Introduction

GitLab CI/CD est devenu un incontournable de l’automatisation du développement. C’est une solution puissante qui permet de rationaliser et d’automatiser le processus de construction, de test et de déploiement de logiciels, ce qui permet aux équipes de développement de gagner du temps, de réduire les erreurs et d’augmenter la qualité de leurs applications.

Lorsque vous travaillez sur des projets de développement, la gestion efficace des **pipelines** est essentielle pour garantir une **intégration continue** et une **livraison continue** sans heurts. **GitLab CI/CD** est un outil puissant qui vous permet de créer ces pipelines et de rationaliser votre processus de développement. Avant de plonger dans les détails de son utilisation, commençons par comprendre les concepts de base.

Comprendre les concepts de base de GitLab CI/CD

Qu’est-ce qu’un pipeline ?

Un pipeline dans **GitLab CI/CD** est constitué une série d’étapes qui sont exécutées automatiquement chaque fois qu’il y a un changement dans votre référentiel de code source. Ces étapes sont conçues pour automatiser diverses tâches, telles que la compilation du code, l’exécution de tests, le déploiement d’applications et bien plus encore. L’objectif principal d’un pipeline est d’assurer la cohérence, la qualité et la rapidité du processus de développement.

Chaque pipeline est déclenché par un événement spécifique, comme une nouvelle validation de code (push) ou la création d'une demande de fusion (merge request). GitLab CI offre une grande flexibilité pour configurer les déclencheurs en fonction de besoins spécifiques.

Les jobs et les étapes dans GitLab CI

Les **pipelines GitLab CI/CD** sont composés de **jobs**, qui sont des unités d'exécution de tâches spécifiques. Ces **jobs** sont regroupés en **étapes** (stages), qui représentent des phases logiques du processus de développement. Par exemple, un pipeline typique pourrait comprendre les étapes suivantes :

1. **Build** : Cette étape compile le code source pour créer une application exécutable ou un artefact.
2. **Test** : Les tests unitaires, les tests d'intégration et d'autres vérifications de qualité sont exécutés ici.
3. **Deploy** : Si les tests réussissent, cette étape déploie l'application sur un environnement de test ou de production.

Les jobs au sein de chaque étape sont exécutés de manière parallèle, ce qui permet d'accélérer le processus global de développement. **GitLab CI/CD** gère également automatiquement les dépendances entre les jobs, de sorte qu'ils sont exécutés dans le bon ordre.

Les artifacts

Un **artifact** peut contenir des fichiers et/ou dossiers qui vont être stockés au sein des pipelines pour être utilisé par d'autres taches.

Les runners

Les runners sont des agents d'exécution qui exécutent les jobs de CI/CD. Ils peuvent être installés sur différentes machines, y compris des serveurs dédiés ou des conteneurs Docker et sont responsables de l'exécution des tâches définies dans vos pipelines.

Les tags

Les **tags** permettent de définir un **runner** spécifique dans la liste de tous les runners disponibles d'un projet.

Création d’un pipeline CI/CD Gitlab

Prérequis

Pour commencer, nous avons besoin de : • Un compte GitLab.com • Un repo GitLab

Ici, nous utiliserons les runners de Gitlab.com, mais si vous avez installé votre propre serveur gitlab, vous devrez ajouter vos propres runners sur vos serveurs.

Configuration de base du fichier `.gitlab-ci.yml`

La configuration des pipelines **GitLab CI/CD** est définie dans un fichier au format YAML nommé `.gitlab-ci.yml`, qui se trouve à la racine de votre projet. Ce fichier contient des instructions pour spécifier les étapes, les jobs, les dépendances et d’autres paramètres importants pour votre pipeline.

Voici un exemple simple de fichier `.gitlab-ci.yml` :

```
job 0:
stage: .pre
script: echo "make something useful before build stage"



build-job:
tags:
- ruby
stage: build
script:
- echo "Hello, $GITLAB_USER_LOGIN!"



test-job1:
stage: test
script:
- echo "This job tests something"
```

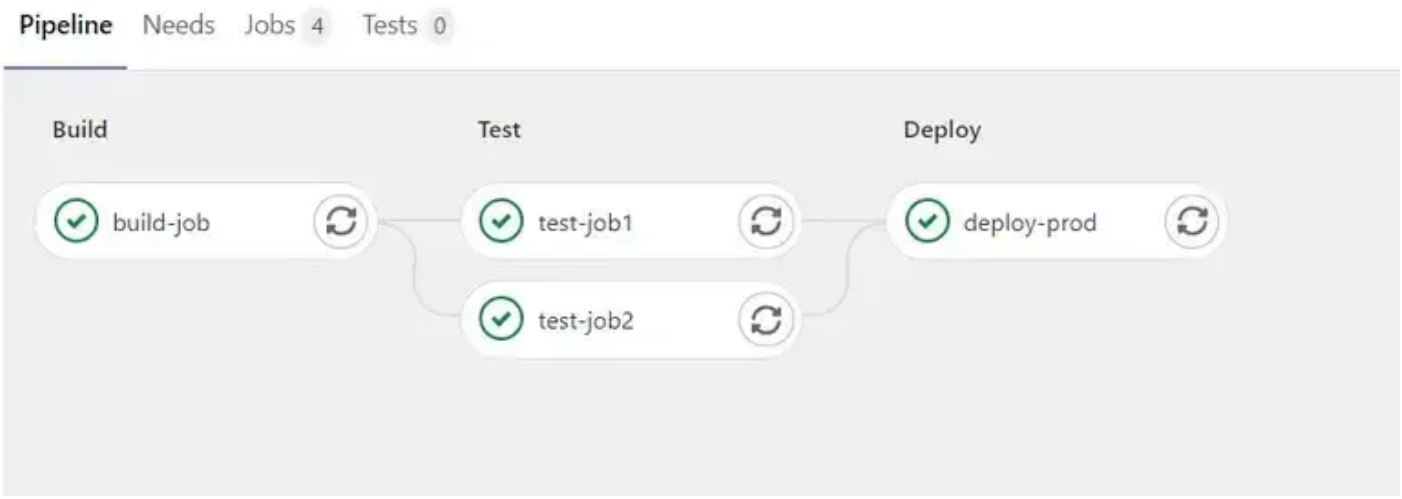
```
test-job2:
stage: test
script:
- echo "This job tests something, but takes more time than test-job1."
- echo "After the echo commands complete, it runs the sleep command for 20 seconds"
- echo "which simulates a test that runs 20 seconds longer than test-job1"
- sleep 20


deploy-prod:
stage: deploy
script:
- echo "This job deploys something from the $CI_COMMIT_BRANCH branch."
```

Dans cet exemple, nous avons :

- Quatre stages (étapes) : .pre, build, test et deploy
- L'étape test contient deux jobs : test-job1 et test-job2
- Les jobs sont de simples echo qui affichent des variables prédéfinies de gitlab.

En fait, il existe deux stages prédéfinis : .pre et .post qui sont toujours lancés respectivement au début et à la fin du ci.

Si vous commitez votre fichier, vous pourrez suivre l'exécution du pipeline dans le menu CI/CD.



Ici le résultat de l'étape build :

Vous remarquerez que `gitlab.com` fait appel à un runner docker utilisant une image `ruby:2.5`.

before_script et after_script

GitLab CI/CD offre la possibilité d'exécuter des scripts de préparation et de nettoyage avant et après l'exécution des jobs des pipelines. Cela peut être utile pour la configuration de l'environnement, la gestion des dépendances ou le nettoyage des artefacts temporaires. On utilise les mots clés `before_script` et `after_script`.

```
before_script:
- echo "Initialisation de l'environnement..."



after_script:
- echo "Nettoyage de l'environnement..."
```

Ces scripts de préparation et de nettoyage vous permettent d'automatiser les tâches courantes, de garantir la cohérence de l'environnement d'exécution et de maintenir votre pipeline propre et efficace.

Utiliser d'autres images dans votre pipeline gitlab CI/CD

On peut en premier lieu changer l'image par défaut de tout le pipeline.

```
default:
image: ruby:2.7.2



build-job:
stage: build
script:
- echo "Hello, $GITLAB_USER_LOGIN!"



...
```

Ce qui donne :

Il est possible d’utiliser différentes images pour chacune des étapes. Pour cela, il suffit de le spécifier :

```
default:
image: ruby:2.7.2



build-job:
image: alpine:3.12
stage: build
script:
- echo "Hello, $GITLAB_USER_LOGIN!"
...
```

Ce qui donne :

Les images utilisées sont celles que vous retrouvez dans le docker hub. Attention au `rate limit` de docker et stockez vos images dans la registry de Gitlab.

Utiliser des variables

Il est possible de créer vos propres variables en utilisant variables dans vos jobs. Ces variables sont de la forme `clé: valeur`

```
default:
image: ruby:2.7.2



build-job:
image: alpine:3.12
stage: build
variables:
test: "je suis un test"
script:
- echo "$test"
...
```

Il est possible d’utiliser des variables dans d’autres variables :

```
job:
variables:
FLAGS: '-al'
LS_CMD: 'ls "$FLAGS"'
script:
- 'eval "$LS_CMD"' # Executes 'ls -al'
```

Il est également possible de créer des variables dans les paramètres CI/CD de votre projet ou dans un groupe.

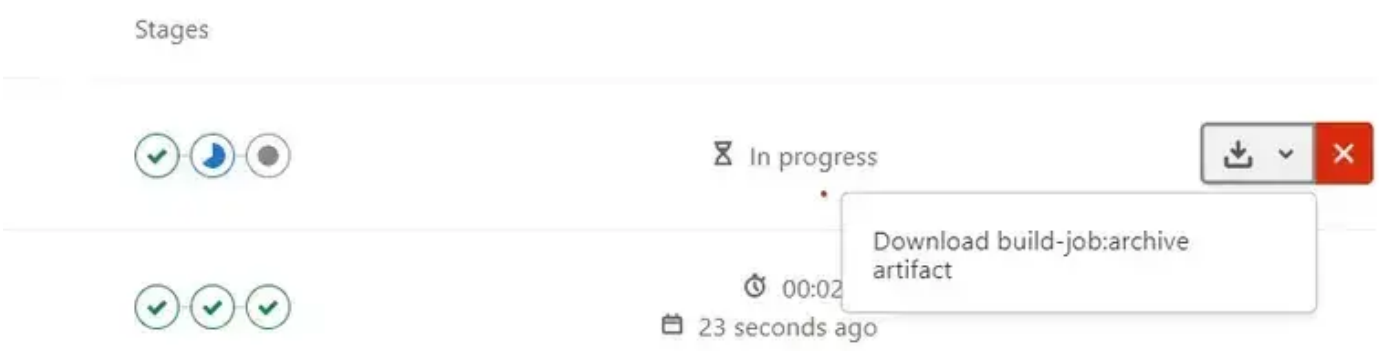
Gérer vos artefacts

On va modifier la tache `build-job` pour stocker le fichier `test.txt` et le conserver pendant une semaine :

```
default:
image: ruby:2.7.2



build-job:
image: alpine:3.12
stage: build
variables:
test: "je suis un test"
script:
- echo "$test" > test.txt
artifacts:
paths:
- test.txt
expire_in: 1 week



...
```

Ce qui donne :

On retrouve ce fichier dans le menu ci/cd pipeline



Utilisation des groupes de ressources pour limiter la concurrence.

Il peut arriver parfois que si deux jobs tournent en même temps cela provoque des erreurs.

Gitlab propose la notion de `resource_group`. Les groupes de ressources permettent de limiter la concurrence des jobs d'un CI. Il est impossible que deux jobs appartenant au même `resource_group` de tourner en même temps, même s'ils sont dans deux pipelines différents, ils s'excluent mutuellement.

Exemple :

```
deploy-to-production:
script: deploy
resource_group: production
```

Bonnes Pratiques et Astuces

Maintenant que vous avez acquis une compréhension de **GitLab CI/CD** et de sa configuration, il est temps d'explorer quelques bonnes pratiques et astuces pour tirer le meilleur parti de cet outil puissant. Les conseils suivants vous aideront à optimiser vos pipelines et à garantir la sécurité de vos déploiements.

Utilisation de Templates CI/CD

Les templates CI/CD de GitLab sont un moyen puissant de réutiliser des configurations de pipeline courantes. Créez et utilisez des templates personnalisés pour vos projets afin de standardiser et de simplifier la configuration des pipelines. Cela permet également de maintenir une cohérence au sein de votre organisation.

Exemple d'utilisation d'un template pour la construction d'une application Java :

```
include:
- template: Java-Maven.gitlab-ci.yml
```

Gestion des Secrets

Lorsque vous travaillez avec **GitLab CI/CD**, il est essentiel de gérer les secrets et les informations sensibles, tels que les clés d'API et les mots de passe. Utilisez **GitLab CI/CD** pour stocker ces informations de manière sécurisée en utilisant les variables d'environnement protégées ou les fichiers de variables. Évitez de les inclure directement dans votre fichier `.gitlab-ci.yml`.

Intégration de la Sécurité

Pensez à intégrer des outils de sécurité tels que SonarQube ou GitLab SAST (Static Application Security Testing) dans vos pipelines de CI/CD. Cela vous permettra d'identifier et de résoudre les vulnérabilités de sécurité dès le début du processus de développement.

Tests en Parallèle

Pour accélérer vos pipelines de tests, envisagez d'exécuter des tests en parallèle sur plusieurs runners. Cela permettra de réduire considérablement le temps nécessaire pour exécuter vos suites de tests et d'obtenir des résultats plus rapidement.

Documentation et Commentaires

N'oubliez pas de documenter vos pipelines et vos scripts. Incluez des commentaires clairs dans votre fichier `.gitlab-ci.yml` pour expliquer le but de chaque étape et de chaque job. Cela facilite la collaboration au sein de l'équipe et la maintenance à long terme.

Optimisation des Performances

Surveillez les performances de vos runners GitLab et assurez-vous qu’ils sont dimensionnés correctement pour gérer la charge de travail de vos pipelines. Optimisez également vos scripts pour minimiser les temps d’exécution.

Conclusion

GitLab CI/CD offre une flexibilité exceptionnelle pour configurer et personnaliser vos **pipelines CI/CD** en fonction des besoins spécifiques de vos projets. Vous avez appris à définir des étapes et des jobs, à utiliser le fichier `.gitlab-ci.yml` pour configurer votre pipeline et à comprendre comment **GitLab CI/CD** automatise le déclenchement des pipelines en réponse à des événements tels que les validations de code ou les demandes de fusion.

Il est important de noter que **GitLab CI/CD** est un outil en **constante évolution**, offrant de nouvelles fonctionnalités et des améliorations régulières. Par conséquent, il est judicieux de rester à jour avec la documentation officielle de GitLab.

Created 2025-02-03 17:37:18 UTC by Nicolas
Updated 2025-02-03 18:09:18 UTC by Nicolas