifacts:
paths:
- test-ci.yml



tests:
stage: test
needs:
- generate
- deploy
trigger:
include:
- artifact: test-ci.yml
job: generate
strategy: depend
```

Avantages de cette Approche

En utilisant les **Dynamic Child Pipelines**, nous bénéficions de plusieurs avantages :

- **Modularité** : Chaque pipeline enfant peut être géré et modifié indépendamment, ce qui facilite la maintenance.
- **Flexibilité** : Les pipelines enfants peuvent être adaptés aux besoins spécifiques de chaque client.
- **Scalabilité** : Il est facile d’ajouter de nouveaux clients ou de modifier les configurations existantes sans perturber l’ensemble du processus.

Pour rappel, GitLab CI/CD offre des options avancées pour contrôler quand et comment les pipelines enfants sont déclenchés. Utilisez judicieusement les règles de déclenchement pour assurer que les pipelines enfants ne s’exécutent que lorsque c’est nécessaire, optimisant ainsi les ressources et le temps de traitement.

Revision #1
Created 3 February 2025 16:59:18 by Nicolas
Updated 3 February 2025 17:09:18 by Nicolas