Maitrisez les templates

L’efficacité et la fiabilité des processus de Continuous Integration/Continuous Deployment (CI/CD) sont importants. GitLab, en tant que plateforme intégrée, offre des outils puissants pour automatiser ces processus. Cependant, avec le nombre et la complexité croissants des projets, la gestion des pipelines CI/CD peut devenir fastidieuse et sujette à erreurs. C’est là qu’intervient le concept de templates des pipelines GitLab CI/CD.

Mais pourquoi donc ?

La factorisation des pipelines consiste à créer et à réutiliser des portions de code de pipeline pour en faciliter la maintenance. Cette approche s’aligne sur le principe DRY (Don’t Repeat Yourself) de la programmation, visant à réduire la répétition du code source. **En factorisant les pipelines, les équipes de développement peuvent mettre à jour des processus complexes en un seul endroit**, garantissant ainsi une plus grande cohérence et facilité de gestion.

En outre, traiter le code des pipelines avec la même rigueur que le code de l’application elle-même est essentiel pour assurer la sécurité et l’optimisation. Dans un environnement où les déploiements fréquents et les changements rapides sont la norme, **les pipelines CI/CD doivent être conçus pour être à la fois robustes et flexibles**.

Notions importantes

Avant de voir comment créer des templates Gitlab CI/CD, il est important d’introduire quelques notions.

Extends

`extends` permet à un job d'hériter des configurations d'un autre, tout en lui permettant d'ajouter ou de surcharger certaines parties de cette configuration. Cela aide à éviter les répétitions et à garder les pipelines DRY (Don't Repeat Yourself).

Imaginons que vous ayez une configuration de base pour tous les jobs de test :

```
.test_base.yml
.test_base:
stage: test
script:
- echo "Exécution des pré-tests..."
```

Un job spécifique de test unitaire pourrait étendre cette base :

```
unit_test:
extends: .test_base
script:
- echo "Exécution des tests unitaires..."
- run-unit-tests.sh
```

Dans cet exemple, `unit_test` hérite du stage `test` et du script initial de `.test_base`, tout en ajoutant ses propres étapes de script.

Les ancres YAML

Les ancres YAML sont une fonctionnalité du langage YAML (YAML Ain't Markup Language) qui permet de réutiliser des parties d'un document YAML. Cette fonctionnalité est particulièrement utile pour éviter la répétition de structures de données similaires et pour maintenir des configurations cohérentes dans de grands fichiers YAML, comme ceux souvent utilisés dans la configuration des pipelines CI/CD, les fichiers Docker Compose, etc.

Comment Fonctionnent les Ancres YAML ?

1. **Définition d'une Ancre (`&`)** :
   - Une ancre est définie en utilisant le symbole `&` suivi d'un nom unique. Elle marque une section du YAML que vous souhaitez réutiliser ailleurs.
   - Exemple : `&default_settings`.
2. **Référencement d'une Ancre (`*`)** :
   - Pour utiliser ou référencer l'ancre ailleurs dans le document, vous utilisez le symbole `*` suivi du nom de l'ancre.

- Exemple : `*default_settings`.

3. **Merge Key (`<<`)** :
   - La clé spéciale `<<` est utilisée pour indiquer que toutes les propriétés de l'ancre référencée doivent être fusionnées dans le dictionnaire courant.
   - C'est utile pour combiner les configurations de base avec des configurations supplémentaires.

Exemple Pratique

Voici un exemple simple pour illustrer l'utilisation des ancres :

```
.job_template: &job_configuration
image: ruby:2.6
services:
- postgres
- redis



test1:
<<: *job_configuration
script:
- test1 project



test2:
<<: *job_configuration
script:
- test2 project
```

Cela peut être utile si on souhaite utiliser deux template de job dans un job :

```
.job_template: &job_configuration
script:
- test project
tags:
- dev



.postgres_services:
```

```
services: &postgres_configuration
- postgres
- ruby


.mysql_services:
services: &mysql_configuration
- mysql
- ruby


test:postgres:
<<: *job_configuration
services: *postgres_configuration
tags:
- postgres


test:mysql:
<<: *job_configuration
services: *mysql_configuration
```

On peut aussi l’utiliser pour étendre des variables, mais aussi les scripts :

```
variables: &global-variables
SAMPLE_VARIABLE: sample_variable_value
ANOTHER_SAMPLE_VARIABLE: another_sample_variable_value


# a job that must set the GIT_STRATEGY variable, yet depend on global variables
job_no_git_strategy:
stage: cleanup
variables:
<<: *global-variables
GIT_STRATEGY: none
script: echo $SAMPLE_VARIABLE
```

Créer et inclure des templates Gitlab-ci

Dans un premier temps, on peut développer ses templates dans un projet et les inclure localement. Une fois cette phase de développement terminé, il sera temps de passer par une solution de centralisation des templates.

Templates locaux

Prenons l'exemple d'un template pour les tests unitaires. Vous pouvez créer un fichier `unit_tests.yml` contenant la configuration suivante :

```
unit_tests.yml
unit_tests:
stage: test
script:
- echo "Exécution des tests unitaires..."
- ./run-unit-tests.sh
```

Dans votre fichier principal `.gitlab-ci.yml`, vous pouvez inclure ce template comme suit :

```
include:
- local: '/templates/unit_tests.yml'


stages:
- build
- test
- deploy


build_job:
stage: build
script:
- echo "Construction du projet..."
```

```
# L'utilisation du template pour les tests unitaires
unit_tests:
extends: .unit_tests
```

Voici un autre exemple pour un template de déploiement :

```
deployment.yml
deployment:
stage: deploy
script:
- echo "Déploiement de l'application..."
- ./deploy-script.sh
```

Et son inclusion dans le pipeline principal :

```
include:
- local: '/templates/deployment.yml'


# ... Autres configurations ...


# Utilisation du template pour le déploiement
deploy_job:
extends: .deployment
```

Templates centralisés

Pour stocker les différents composants de votre template, créez tout simplement un **projet gitlab indépendant**. Créez-y vos fichiers `yaml`.

Par exemple, nous voulons définir un template d'installation de packages `nodejs` (le premier commentaire est le nom du projet suivi du nom du fichier) :

```
#template-ci/install.yml
install:
script:
- npm install
```

Ensuite dans vos projets d’applications, il suffit d’inclure dans votre `.gitlab-ci.yml` un appel à ce script.

```
#mon-app1/.gitlab-ci.yml
include:
- project: 'template-ci'
file: 'install.yml'
ref: 'master'
```

Vous remarquez certainement la balise `ref`. Elle permet de pointer **vers une branche du projet de template**. Cela va vous permettre de pouvoir faire évoluer votre ci sans impacter toute votre production. En effet, dans un projet précis, vous pourrez utiliser une branche de votre template pour le faire évoluer par exemple. Vous pourriez également créer des branches pour des variantes de votre template de ci/cd.

Conditionnement des include

Depuis la version 14.3 de gitlab, il est possible de conditionner les include par des règles.

```
include:
- local: builds.yml
rules:
- if: '$INCLUDE_BUILDS == "true"'
```

Bonnes pratiques

Dans l’écosystème DevOps, les pipelines CI/CD ne sont pas de simples bouts de code ; ils sont une part essentielle de la base de code. Cela signifie qu’ils doivent être traités avec la même rigueur que le code source de l’application. La gestion de ces pipelines implique :

1. **Versionnage** : Tout comme le code source, les pipelines doivent être versionnés. Cela permet de suivre les modifications, de revenir à des versions antérieures en cas de problème et de comprendre l'évolution du pipeline au fil du temps.
2. **Revue de Code** : Les modifications apportées aux fichiers de pipeline doivent passer par des revues de code. Cela assure une vérification par les pairs et aide à maintenir la qualité et la sécurité du pipeline.
3. **Tests Automatisés** : Les pipelines devraient être testés pour vérifier leur fiabilité. Cela peut inclure des tests d'intégration pour s'assurer que le pipeline fonctionne correctement dans différents environnements.
4. **Documentation** : Une documentation claire est essentielle pour assurer que les pipelines soient compréhensibles et maintenables. Elle devrait inclure des informations sur la structure du pipeline, son fonctionnement et les modifications apportées au fil du temps.
5. **Exposer vos templates** : C'est bien d'avoir des templates mais il faut s'assurer que les utilisateurs puissent les trouver facilement. C'est là qu'intervient un outil comme **R2DevOps** qui permet d'afficher un catalogue de templates de manière simplifié et structuré. R2DevOps permet aussi de d'identifier où et comment sont utilisés les templates.

Revision #1
Created 3 February 2025 16:44:08 by Nicolas
Updated 3 February 2025 17:09:18 by Nicolas