

# Gitlab CI/CD

- [Introduction](#)
- [Les Variables et Secrets](#)
- [Maîtrisez les templates](#)
- [Les Environnements](#)
- [Les Conditions](#)
- [Les pipelines Parent/Enfant](#)
- [Les pipelines dynamiques](#)
- [Maîtriser la CLI](#)

# Introduction



# GitLab

**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 tâches.

## 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
```

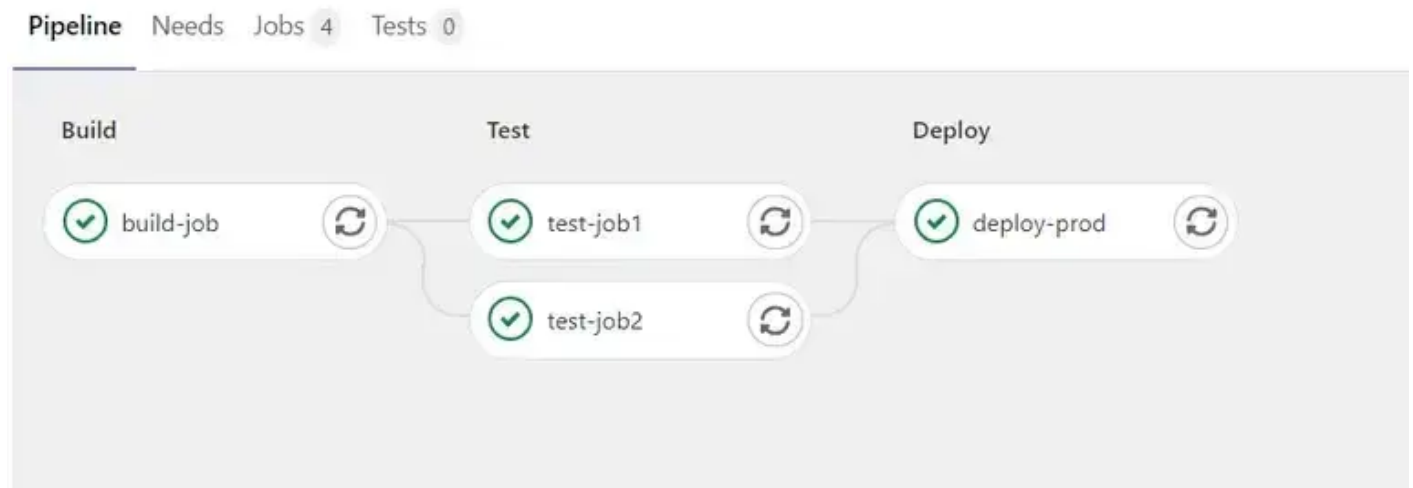
```
deploy-prod:
  stage: deploy
  script:
    - echo "This job deploys something from the $CI_COMMIT_BRANCH branch."
```

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



Ici le résultat de l'étape build :

```
Running with gitlab-runner 13.9.0-rc2 (69c049fd)
on docker-auto-scale ed2dce3a
feature flags: FF_GITLAB_REGISTRY_HELPER_IMAGE:true
Preparing the "docker+machine" executor
Using Docker executor with image ruby:2.5 ...
Pulling docker image ruby:2.5 ...
Using docker image
sha256:b8f85f05a4c615b08acb073a366ccf8559bdde860861712bb178fb4ee01102a3 for
ruby:2.5 with digest
```

```
ruby@sha256:edc40b439ce1e771849bb398f70b4e202c18d30a0db391f437820bd712774c75 ...
Preparing environment
Running on runner-ed2dce3a-project-25219583-concurrent-0 via runner-ed2dce3a-srm-
1615992594-66af060f...
Getting source from Git repository
$ eval "$CI_PRE_CLONE_SCRIPT"
Fetching changes with git depth set to 50...
Initialized empty Git repository in /builds/Bob74/test-gitlab/.git/
Created fresh repository.
Checking out eb2lab21 as master...
Skipping Git submodules setup
Executing "step_script" stage of the job script
00:01
Using docker image
sha256:b8f85f05a4c615b08acb073a366ccf8559bdde860861712bb178fb4ee01102a3 for
ruby:2.5 with digest
ruby@sha256:edc40b439ce1e771849bb398f70b4e202c18d30a0db391f437820bd712774c75 ...
$ echo "Hello, $GITLAB_USER_LOGIN!"
Hello, Bob74!
Cleaning up file based variables
00:01
Job succeeded
```

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 :

```
Running with gitlab-runner 13.9.0-rc2 (69c049fd)
on docker-auto-scale 0277ea0f
feature flags: FF_GITLAB_REGISTRY_HELPER_IMAGE:true
Preparing the "docker+machine" executor
00:34
Using Docker executor with image ruby:2.7.2 ...
Pulling docker image ruby:2.7.2 ...
```

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 :

```
Running with gitlab-runner 13.9.0-rc2 (69c049fd)
on docker-auto-scale fa6cab46
feature flags: FF_GITLAB_REGISTRY_HELPER_IMAGE:true
Preparing the "docker+machine" executor
Using Docker executor with image alpine:3.12 ...
```

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 tâche `build-job` pour stocker le fichier `test.txt` et le conserver pendant une semaine :

```
default:
  image: ruby:2.7.2

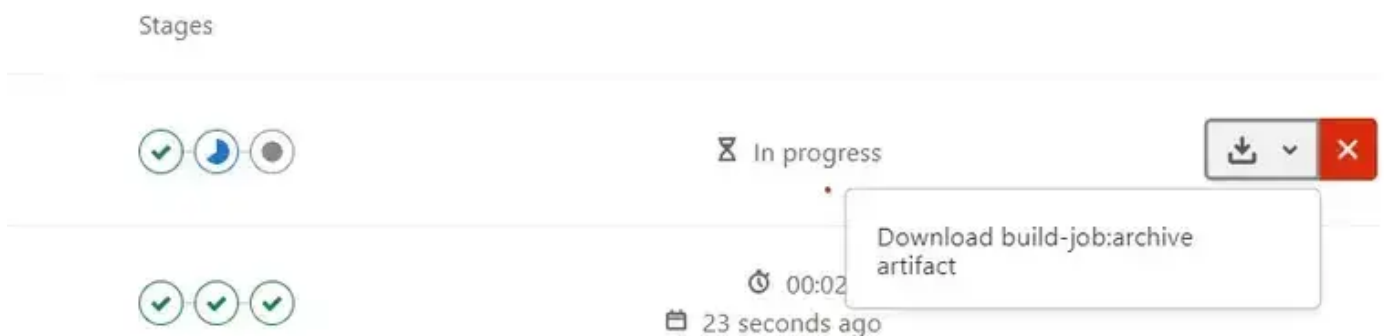
build-job:
  image: alpine:3.12
  stage: build
  variables:
    test: "je suis un test"
  script:
    - echo "$test" > test.txt
  artifacts:
    paths:
      - test.txt
    expire_in: 1 week

...
```

Ce qui donne :

```
Uploading artifacts for successful job
Uploading artifacts...
test.txt: found 1 matching files and directories
Uploading artifacts as "archive" to coordinator... ok id=1106004226
responseStatus=201 Created token=bezhCc6-
Cleaning up file based variables
00:01
Job succeeded
```

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.

# Les Variables et Secrets



# GitLab

**Une bonne gestion des variables et des secrets est essentielle pour sécuriser les pipelines de déploiement, en particulier dans des environnements comme GitLab CI/CD. Ces éléments jouent un rôle important dans la configuration des tâches d'intégration et de livraison continues, impactant directement la performance et la sécurité des applications.**

## Quelques Concepts et Rappels Importants

Les variables dans GitLab CI/CD peuvent être des chaînes de caractères ou des fichiers utilisés pour stocker des données pouvant varier entre les jobs ou les pipelines. Elles incluent des informations comme des chemins de fichiers, des noms d'environnement ou des configurations spécifiques. Leur flexibilité permet d'adapter les pipelines aux différents environnements, tels que la production, le développement ou les tests, sans modifier le code source.

Les secrets, quant à eux, sont des types spéciaux de variables destinés à stocker des informations sensibles. Il peut s'agir de mots de passe, de clés API, ou de certificats SSL. La principale préoccupation avec les secrets est leur sécurité. Une exposition accidentelle peut entraîner des risques de sécurité majeurs, notamment des fuites de données ou des intrusions non autorisées.

GitLab offre plusieurs moyens de déclarer ces variables et secrets. Par exemple, les variables peuvent être définies au niveau du groupe ou du projet et les secrets peuvent être masqués dans les logs de GitLab pour éviter toute exposition accidentelle.

## Les Différents Types de Variables dans GitLab

GitLab CI/CD propose plusieurs types de variables, chacune adaptée à des besoins spécifiques. Comprendre ces types est essentielle pour une utilisation efficace dans vos pipelines.

## Variables d'Environnement

Les variables d'environnement sont les plus courantes dans GitLab CI/CD. Elles servent à stocker des données qui peuvent être utilisées par les scripts des jobs. Par exemple, vous pouvez définir une variable `DATABASE_URL` pour stocker l'URL de votre base de données. Ces variables sont accessibles dans les scripts de pipeline en utilisant la syntaxe standard des variables d'environnement, comme `$DATABASE_URL` dans un script shell.

## Variables de type Fichier

GitLab permet également de définir des variables sous forme de fichiers. Cela est utile pour les certificats, les fichiers de configuration ou tout autre type de données qui doivent être stockées sous forme de fichiers plutôt que de chaînes de texte. Ces variables sont stockées dans un fichier temporaire et le chemin d'accès au fichier est fourni en tant que variable d'environnement.

## Variables Protégées

Les variables protégées sont des variables d'environnement spéciales qui ne sont accessibles que dans les branches ou les tags protégés. Elles sont utiles pour stocker des données sensibles qui ne doivent être utilisées que dans un environnement de production ou un environnement similaire sécurisé. Par exemple, vous pouvez avoir une clé API qui ne doit être utilisée que lors du déploiement en production.

## Variables Masquées

Les variables masquées sont une fonctionnalité de sécurité qui empêche la valeur de la variable d'être affichée dans les logs de GitLab. C'est particulièrement important pour les secrets, comme les mots de passe ou les clés API. Lorsqu'une variable est masquée, sa valeur est remplacée par des astérisques dans les logs d'exécution de pipeline.

## Exemple d'Utilisation de Variables dans un Pipeline

Voici un exemple simple montrant comment ces variables peuvent être utilisées dans un pipeline GitLab CI/CD :

```
stages:
- test
- deploy

test_job:
stage: test
script:
- echo "Running tests with DATABASE_URL=$DATABASE_URL"

deploy_job:
stage: deploy
script:
- echo "Deploying to production with PROD_API_KEY"
only:
- master
variables:
PROD_API_KEY: $PRODUCTION_API_KEY
```

Dans cet exemple, `DATABASE_URL` est une variable d'environnement standard utilisée pour les tests, tandis que `PROD_API_KEY` est une variable protégée utilisée uniquement pour les déploiements en production.

La compréhension et l'utilisation correctes de ces différents types de variables sont essentielles pour créer des pipelines CI/CD flexibles, sécurisés et efficaces dans GitLab.

# Gestion des Secrets

La gestion des secrets est un aspect vital de la sécurité dans **GitLab CI/CD**. Un "secret" est une information sensible, telle qu'un mot de passe, une clé d'API, ou un certificat, qui doit être manipulée avec une extrême prudence pour éviter toute exposition ou utilisation malveillante.

La sécurisation des secrets est primordiale, car leur compromission peut entraîner des failles de sécurité majeures. Dans un pipeline CI/CD, les secrets sont souvent nécessaires pour accéder à des ressources, effectuer des déploiements, ou intégrer des services tiers.

GitLab fournit plusieurs mécanismes pour stocker de manière sécurisée les secrets. L'un d'entre eux est l'utilisation de variables protégées et masquées, comme mentionné précédemment. Il est également conseillé de limiter l'accès aux secrets aux seuls utilisateurs et processus qui en ont absolument besoin, conformément au principe du moindre privilège.

Lors de l'utilisation des secrets dans les pipelines, il est important de veiller à ce qu'ils ne soient pas exposés dans les logs ou transmis de manière non sécurisée. Voici un exemple de comment un secret peut être utilisé dans un job de pipeline :

```
deploy_job:
  stage: deploy
  script:
    - echo "Deploying application"
    - scp -i $DEPLOY_KEY package.zip user@server:/path/to/deploy
  only:
    - master
  variables:
    DEPLOY_KEY: $PRODUCTION_DEPLOY_KEY
```

Dans cet exemple, `DEPLOY_KEY` est un secret utilisé pour authentifier le processus de déploiement sur un serveur distant. Cette clé est stockée en tant que variable protégée dans GitLab et n'est exposée dans aucun log.

# Définition des variables dans Gitlab

GitLab permet de définir des variables à deux niveaux principaux : au niveau du groupe et au niveau du projet. Chacun a ses avantages et utilisations spécifiques.

## Définition de Variables au Niveau du Groupe

Les variables définies au niveau du groupe sont accessibles à tous les projets sous ce groupe. C'est idéal pour partager des configurations communes ou des secrets entre plusieurs projets.

### Comment Définir des Variables de Groupe

1. **Accédez à la page de votre groupe** : Connectez-vous à GitLab et naviguez jusqu'à la page de votre groupe.
2. **Ouvrez les paramètres CI/CD** : Allez dans `Settings > CI/CD`.
3. **Ajoutez une variable de groupe** : Dans la section `Variables`, cliquez sur `Expand` et utilisez le formulaire pour ajouter une nouvelle variable. Vous pouvez spécifier la clé (nom de la variable), la valeur et si elle est protégée ou masquée.



## Variables

[Collapse](#)

Variables store information that you can use in job scripts. Each project can define a maximum of 8000 variables. [Learn more.](#)

Variables can be accidentally exposed in a job log, or maliciously sent to a third party server. The masked variable feature can help reduce the risk of accidentally exposing variable values, but is not a guaranteed method to prevent malicious users from accessing variables. [How can I make my variables more secure?](#)

Variables can have several attributes. [Learn more.](#)

- **Protected:** Only exposed to protected branches or protected tags.
- **Masked:** Hidden in job logs. Must match masking requirements.
- **Expanded:** Variables with `$` will be treated as the start of a reference to another variable.

CI/CD Variables <span>&lt;/&gt; 0</span>			
<a href="#">Add variable</a>			
↑ Key	Value	Environments	Actions
There are no variables yet.			

## Exemple d'Utilisation

Supposons que vous ayez un token d'authentification utilisé par plusieurs projets pour accéder à un service externe. En le définissant comme une variable de groupe, tous les projets sous ce groupe peuvent l'utiliser sans avoir besoin de le définir individuellement.

# Définition de Variables au Niveau du Projet

Les variables de projet sont spécifiques à un projet donné. Elles sont utilisées pour stocker des configurations ou des secrets qui ne sont pertinents que pour ce projet.

## Comment Définir des Variables de Projet

1. **Accédez à votre projet** : Connectez-vous à GitLab et naviguez jusqu'à la page de votre projet.
2. **Ouvrez les paramètres CI/CD** : Allez dans [Settings > CI/CD](#).
3. **Ajoutez une variable de projet** : Dans la section [Variables](#), cliquez sur [Expand](#) et ajoutez votre variable. Vous avez également les options de la protéger ou de la masquer.

## Exemple d'Utilisation

Si votre projet nécessite une clé API spécifique pour se connecter à une base de données, vous pouvez la définir comme une variable de projet. Elle sera isolée de tout autre projet et ne sera disponible que pour les pipelines de ce projet spécifique.

# Bonnes Pratiques

- Utilisez des variables de groupe pour des configurations communes à plusieurs projets, comme des informations d'authentification pour des outils ou services partagés.
- Privilégiez les variables de projet pour des informations spécifiques à un seul projet, afin de renforcer la sécurité et la clarté.

- Utilisez des noms de variables clairs et descriptifs pour faciliter la compréhension et la maintenance des pipelines.

# Précédence des variables

*La notion de “précédence des variables” fait référence à la hiérarchie selon laquelle les variables sont recherchées et utilisées lorsque plusieurs variables du même nom sont définies à différents niveaux.*

Dans **GitLab CI/CD**, les variables peuvent être définies à différents niveaux, et leur priorité varie en fonction de l'endroit où elles sont déclarées. Voici les niveaux de priorité des variables dans **GitLab CI/CD**, du plus élevé au plus bas :

1. **Variables de projet** : Les variables de projet sont définies au niveau du projet GitLab. Elles ont la priorité la plus élevée et sont accessibles à tous les pipelines et jobs du projet. Vous pouvez les définir dans les paramètres du projet via l'interface Web GitLab.
2. **Variables de groupe** : Les variables de groupe sont définies au niveau du groupe GitLab qui contient le projet. Elles ont une priorité inférieure aux variables de projet. Si une variable de projet a le même nom qu'une variable de groupe, la variable de projet a la priorité.
3. **Variables de CI/CD pipeline (variables d'environnement)** : Les variables d'environnement sont définies au niveau du pipeline **GitLab CI/CD** dans le fichier `.gitlab-ci.yml`. Elles ont une priorité inférieure aux variables de projet et de groupe. Vous pouvez les définir directement dans votre fichier de configuration de pipeline et elles seront disponibles uniquement pour ce pipeline spécifique.

Voici comment la priorité des variables fonctionne en pratique :

- Si une variable est définie à la fois au niveau du projet et du groupe avec le même nom, la variable de projet a la priorité et sera utilisée dans les pipelines du projet.
- Si une variable est définie au niveau du groupe, mais pas au niveau du projet, elle sera utilisée dans les pipelines de tous les projets appartenant à ce groupe, sauf si un projet définit sa propre variable avec le même nom.
- Si une variable d'environnement est définie dans le fichier `.gitlab-ci.yml` pour un pipeline spécifique, elle a la priorité pour ce pipeline, même si une variable du même nom est définie au niveau du projet ou du groupe.

Il est important de comprendre cette hiérarchie des variables dans **Gitlab CI/CD**, car elle vous permet de gérer efficacement les secrets et les configurations spécifiques à différents niveaux de votre organisation GitLab. Vous pouvez utiliser cette hiérarchie pour contrôler l'accès aux informations sensibles, tout en permettant la personnalisation des pipelines au niveau du projet.

# Bonnes Pratiques pour les Variables et Secrets

Dans ce chapitre, je mets l'accent sur les meilleures pratiques pour gérer les variables et les secrets dans **GitLab CI/CD**. L'adoption de ces pratiques assure non seulement la sécurité, mais aussi l'efficacité et la fiabilité des pipelines.

## Principe du Moindre Privilège

Appliquer le principe du moindre privilège est essentiel dans la gestion des variables et des secrets. Cela signifie limiter l'accès aux informations sensibles uniquement aux utilisateurs et aux processus qui en ont strictement besoin pour accomplir une tâche spécifique. Par exemple, un secret utilisé pour le déploiement ne devrait pas être accessible lors de l'exécution des tests.

## Rotation des Secrets

La rotation régulière des secrets est une pratique de sécurité essentielle. Elle implique de changer périodiquement les mots de passe, les clés API et autres informations sensibles. Cette approche réduit le risque d'attaques en cas de fuite de secret.

## Gestion Centralisée des Secrets

Utiliser une solution centralisée pour gérer les secrets peut grandement améliorer la sécurité. Des outils comme HashiCorp Vault, Infisical, Passbolt, Sops... ou les services de gestion des secrets cloud (AWS Secrets Manager, Azure Key Vault, etc.) offrent des fonctionnalités avancées telles que le chiffrement, le contrôle d'accès et la rotation automatique des secrets.

## Audit et Journalisation

Mettre en place un système d'audit et de journalisation pour suivre l'utilisation des variables et des secrets. Cela aide à détecter rapidement toute activité anormale ou non autorisée, ce qui est essentiel pour une réponse rapide en cas de compromission.

## Environnements de Variables Séparés

Il est recommandé de séparer les variables en fonction des environnements (par exemple, développement, test, production). Cela aide à éviter les erreurs communes telles que l'utilisation accidentelle de données de production dans des environnements de test.

```
stages:
- test
- deploy

test_job:
stage: test
script:
- echo "Running tests"
variables:
DATABASE_URL: $DEV_DATABASE_URL

deploy_job:
stage: deploy
script:
- echo "Deploying to production"
only:
- master
variables:
DATABASE_URL: $PROD_DATABASE_URL
DEPLOY_KEY: $PRODUCTION_DEPLOY_KEY
```

Dans cet exemple, `DATABASE_URL` est définie séparément pour les environnements de test et de production et `DEPLOY_KEY` est une variable protégée utilisée uniquement dans l'environnement de production.

## Conclusion

En suivant ces pratiques, vous pouvez non seulement améliorer la sécurité de vos pipelines CI/CD, mais aussi leur fiabilité et efficacité. La gestion des variables et des secrets ne doit pas être prise à la légère, étant donné leur impact direct sur la sécurité et la performance des processus de déploiement automatique.

# Maitrisez les templates



# GitLab

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'ancree 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 templates 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 terminée, 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.

# Les Environnements



# GitLab

Dans ce guide, je vais vous guider à travers la configuration et l'utilisation des environnements dans GitLab CI/CD, en mettant l'accent sur les meilleures pratiques et les stratégies avancées.

## Comprendre les environnements Gitlab CI/CD

Un environnement dans le contexte de GitLab CI/CD et plus largement dans le développement logiciel, désigne un contexte ou un cadre dans lequel une application ou un logiciel est exécuté. Cet environnement comprend les serveurs sur lesquels l'application est déployé et s'exécute, les logiciels tiers qu'elle utilise, les variables d'environnement, la configuration réseau et d'autres composants logiciels et matériels. Le concept d'environnement est central en et dans les pratiques DevOps, où différentes phases du développement et du déploiement nécessitent des contextes distincts.

## Caractéristiques Principales d'un Environnement

**Isolation** : Chaque environnement est isolé des autres. Cela signifie que les changements effectués dans un environnement, comme le développement, n'affectent pas directement les autres, comme la production.

**Configuration Spécifique** : Un environnement peut avoir sa propre configuration, incluant des versions de logiciels spécifiques, des paramètres de base de données, des variables d'environnement et des politiques de sécurité adaptées à son rôle.

**Rôle Dédié** : Les environnements sont généralement dédiés à des tâches spécifiques. Par exemple, un environnement de développement est utilisé pour écrire et tester le code, un

environnement de test pour les tests rigoureux et un environnement de production pour l'exécution du logiciel en conditions réelles.

## Exemples d'Environnements

**Développement** : Utilisé par les développeurs pour écrire et tester le code de manière initiale. Il est souvent configuré sur les machines locales des développeurs.


**Test** : Un environnement qui imite de près la production, mais est utilisé pour tester le code dans des conditions contrôlées avant qu'il ne soit déployé en production.

**Staging** : Un environnement de pré-production qui sert de dernière étape de test. Il est conçu pour être aussi proche que possible de l'environnement de production.

**Production** : L'environnement où l'application est finalement déployée et accessible aux utilisateurs finaux. Il est optimisé pour la performance, la sécurité et la stabilité.

## Les environnements dans Gitlab CI/CD

Dans GitLab CI/CD, ces environnements sont définis et gérés via le fichier `.gitlab-ci.yml`, permettant aux développeurs de déployer automatiquement leur application dans différents environnements. Cela aide à assurer que les applications sont développées, testées et déployées de manière systématique, automatisée et contrôlée.



Environment	Deployment	Job	Commit	Updated	Upcoming	Auto stop in
staging	#15 by	deploy_staging #106...	master -> 04ba6a44 Update .gitlab-ci.yml	1 day ago		
production	#14 by	deploy_prod #105931...	master -> 04ba6a44 Update .gitlab-ci.yml	1 day ago		

## Les différents types d'environnement dans Gitlab CI/CD

Il existe deux types d'environnements :

- Les **environnements statiques** ont des noms statiques comme `staging`, `testing`, `development` ou encore `production`.
- Les **environnements dynamiques** ont des noms dynamiques utilisant des variables CI/CD

# Les Environnements Statiques

Les environnements statiques sont des environnements prédéfinis dans GitLab CI/CD. Ils sont généralement constants et représentent des étapes standard du cycle de vie d'une application, comme les environnements de **développement**, **test** et **production**. Ces environnements sont définis une fois et réutilisés tout au long du processus de développement.

Il est possible de créer des environnements statiques soit :

- Dans l'interface de gitlab : Operate > Environments > New Environment
- Dans votre fichier `.gitlab-ci.yml` :

```
deploy_staging:  
  stage: deploy  
  script:  
    - echo "Deploy to staging server"  
  environment:  
    name: staging  
    url: https://staging.example.com
```

Ici, `staging` est un environnement statique. Il est défini une seule fois et reste le même quel que soit le nombre de déploiements effectués. L'avantage des environnements statiques réside dans leur simplicité et leur prévisibilité. Ils sont idéaux pour des scénarios de déploiement standardisés et ne nécessitent pas de configuration supplémentaire pour chaque déploiement.

## Environnements Dynamiques

Les environnements dynamiques, en revanche, sont générés à la volée en fonction de certaines conditions ou actions, comme la création d'une nouvelle branche ou d'une merge request. Ils sont extrêmement utiles pour des scénarios tels que le test de nouvelles fonctionnalités, où vous souhaitez créer un environnement de test unique pour chaque branche de fonctionnalité sans affecter l'environnement principal de test.

Voici un exemple de configuration pour un environnement dynamique :

```
deploy_review:  
  stage: deploy  
  script: echo "Déploiement de l'environnement de revue pour $CI_COMMIT_REF_NAME"  
  environment:  
    name: review/$CI_COMMIT_REF_NAME
```

```
url: https://$CI_COMMIT_REF_NAME.example.com
only:
- branches
except:
- main
- develop
```

Dans cet exemple, un nouvel environnement de revue est créé à chaque fois qu’une nouvelle branche est poussée, à l’exception des branches `main` et `develop`. Cela signifie que chaque nouvelle branche aura son propre environnement unique, basé sur le nom de la branche. Ces environnements sont dynamiques, car ils sont créés et détruits dynamiquement en fonction du flux de travail de développement.

En résumé, les environnements statiques sont constants et prévisibles, idéaux pour les déploiements réguliers, tandis que les environnements dynamiques offrent flexibilité et spécificité, adaptés aux besoins de développement et de test en constante évolution.

## Variables CI/CD propre aux environnements

Lorsque vous créez un environnement, vous indiquez son nom et son URL. Si vous souhaitez les utiliser dans vos scripts, sachez qu’ils sont accessibles par des variables spécifiques. Cela peut être particulièrement utiles pour personnaliser et automatiser des tâches en fonction de l’environnement dans lequel le pipeline CI/CD est exécuté. Elles permettent une intégration plus fine et une meilleure adaptabilité des pipelines aux différents environnements de déploiement.

## Variables Propres aux Environnements GitLab

- **CI\_ENVIRONMENT\_NAME**

- **Description** : Cette variable contient le nom de l’environnement pour le job en cours. Par exemple, si vous avez un environnement nommé “Production”, `CI_ENVIRONMENT_NAME` sera égal à “Production” lors de l’exécution d’un job dans cet environnement.
- **Utilisation typique** : Souvent utilisée pour des scripts ou des configurations qui nécessitent de connaître le nom de l’environnement actuel.

- **CI\_ENVIRONMENT\_SLUG**

- **Description** : Il s’agit d’une version simplifiée de `CI_ENVIRONMENT_NAME`, conçue pour être utilisée dans les URL ou les noms de domaine. Elle est automatiquement générée à partir de `CI_ENVIRONMENT_NAME` en remplaçant les caractères spéciaux.
- **Utilisation typique** : Pratique pour créer des sous-domaines ou des chemins basés sur le nom de l’environnement, par exemple, pour des environnements de revue



dynamiques.

- **CI\_ENVIRONMENT\_URL**

- **Description** : Cette variable contient l'URL de l'environnement définie dans le fichier `.gitlab-ci.yml`. Cela permet d'accéder directement à l'environnement depuis l'interface utilisateur de GitLab.
- **Utilisation typique** : Utile pour fournir un accès rapide à l'environnement depuis les rapports de pipeline, les merge requests ou les notifications.

Il est possible de surcharger la variable `$CI_ENVIRONMENT_NAME` mais la variable `$CI_ENVIRONMENT_SLUG` restera inchangé pour éviter les effets de bords.

## Restreindre un environnement à une ou des branches spécifiques

Avec l'ajout de règles gitlab-ci, il est possible de restreindre des environnements à des branches définies.

Par exemple si nous souhaitons déployer sur l'environnement `review/$CI_COMMIT_REF_NAME` ci-dessus toutes les branches à l'exception de la branche master :

```
deploy_review:
  stage: deploy
  script:
    - echo "Deploy a review app on $CI_ENVIRONMENT_SLUG"
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://$CI_ENVIRONMENT_SLUG.example.com
  only:
    - branches
  except:
    - master
```

Par contre, si on ne souhaite déployer que sur l'environnement de production que la branche master :

```
deploy_prod:
  stage: deploy
  script:
```

```
- echo "Deploy on prod"
environment:
  name: production
  url: https://www.example.com
  only:
    - master
  when: manual
```

Vous pouvez aussi ajouter la condition `when: manual` pour ne pas déclencher automatiquement ce déploiement en production.

# Gestion des environnements

## Arrêt d'un environnement

Il est possible de stopper un environnement, mais pour cela, il faut que dans votre CI contienne une étiquette `on_stop: nom de l'étape` et une `action: stop` associée :

```
deploy_review:
  stage: deploy
  script:
    - echo "Deploy a review app"
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://$CI_ENVIRONMENT_SLUG.example.com
    on_stop: stop_review
  rules:
    - if: $CI_MERGE_REQUEST_ID

stop_review:
  stage: deploy
  script:
    - echo "Remove review app"
  environment:
    name: review/$CI_COMMIT_REF_NAME
  action: stop
```

```
rules:
- if: $CI_MERGE_REQUEST_ID
when: manual
```

Il est également possible de stopper un environnement au bout d'un certain temps via l'ajout de l'étiquette `auto_stop_in: 1 week` :

```
review_app:
script: deploy-review-app
environment:
name: review/$CI_COMMIT_REF_NAME
on_stop: stop_review_app
auto_stop_in: 1 week
rules:
- if: $CI_MERGE_REQUEST_ID

stop_review_app:
script: stop-review-app
environment:
name: review/$CI_COMMIT_REF_NAME
action: stop
rules:
- if: $CI_MERGE_REQUEST_ID
when: manual
```

Si vous souhaitez que cette durée soit prolongée, il faut se rendre dans l'interface de gitlab : Operate > Environnements et d'épingler celui-ci.

GitLab.org > GitLab > Environments > review/16950-add-marker-ranges

review/16950-add-marker-ranges Auto stops in 1 day



View deployment

Status	ID	Triggerer	Commit	Job	Created
success	#372316		Y refs/merge-... 47aee922 Merge branch '16950_add_marker...	review-deploy...	1 hour ago

## Rollback d'un environnement

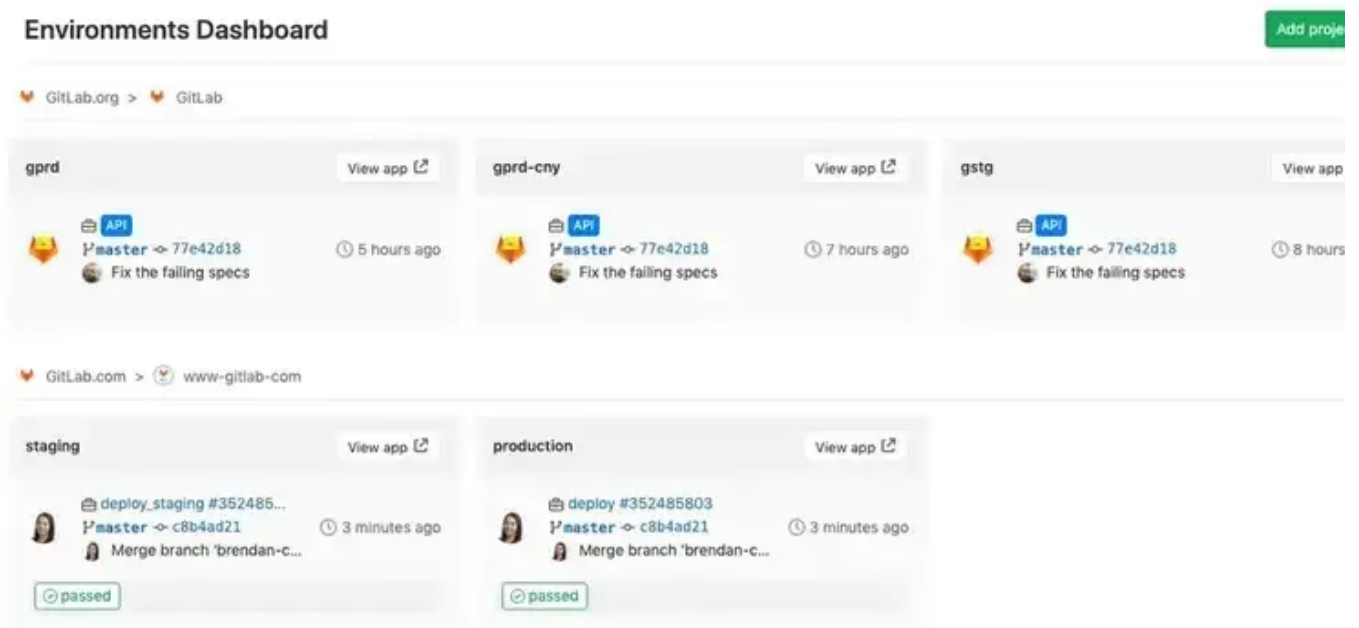
Comme dit plus haut, il est possible de faire un rollback d'environnement. Pour que cela fonctionne correctement, il faut que vos scripts le gère correctement.



# Surveillance des Environnements

GitLab fournit des outils intégrés pour surveiller l'état et la performance des environnements. Cela inclut la surveillance de la santé des applications, le suivi des déploiements et la détection précoce des incidents.

1. **Tableaux de Bord d'Environnements** : GitLab propose des tableaux de bord spécifiques pour chaque environnement. Ces tableaux de bord affichent des informations en temps réel sur l'état des déploiements, les performances des applications et d'autres métriques clés. Ils sont essentiels pour avoir une vue d'ensemble rapide de la santé de vos environnements.



1. **Alertes et Notifications** : Configurer des alertes et des notifications est indispensable pour une réponse rapide en cas d'incidents. GitLab permet de configurer des alertes personnalisées qui peuvent être envoyées via email, Slack ou d'autres canaux de communication en cas de détection d'anomalies.

## Conclusion

En parcourant les divers aspects des environnements CI/CD de GitLab, nous avons vu comment une compréhension approfondie et une utilisation efficace de ces outils peuvent transformer les processus de développement et de déploiement. De la configuration des environnements jusqu'à la surveillance et la gestion rigoureuse, GitLab CI/CD offre une plateforme robuste et flexible pour répondre aux défis complexes du DevOps moderne.

En conclusion, maîtriser les environnements GitLab CI/CD est un élément clé pour tout consultant DevOps cherchant à améliorer l'efficacité, la rapidité et la fiabilité de ses processus de déploiement. En exploitant pleinement les capacités de GitLab, vous vous assurez de répondre efficacement aux besoins en constante évolution de vos projets et de vos équipes.

# Les Conditions

## Pourquoi les Conditions sont-elles Essentielles ?

Les pipelines GitLab CI/CD sont des workflows automatisés qui permettent d'automatiser des tâches telles que la construction, les tests, le déploiement et bien d'autres. Cependant, toutes les étapes d'un pipeline ne doivent pas nécessairement être exécutées à chaque commit ou à chaque modification du code source. C'est là que les conditions entrent en jeu.

Les conditions définissent les règles qui déterminent si une étape du pipeline doit être exécutée ou non. Elles permettent d'optimiser le processus de développement en exécutant uniquement les étapes nécessaires, ce qui économise du temps et donc des ressources.

## Définir des Conditions

Pour définir des conditions sur un job du pipeline, il suffit d'y ajouter une section `rules`. Chaque règle (rule) spécifie une condition sous la forme d'une expression conditionnelle et le job sera exécuté si cette condition est évaluée comme vraie.

```
job:
  script:
    - echo "Cette étape s'exécute si la condition est satisfaite"
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
```

Dans cet exemple, le job sera exécuté uniquement si la branche du commit est "main". Cependant, les possibilités sont vastes et vous pouvez créer des conditions complexes en utilisant des opérateurs logiques et des variables d'environnement.

Voyons quelques exemples d'utilisation courants de conditions :

```
rules:
  - if: '$CI_COMMIT_TAG'
```

Ici, le job sera exécuté uniquement si le commit est associé à un tag.

```
rules:  
- if: '$CI_PIPELINE_SOURCE == "manual"'
```

Ce job s'exécutera seulement si le pipeline a été déclenché manuellement.

## Personnalisation des Conditions

La flexibilité de `rules` vous permet de personnaliser les conditions selon vos besoins spécifiques. Vous pouvez combiner plusieurs règles, utiliser des opérateurs logiques (comme `&&` et `||`) et accéder à un large éventail de variables d'environnement pour créer des conditions complexes.

## Utilisation de `when` pour Contrôler le Déclenchement

`when` est utilisé à l'intérieur des règles (`rules`) pour définir le moment où un job doit être exécuté. GitLab offre plusieurs options pour `when` afin de personnaliser le déclenchement de vos jobs en fonction de scénarios spécifiques.

### `on_success`

```
rules:  
- if: '$CI_COMMIT_BRANCH == "main"'  
  when: on_success
```

Dans cet exemple, le job sera exécuté uniquement si la branche du commit est "main" et si tous les jobs précédents dans le pipeline ont été exécutés avec succès.

### `on_failure`

```
rules:  
- if: '$CI_COMMIT_BRANCH == "main"'  
  when: on_failure
```

Ce job s'exécutera uniquement si la branche du commit est "main" et si au moins l'un des jobs précédents dans le pipeline a échoué.

## always

```
rules:  
- if: '$CI_COMMIT_BRANCH == "main"'  
when: always
```

Ici, le job sera toujours exécuté, quelle que soit la réussite ou l'échec des jobs précédents, tant que la branche du commit est "main".

## manual

```
rules:  
- if: '$CI_COMMIT_BRANCH == "main"'  
when: manual
```

Dans cet exemple, le job ne s'exécute que lorsque l'utilisateur le déclenche manuellement, indépendamment des autres conditions.

## delayed

```
rules:  
- if: '$CI_COMMIT_BRANCH == "main"'  
when: delayed  
start_in: 30 minutes
```

Ici, le job s'exécute automatiquement, mais il est retardé de 30 minutes à partir du déclenchement du pipeline.

## scheduled



```
rules:
- if: '$CI_COMMIT_BRANCH == "main"'
when: scheduled
cron: "0 12 * * *"
```

Dans cet exemple, le job est planifié pour s'exécuter tous les jours à midi.

# Les Conditions sur des Fichiers

Dans ce chapitre, nous allons voir comment utiliser `change` et `exist` pour gérer conditions des jobs sur la présence ou non de certains fichiers.

## change

La condition `change` vous permet de spécifier des fichiers ou des répertoires qui, lorsqu'ils sont modifiés dans un commit, autorise l'exécution d'un job. Cela peut être utile pour des tâches telles que la compilation du code uniquement lorsqu'un fichier source est modifié.

Voici un exemple :

```
job:
script:
- echo "Cette étape s'exécute si le fichier 'config.yaml' est modifié"
rules:
- changes:
- config.yaml
```

Dans cet exemple, le job sera exécuté uniquement si le fichier `config.yaml` est modifié dans le commit actuel.

## exist

La condition `exist` vous permet de vérifier l'existence d'un fichier ou d'un répertoire dans le référentiel. Si le fichier ou le répertoire existe, le job est exécuté.

Voici un exemple :

```
rules:
- exists:
- data.csv
```

Dans cet exemple, le job sera exécuté uniquement si le fichier `data.csv` existe dans le référentiel.

## Utilisation de `change` et `exist` en //

Vous pouvez également combiner `change` et `exist` pour des conditions plus complexes. Par exemple, vous pourriez vouloir exécuter un job uniquement si un fichier spécifique est modifié et qu'un autre fichier existe.

```
rules:
- changes:
- config.yaml
- exists:
- data.csv
```

Dans cet exemple, le job ne s'exécute que si `config.yaml` est modifié dans le commit actuel et que `data.csv` existe dans le référentiel.

## Les règles sur les pipelines : `workflow`

La directive `workflow` détermine si un pipeline doit être créé et exécuté. Contrairement aux règles appliquées à des jobs individuels, `workflow` permet de contrôler le déclenchement de l'ensemble du pipeline basé sur des critères globaux. Cette capacité de gestion à un niveau supérieur offre un contrôle précis et efficace sur l'exécution des pipelines, en particulier dans des projets complexes avec de multiples branches et conditions.

Un cas d'utilisation courant de `workflow` est la mise en place de conditions pour exécuter des pipelines uniquement pour certains événements ou branches. Par exemple, vous pourriez vouloir déclencher un pipeline seulement lors de pushes sur la branche principale ou sur des branches de fonctionnalités, mais pas sur des branches de corrections de bugs. Voici comment cela peut être configuré :

```
workflow:
rules:
```

```
- if: '$CI_PIPELINE_SOURCE == "push" '
- if: '$CI_COMMIT_BRANCH == "main" '
- if: '$CI_COMMIT_BRANCH =~ /^feature-/ '
```

Dans cet exemple, le pipeline est déclenché si le déclencheur est un push sur la branche principale (`main`) ou sur une branche dont le nom commence par `feature-`. Cette configuration permet d'assurer que les ressources ne sont pas gaspillées sur des pipelines non essentiels, tout en garantissant que les branches importantes reçoivent l'attention nécessaire.

`workflow` peut également intégrer des logiques plus complexes, comme des conditions basées sur des variables d'environnement, des tags ou des changements spécifiques dans le code. Cela permet une flexibilité et une personnalisation élevées, adaptant vos pipelines aux besoins spécifiques de votre projet.

## Exemple : Éviter les Exécutions Redondantes de Pipelines

Imaginons un cas où vous souhaitez éviter que le même pipeline soit déclenché à la fois par un push et par un tag sur le même commit. Ceci est un scénario courant où des pipelines redondants peuvent se lancer si les règles ne sont pas bien configurées. Voici comment `workflow` peut être utilisé pour gérer cela :

```
workflow:
  rules:
    - if: '$CI_COMMIT_TAG'
  when: never
    - if: '$CI_PIPELINE_SOURCE == "push" '
```

Dans cet exemple, le pipeline est configuré pour ne pas se déclencher lorsqu'un tag est appliqué (`when: never`). Cela signifie que si un commit est à la fois poussé et tagué, le pipeline ne se déclenchera que pour le push, évitant ainsi un déclenchement redondant pour le tag.

## Exemple : Gérer les Pipelines pour les Merge Requests

Un autre cas d'usage fréquent est la gestion des pipelines pour les merge requests. Vous pourriez vouloir exécuter des pipelines uniquement pour les merge requests, mais pas pour les branches individuelles, afin d'économiser des ressources. Voici un exemple de configuration :

```
workflow:
  rules:
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
    - if: '$CI_COMMIT_BRANCH && $CI_OPEN_MERGE_REQUESTS'
  when: never
```

Dans cette configuration, le pipeline se déclenche uniquement pour les événements de merge requests. Si un commit est poussé sur une branche qui a une merge request ouverte, le pipeline ne se déclenchera pas séparément pour cette branche, évitant ainsi des exécutions en double.

Ces exemples illustrent comment la directive `workflow` peut être utilisée pour affiner la logique de déclenchement de vos pipelines dans GitLab CI/CD, vous permettant de gérer efficacement les ressources et d'éviter des exécutions inutiles ou redondantes. En personnalisant ces règles selon les besoins spécifiques de votre projet, vous pouvez optimiser vos processus de CI/CD pour une efficacité et une performance maximales.

## Combinaison Stratégique de `rules` et `workflow`

L'intégration et l'orchestration efficaces des pipelines dans GitLab CI/CD peuvent être grandement améliorées en combinant judicieusement les `rules` au niveau des jobs avec la directive `workflow` pour le pipeline global. Cette combinaison permet une gestion fine et une optimisation des déclenchements et exécutions des jobs, adaptant ainsi les pipelines aux exigences spécifiques et aux conditions de votre projet.

## Exemple : Gestion de Pipelines en Fonction des Branches et des Événements

Imaginez un scénario où vous souhaitez exécuter des jobs différents en fonction de la branche sur laquelle les commits sont poussés, tout en gérant globalement le déclenchement des pipelines pour éviter les redondances. Voici comment cela peut être configuré :

```
workflow:
  rules:
    - if: '$CI_PIPELINE_SOURCE == "push" && $CI_COMMIT_BRANCH == "main"'
    - if: '$CI_PIPELINE_SOURCE == "schedule"'
```

```
- if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
deploy:
script: deploy.sh
rules:
- if: '$CI_COMMIT_BRANCH == "main"'
when: manual
- if: '$CI_COMMIT_BRANCH =~ /^release-/'
when: on_success

test:
script: test.sh
rules:
- if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
```

Ici, le job `deploy` est configuré pour s'exécuter manuellement sur la branche principale et automatiquement en cas de succès sur les branches commençant par `release-`. Le job `test` est quant à lui déclenché uniquement pour les merge requests.

# Avantages de cette Combinaison

En combinant `workflow` et `rules`, vous pouvez :

- **Optimiser les Ressources** : Réduire les exécutions inutiles ou redondantes de pipelines, économisant ainsi des ressources.
- **Améliorer la Clarté** : Clarifier la logique de déclenchement des pipelines, rendant vos configurations plus lisibles et maintenables.
- **Personnaliser les Pipelines** : Adapter les pipelines aux besoins spécifiques de différentes branches ou types d'événements dans votre projet.

La flexibilité et la puissance de cette combinaison offrent un contrôle approfondi et une personnalisation poussée de vos pipelines GitLab CI/CD. En maîtrisant ces outils, vous pouvez construire des systèmes de CI/CD qui non seulement répondent aux besoins spécifiques de votre projet, mais le font également de manière efficace et optimisée.

# Conclusion

En comprenant et en appliquant ces concepts dans vos projets, vous pourrez assurer que vos pipelines ne sont pas seulement fonctionnels, mais qu'ils sont également conçus de manière à répondre efficacement aux changements et exigences de votre environnement de développement.

Cela conduira à une intégration et une livraison continues plus robustes, fiables et efficaces, un objectif clé dans toute démarche DevOps.

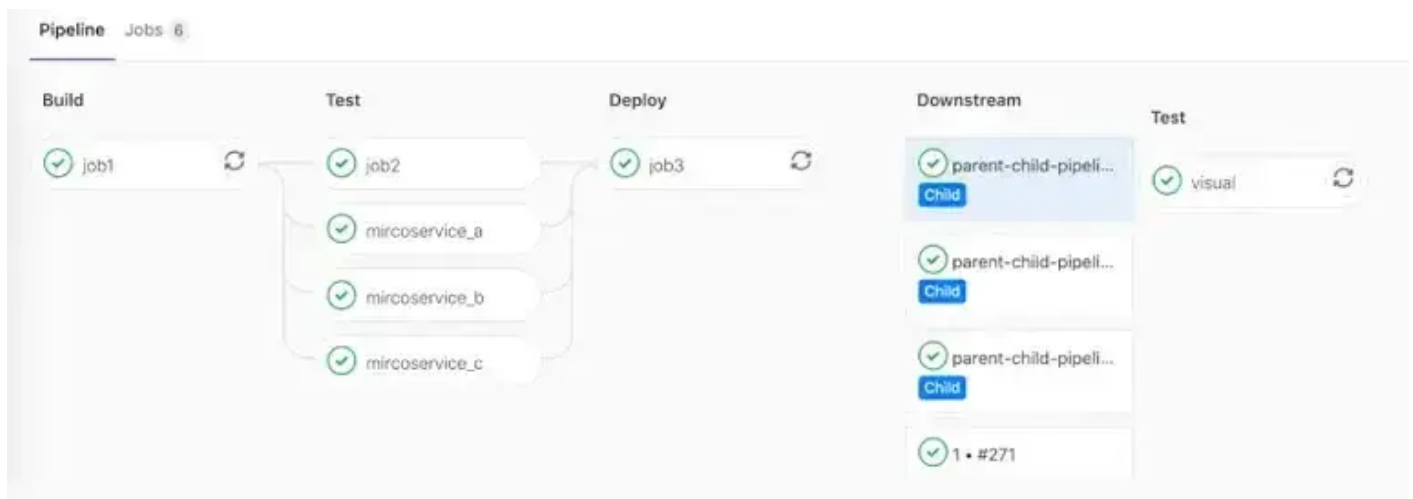
Je vous encourage à expérimenter ces règles dans vos propres pipelines GitLab CI/CD, à découvrir leurs potentiels et à les adapter pour répondre au mieux aux besoins de vos projets. La maîtrise de ce concept est un pas essentiel vers l'excellence en matière de CI/CD, vous permettant de pousser vos capacités de développement et de déploiement vers de nouveaux sommets d'efficacité et de performance.

# Les pipelines Parent/Enfant



# GitLab

GitLab CI/CD depuis la version 12.7 a apporté le concept de **pipelines parent-child** (Parent-enfant) qui permet de décomposer des pipelines.



## Comprendre les Pipelines Parent-Enfant

Avec les **pipelines parent-enfant**, un pipeline principal, ou “parent”, peut déclencher un ou plusieurs pipelines “enfants”. Cette hiérarchisation permet une meilleure organisation, une séparation claire des tâches et une plus grande modularité. Par exemple, un pipeline parent peut être responsable de la construction générale et des tests, tandis que les pipelines enfants peuvent être dédiés à des déploiements spécifiques ou à des tests plus approfondis.

Le principal avantage de cette approche est la flexibilité. En séparant les différentes étapes du processus CI/CD en pipelines distincts, on peut mieux gérer les dépendances, les ressources et les autorisations. Cela permet également de réutiliser des configurations de pipeline dans différents projets, améliorant ainsi l'efficacité et réduisant les redondances.

En outre, les **pipelines parent-enfant** favorisent un flux de travail plus propre et plus organisé. Plutôt que d'avoir un pipeline monolithique avec de nombreuses étapes, ce modèle permet de diviser les tâches en unités plus petites et plus gérables. Cela se traduit par une meilleure visibilité sur chaque étape du processus et une détection plus rapide des problèmes potentiels.

## La Directive Trigger de Gitlab CI/CD

La directive `trigger` est utilisée dans le fichier de configuration `.gitlab-ci.yml` d'un projet GitLab. Elle spécifie qu'une tâche dans le pipeline parent doit déclencher un autre pipeline, généralement dans un projet différent. Cela crée une relation hiérarchique où le pipeline parent orchestre le déroulement des pipelines enfants selon les besoins du processus de CI/CD.

Pour configurer cette directive, vous devez spécifier un chemin vers un fichier ou un projet cible. Voici un exemple simple avec un projet :

```
deploy_to_production:
  stage: deploy
  trigger:
    project: my-group/my-production-project
    branch: master
```

Dans cet exemple, la tâche `deploy_to_production` dans le pipeline parent va déclencher un pipeline dans le projet `my-group/my-production-project` sur la branche `master`.

Un autre aspect important de la directive `trigger` est la possibilité de passer des variables d'un pipeline parent à un pipeline enfant. Cela permet une personnalisation fine des pipelines enfants en fonction des résultats ou des paramètres du pipeline parent. Par exemple, vous pourriez passer une variable indiquant le succès des tests dans le pipeline parent pour conditionner le déroulement du pipeline enfant.

La directive `trigger` offre également des options avancées, comme le déclenchement conditionnel des pipelines enfants, basé sur certains critères ou résultats du pipeline parent. Cela ajoute une couche supplémentaire de contrôle et d'efficacité, permettant une gestion plus fine des processus de CI/CD.

## Exemple dans un seul projet

On peut utiliser ce concept dans un seul projet. L'écriture des pipelines enfants est identique à celui qu'on utilise classiquement. Par contre, pour celui du parent, il faudra utiliser la clé `trigger` pour lancer son exécution.



```
stages:
- builds

build1:
stage: builds
trigger:
include: build1.yml
strategy: depend

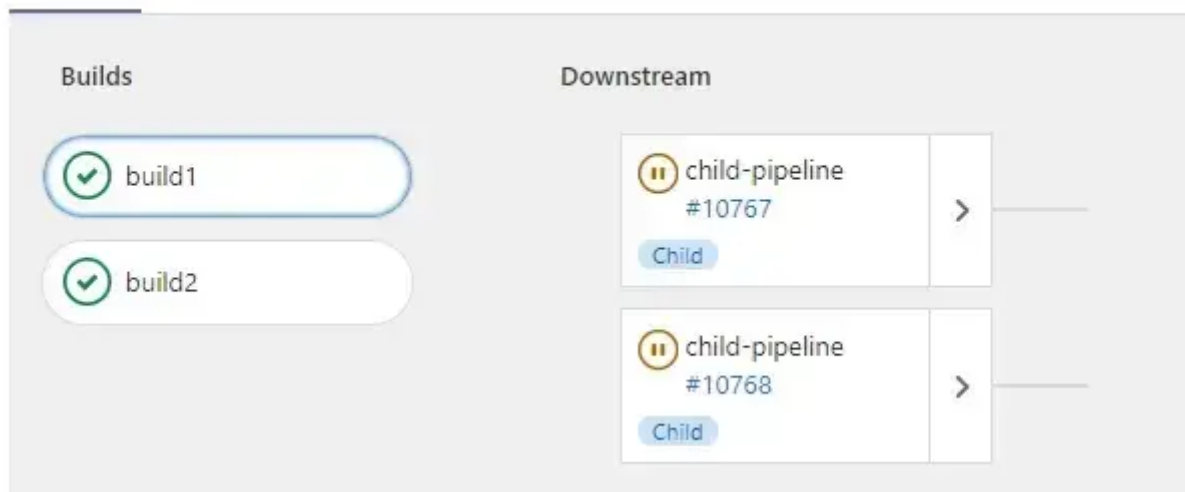
build2:
stage: builds
trigger:
include: build2.yml
strategy: depend
```

Par défaut le pipeline parent se termine tout de suite en status `success` avant même que les enfants ne se soient exécutés. Pour l'obliger à attendre la fin de l'exécution des enfants et de lier son status avec ceux-ci, il faut indiquer une stratégie avec la clé `strategy: depend`

Un des pipelines enfant (l'autre est identique, c'est pour l'exemple) :

```
#build1.yml
build:
stage: build
script:
- touch hello.md
artifacts:
paths:
- hello.md
```

Ce qui donne :



## Exemple en utilisant des templates

Il est possible de lancer des templates de pipelines. Il suffit juste d'indiquer le nom du projet avec la clé `project: stephane.r/test2`

Ce qui donne dans notre exemple :

```
stages:
- builds

build1:
stage: builds
trigger:
include: build1.yml
strategy: depend

build2:
stage: builds
trigger:
include: build2.yml
strategy: depend
```

```
build3:
  stage: builds
  trigger:
  project: stephane.r/test2
  strategy: depend
```

Dans le projet test2, il suffit de créer un fichier `.gitlab-ci.yml`.

Pipeline parent-enfant

## Passer des variables à un enfant

Pour passer des variables aux enfants, il suffit d'utiliser la clé `variables`.

```
build3:
  variables:
  ENVI: staging
  stage: builds
  trigger:
  project: stephane.r/test2
  strategy: depend
```

Dans le ci du projet test2, j'ai modifié le script pour qu'il utilise la variable :

```
build:
  tags:
  - php
  stage: build
  script:
  - echo $ENVI > hello.md
  artifacts:
  paths:
  - hello.md
```

Dans le fichier hello.md, je retrouve bien le contenu de ma variable ENVI.

# Passer des artefacts aux enfants

On peut aussi passer des artefacts, pour cela, il suffit d'ajouter `needs` dans le trigger. Par exemple pour passer des variables du projet test à test2 en utilisant build.env :

```
#test/.gitlab-ci.yml

stages:
- version
- deploy

version:
tags:
- javascript
stage: version
script:
- echo "VERSION=$CI_COMMIT_TAG" >> build.env
- echo "$CI_COMMIT_TAG"
artifacts:
reports:
dotenv: build.env

deploy:
needs:
- build
variables:
VERSION: $VERSION
stage: deploy
trigger:
project: stephane.r/test2
strategy: depend
only:
- tags
```

```
#test2/.gitlab-ci.yml
# build1.yml
```

```
build:
tags:
- javascript
stage: build
script:
- echo "$VERSION"
```

En sortie de script, je récupère bien le tag de test.

```
...
```

```
$ echo "$VERSION"
v0.1
Cleaning up file based variables
00:01
Job succeeded
```

## Cas d'usages

Pour illustrer l'impact et l'efficacité des **pipelines parent-enfant** dans GitLab, examinons quelques cas d'utilisation réels.

## Déploiements Multi-Environnements

En utilisant les **pipelines parent-enfant**, on peut créer un pipeline parent qui gère les étapes de construction et de test et des pipelines enfants distincts pour chaque environnement de déploiement. Cela permet de personnaliser les déploiements pour chaque environnement tout en maintenant un processus centralisé et cohérent.

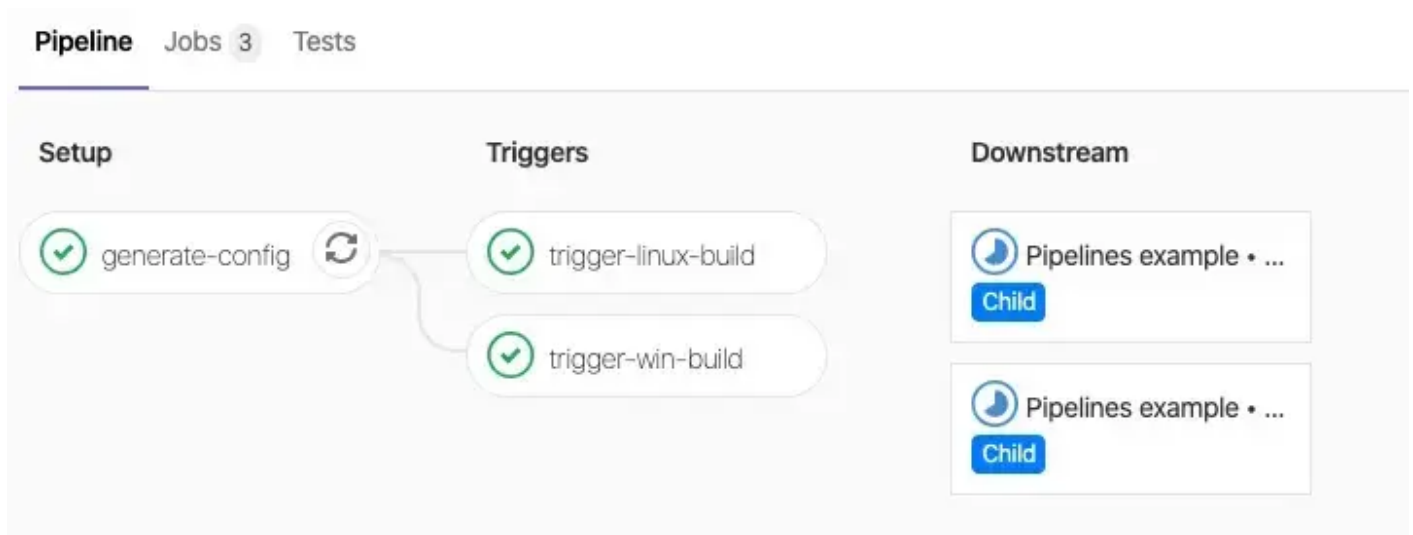
# Les pipelines dynamiques



# GitLab

## 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 efficace. 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>.yaml` 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.yaml` 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.

# Maîtriser la CLI



# GitLab

**Nous connaissons tous l'importance des outils qui optimisent notre travail. Parmi ces outils, la CLI de GitLab CI/CD, anciennement `glab`, est pour moi une des plus importantes.**

Ce qui rend `glab` si important, c'est sa capacité à offrir presque toutes les fonctionnalités de GitLab directement depuis la ligne de commande. Cette approche est idéale pour les utilisateurs qui préfèrent une interface en ligne de commande, plus rapide et plus scriptable, par rapport à l'interface graphique web de GitLab. Avec `glab`, vous pouvez gérer les projets, les issues, les merge requests et bien plus encore.

## Installation et Configuration de `glab`

Pour profiter pleinement de `glab`, la première étape consiste à l'installer et à le configurer correctement sur votre système.

### Installation de `glab`

#### Sur Linux

Si vous êtes un utilisateur de Linux, l'installation de `glab` peut se faire facilement avec asdf-vm :

```
asdf plugin add glab
asdf install glab latest
asdf global glab latest
```

#### Sur Alpine Linux

Idéal pour intégrer la cli de gitlab dans vos Dockerfiles pour vos runs de Pipeline. Cela va vous ouvrir des portes que vous n'aviez pas imaginées :

```
apk add --no-cache glab
```

## Sur macOS

Pour les utilisateurs de macOS, `glab` peut être installé via Homebrew. Ouvrez simplement votre terminal et tapez :

```
brew install glab
```

## Sur Windows

Sous Windows, `glab` peut être installé à l'aide de Scoop ou de Chocolatey. Avec Chocolatey, la commande est la suivante :

```
choco install glab
```

# Les Principales Fonctionnalités de la CLI Gitlab

`glab` offre une suite complète d'options qui permettent une gestion efficace des projets GitLab. Voici une liste des principales commandes disponibles dans `glab`, chacune accompagnée d'une brève description de sa fonction :

1. `alias` : Permet de créer des alias pour les commandes `glab`, simplifiant ainsi l'utilisation des commandes fréquentes.
2. `api` : Offre un accès direct à l'API de GitLab, permettant d'effectuer des requêtes API personnalisées.
3. `auth` : Utilisée pour l'authentification et la configuration de vos identifiants GitLab dans `glab`.
4. `check-update` : Vérifie si une nouvelle version de `glab` est disponible.
5. `ci` : Permet de gérer les pipelines de CI (Continuous Integration), y compris leur création, visualisation et suppression.
6. `completion` : Génère des scripts d'auto-complétion pour différents shells, facilitant la saisie des commandes `glab`.



7. `config` : Utilisée pour configurer les paramètres globaux de `glab`.
8. `incident` : Permet de gérer les incidents, notamment leur création, mise à jour et fermeture.
9. `issue` : Fournit des outils pour gérer les issues, incluant la création, la liste, la mise à jour et la fermeture.
10. `label` : Permet de gérer les labels pour les issues et les merge requests.
11. `mr` : Fournit des commandes pour gérer les merge requests, y compris leur création, liste, mise à jour et fusion.
12. `release` : Permet de gérer les releases, incluant leur création, listage et suppression.
13. `repo` : Offre des outils pour gérer les dépôts GitLab, y compris la création, le clonage et la gestion des branches.
14. `schedule` : Permet de gérer les pipelines planifiés (scheduled pipelines).
15. `snippet` : Fournit des commandes pour créer et gérer des extraits de code (snippets).
16. `ssh-key` : Permet de gérer les clés SSH associées à votre compte GitLab.
17. `user` : Offre des outils pour visualiser et gérer les informations utilisateur sur GitLab.
18. `variable` : Permet de gérer les variables de CI/CD au niveau du projet ou du groupe.

Chacune de ces commandes est conçue pour intégrer GitLab de manière plus profonde dans votre flux de travail DevOps, en rendant la gestion de vos projets, issues, merge requests et CI/CD plus efficace et automatisée.

## Configuration de `glab`

Une fois `glab` installé, la prochaine étape est de le configurer pour se connecter à votre compte GitLab. Cette connexion est essentielle pour permettre à `glab` d'interagir avec vos projets GitLab.

## Les commandes de configuration

### Authentification avec `glab auth`

La commande `glab auth` est utilisée pour l'authentification avec GitLab. Elle permet de configurer vos identifiants pour une intégration transparente avec vos projets GitLab.

#### **Authentification Interactive :**

Tapez la commande suivante :

```
* Logging into gitlab.com
? How would you like to login? Token
```

```
Tip: you can generate a Personal Access Token here https://gitlab.com/-
/profile/personal_access_tokens?scopes=api,write_repository
The minimum required scopes are 'api' and 'write_repository'.
? Paste your authentication token: *****
? Choose default git protocol SSH
* glab config set -h gitlab.com git_protocol ssh
✓ Configured git protocol
* glab config set -h gitlab.com api_protocol https
✓ Configured API protocol
✓ Logged in as Bob74
```

Suivez les instructions pour entrer le token d'accès personnel. Choisissez l'instance GitLab (GitLab.com ou instance auto-gérée).

### Authentification Non-Interactive :

Utilisez la variable `GITLAB_TOKEN` pour une authentification sans interaction, idéale pour les scripts ou les environnements CI/CD.

Pour vérifier que tout est correctement configuré, vous pouvez utiliser la commande suivante :

```
glab auth status 10:54:29

gitlab.com
x gitlab.com: api call failed: GET https://gitlab.com/api/v4/user: 401 {message:
401 Unauthorized}
✓ Git operations for gitlab.com configured to use ssh protocol.
✓ API calls for gitlab.com are made over https protocol
✓ REST API Endpoint: https://gitlab.com/api/v4/
✓ GraphQL Endpoint: https://gitlab.com/api/graphql/
x No token provided
```

Ici un problème d'authentification.

## Gestion des Paramètres Globaux avec `glab config`

`glab config` permet de personnaliser, `set` l'utilisation de `glab` en configurant des paramètres globaux.

- Définissez votre éditeur de texte préféré : `glab config set editor vim`.

- Configurez le navigateur pour ouvrir les liens : `glab config set browser firefox`.

Pour afficher les paramètres, il faut utiliser l'option `get` :

```
glab config get browser
/home/bob/.vscode-
server/bin/0ee08df0cf4527e40edc9aa28f4b5bd38bbff2b2/bin/helpers/browser.sh
```

## Gestion des Clés SSH avec `glab ssh-key`

La commande `glab ssh-key` aide à gérer les clés SSH pour sécuriser vos opérations Git.

- Ajoutez une clé SSH à GitLab : `glab ssh-key add /chemin/de/la/cle.pub -t "Titre de la clé"`.
- Listez les clés SSH associées à votre compte : `glab ssh-key list`.
- Supprimez une clé SSH spécifique : `glab ssh-key delete <id_de_la_clé>`.

## Configuration de l'Auto-Complétion `glab completion`

La commande `glab completion` de la CLI GitLab permet de générer des fichiers de configuration pour activer la complétion automatique (auto-complétion) dans différents shells, facilitant ainsi l'utilisation de la CLI GitLab.

### Bash

Intégrez-le dans votre configuration Bash en ajoutant :

```
source <(glab completion bash)
```

à votre fichier `~/.bashrc`.

### Zsh

Ajoutez ce script à votre fichier de configuration Zsh (`~/.zshrc`) :

```
source <(glab completion -s zsh); compdef _glab glab
```

### Fish

Intégrez-le dans votre configuration Fish en ajoutant :

```
glab completion -s fish > ~/.config/fish/completions/glab.fish
```

# Les variables d'environnement

L'utilisation de `glab` peut être personnalisée à travers diverses variables d'environnement et commandes de configuration.

- `GITLAB_TOKEN` : Évite la nécessité de s'authentifier à chaque requête API.
- `GITLAB_URI` ou `GITLAB_HOST` : Spécifie l'URL du serveur GitLab pour les instances auto-gérées.
  - Exemple : `https://gitlab.example.com` (par défaut `https://gitlab.com`).
- `GITLAB_API_HOST` : Spécifie l'hôte où se trouve l'API GitLab, utile lorsque Git et l'API sont sur des (sous)domaines distincts.
- `GITLAB_REPO` : Définit le dépôt GitLab par défaut pour les commandes acceptant l'option `--repo`.
- `GITLAB_GROUP` : Définit le groupe GitLab par défaut pour lister les merge requests, issues et variables.
- `REMOTE_ALIAS` ou `GIT_REMOTE_URL_VAR` : Variable ou alias de remote git contenant l'URL GitLab.
- `GLAB_CONFIG_DIR` : Répertoire de configuration globale de `glab`.
  - Par défaut : `~/.config/glab-cli/`.
- `VISUAL`, `EDITOR` : Définit l'éditeur de texte pour la rédaction.
- `BROWSER` : Définit le navigateur web pour ouvrir les liens.
- `GLAMOUR_STYLE` : Style de rendu Markdown personnalisé (options : `dark`, `light`, `notty`).
  - `NO_COLOR` : Désactive la coloration ANSI dans la sortie.
- `FORCE_HYPERLINKS` : Force l'affichage de liens hypertextes, même en dehors d'un TTY.

# Les Commandes Générales `glab`

Les commandes générales sont celles qui ne sont pas spécifiques aux repositories. Elles fonctionnent même si elles ne sont pas exécutées depuis un répertoire contenant un projet gitlab.

## Une Nouvelle Version ? `glab check-update`

La commande update permet de vérifier si une nouvelle version de `glab` est disponible.

```
glab check-update  
You are already using the latest version of glab
```

# Gestion des Repositories avec `glab repo`

La commande `glab repo` est essentielle pour gérer les repositories sur GitLab, offrant de nombreuses options pour une gestion efficace.

- `archive`: Pour archiver un repository.
- `clone`: Pour cloner un repository GitLab.
- `contributors`: Pour afficher les contributeurs d'un repository.
- `create`: Pour créer un nouveau repository.
- `delete`: Pour supprimer un repository.
- `fork`: Pour faire un fork d'un repository.
- `list`: Pour lister les repositories disponibles.
- `mirror`: Pour configurer un miroir pour un repository.
- `search`: Pour rechercher des repositories.
- `transfer`: Pour transférer un repository à un autre utilisateur ou groupe.
- `view`: Pour afficher les détails d'un repository.

## Exemples de commande :

- Lister les repositories disponibles :

```
glab repo list
Showing 30 of 57 projects (Page 1 of 2)

mon-site1/docusaurus git@gitlab.com:mon-site1/docusaurus.git
dockerfiles6/images/container-structure-test
git@gitlab.com:dockerfiles6/images/container-st...
Bob74/aws-blog git@gitlab.com:Bob74/aws-blog.git
dockerfiles6/images/demo-cosign git@gitlab.com:dockerfiles6/images/demo-cosign.git
```

- Rechercher des repositories :

```
glab repo search -s docker
Showing results for "docker"
🔍 nexylan/docker/docker On steroids Docker image. 0 stars 0 forks 0 issues updated
about 19 days ago
https://gitlab.com/nexylan/docker/docker
🔍 jitesoft/dockerfiles/docker Docker image with docker and 0 stars 0 forks 0 issues
updated about 2 months ago
```

```
docker in docker! https://www.docker.com/
https://gitlab.com/jitesoft/dockerfiles/docker
```

En vrac :

```
# Create
glab repo create --group glab-cli
glab repo create my-project
# Un projet privé dans un groupe (existant) avec initialisation du README
glab repo create glab-cli/my-project -d 'ma description' --private --readme -t
ansible,devops

# Review
glab repo view my-project
glab repo view user/repo
glab repo view group/namespace/repo
```

## Les requêtes d'API avec `glab api`

La commande `glab api` permet de lancer des requêtes à l'API GitLab.

### Requetes Standards

La requête `glab api` permet de lancer des call sur l'API de Gitlab. Vous pouvez retrouver les endpoints d'API sur la [documentation](#)

```
glab api /projects/:id/repository/branches/

[
  {
    "name": "main",
    "commit": {
      "id": "011677bac1fb93bac5cea563c0697287c6c926f8",
      "short_id": "011677ba",
      "created_at": "2023-02-05T10:41:25.000+01:00",
```

```
"parent_ids": ["85c5234352df02cad639787560c4137ec75b5a61"],
"title": "Initial commit",
"message": "Initial commit\n",
"author_name": "Robert Stéphane",
"author_email": "robert.stephane.28@gmail.com",
"authored_date": "2023-02-05T10:41:25.000+01:00",
"committer_name": "Robert Stéphane",
"committer_email": "robert.stephane.28@gmail.com",
"committed_date": "2023-02-05T10:41:25.000+01:00",
"trailers": {},
"extended_trailers": {},
"web_url": "https://gitlab.com/Bob74/ansible-gendoc/-
/commit/011677bac1fb93bac5cea563c0697287c6c926f8"
},
"merged": false,
"protected": true,
"developers_can_push": false,
...
```

## Requêtes GraphQL

`glab api` supporte l'utilisation de GraphQL, un langage de requête puissant pour les APIs. Avec l'option GraphQL, vous pouvez :

- Créer des requêtes complexes et précises.
- Obtenir des données spécifiques en une seule requête.
- Manipuler et accéder aux données de manière plus flexible et efficace.

Voici comment vous pourriez utiliser `glab api` pour effectuer une requête GraphQL :

```
glab api graphql -f query='
query {
  project(fullPath: "gitlab-org/gitlab-docs") {
    name
    forksCount
    statistics {
      wikiSize
    }
    issuesEnabled
    boards {
```

```
nodes {  
  id  
  name  
}  
}  
}  
}
```

Ce type de requête permet une interaction profonde avec les données de GitLab, offrant une flexibilité et une puissance bien supérieures aux requêtes API REST classiques.

# Les commandes spécifiques aux projets

Les commandes décrites ci-dessous ne fonctionnent que si elles sont exécutées depuis un répertoire contenant un repository gitlab.

## Gestion des Labels avec `glab label`

`glab label` est une commande clé pour la gestion des labels dans les projets GitLab, offrant diverses options pour une organisation efficace des issues et des merge requests.

Voici la liste des principales options :

- `create` : Pour créer un nouveau label.
- `list` : Pour lister tous les labels d'un projet.

Ces options facilitent la création, la visualisation, la modification et la suppression de labels, permettant ainsi une meilleure organisation et catégorisation des éléments de travail dans vos projets GitLab.

## Gestion des Issues avec `glab issue`

La commande `glab issue` fournit un ensemble complet d'outils pour gérer les issues dans vos projets GitLab.

Voici la liste des principales options :

- `board` : Pour visualiser et gérer les boards d'issues.



- `close` : Pour fermer une issue.
- `create` : Pour créer une nouvelle issue.
- `delete` : Pour supprimer une issue.
- `list` : Pour lister les issues.
- `note` : Pour ajouter une note à une issue.
- `reopen` : Pour rouvrir une issue fermée.
- `subscribe` : Pour s'abonner aux mises à jour d'une issue.
- `unsubscribe` : Pour se désabonner des mises à jour d'une issue.
- `update` : Pour mettre à jour une issue.
- `view` : Pour afficher les détails d'une issue.

## Exemples :

```
glab issue list
```

```
No open issues match your search in mon-site1/docusaurus
```

# Gestion des Merge Requests avec `glab mr`

`glab mr` est une commande qui permet de gérer les merge requests dans GitLab, offrant une large gamme d'options pour une gestion efficace.

Voici la liste des principales options :

- `approve` : Approuvez une merge request.
- `approvers` : Gérez les approbateurs d'une merge request.
- `checkout` : Récupérez localement une merge request.
- `close` : Fermez une merge request.
- `create` : Créez une nouvelle merge request.
- `delete` : Supprimez une merge request.
- `diff` : Affichez les différences dans une merge request.
- `fork` : Faites un fork d'une merge request.
- `issues` : Listez les issues associées à une merge request.
- `list` : Listez les merge requests.
- `merge` : Fusionnez une merge request.
- `note` : Ajoutez une note à une merge request.
- `rebase` : Rebasez une merge request.
- `reopen` : Rouvrez une merge request fermée.
- `revoke` : Révoquez une approbation de merge request.
- `subscribe` : Abonnez-vous aux mises à jour d'une merge request.
- `todo` : Ajoutez une merge request à votre liste de tâches.
- `unsubscribe` : Désabonnez-vous des mises à jour d'une merge request.
- `update` : Mettez à jour une merge request.
- `view` : Affichez les détails d'une merge request.

Ces options permettent une gestion complète et flexible des merge requests dans vos projets GitLab. Chacune de ces options offre une flexibilité et un contrôle précis sur la gestion des merge requests, essentiels pour une collaboration et une intégration de code efficaces dans vos projets GitLab.

## Gestion des Incidents avec `glab incident`

`glab incident` offre des commandes spécifiques pour gérer efficacement les incidents dans vos projets GitLab.

- `close` : Pour fermer un incident.
- `list` : Pour lister les incidents d'un projet.
- `note` : Pour commenter un incident dans GitLab.
- `reopen` : Pour rouvrir un incident résolu.
- `subscribe` : Pour s'abonner à un incident.
- `unsubscribe` : Pour se désabonner d'un incident.
- `view` : Pour afficher le titre, le corps et d'autres informations sur un incident.

Ces options permettent un suivi et une gestion complets des incidents, essentiels pour la résolution rapide et efficace des problèmes critiques.

## Gestion des Releases avec `glab release`

La commande `glab release` fournit un ensemble d'outils pour la gestion des releases dans vos projets GitLab.

- `create` : Pour créer une nouvelle release.
- `delete` : Pour supprimer une release.
- `download` : Pour télécharger les assets d'une release.
- `list` : Pour lister toutes les releases d'un projet.
- `upload` : Pour téléverser des assets à une release.
- `view` : Pour afficher les détails d'une release.

Ces commandes couvrent l'ensemble du cycle de vie d'une release, de sa création à sa suppression, en facilitant la distribution et la gestion des versions dans vos projets GitLab.

## Création de snippets avec `glab snippet`

La commande `glab snippet` est un outil puissant pour gérer les extraits de code (snippets) dans GitLab, permettant un partage et une collaboration efficaces sur des morceaux de code ou des informations techniques.

**Exemple :**

```
glab snippet create sidebars.js --title "Title of the snippet"
* Creating snippet in mon-site1/docusaurus
$3634399 Title of the snippet (sidebars.js)
https://gitlab.com/mon-site1/docusaurus/-/snippets/3634399
```

## Lister les événements avec `glab user events`

La commande `glab user events` est spécialement conçue pour suivre les activités et les événements associés à un utilisateur GitLab. Elle est particulièrement utile pour surveiller l'engagement et les contributions au sein d'un projet ou à travers plusieurs projets sur GitLab.

Cette commande peut être utilisée pour suivre les contributions récentes d'un utilisateur, comme les commits, les merge requests et les commentaires sur les issues.

```
glab user events

Pushed to branch dev at mon-site / docusaurus
"corrections"
```

## Les commandes pour les pipelines

### Gestion des Pipelines avec `glab ci`

La commande `glab ci` est essentielle pour gérer les aspects de l'intégration continue (CI) dans GitLab, offrant un large éventail de fonctionnalités pour optimiser et surveiller les pipelines CI.

Une option importante qui permet d'indiquer un autre repository :

```
-R, --repo OWNER/REPO
```

Les principales options :

- `artifact` : Pour gérer les artefacts d'un job CI.
- `ci` : Pour afficher les pipelines récents.
- `delete` : Pour supprimer un pipeline.

- `get` : Pour obtenir des détails spécifiques sur un pipeline.
- `lint` : Pour valider le fichier `.gitlab-ci.yml`.
- `list` : Pour lister les pipelines.
- `retry` : Pour relancer un pipeline échoué.
- `run` : Pour exécuter un pipeline manuellement.
- `status` : Pour vérifier le statut d'un pipeline.
- `trace` : Pour suivre les logs d'exécution d'un job CI.
- `trigger` : Pour déclencher un pipeline via une API.
- `view` : Pour visualiser les détails d'un pipeline.

Ces commandes permettent une gestion complète et efficace des pipelines CI, depuis leur création jusqu'à leur surveillance, en passant par l'optimisation et le débogage.

## Planifier des pipelines avec `glab schedule`

La commande `glab schedule` est conçue pour gérer les pipelines planifiés dans GitLab, permettant une automatisation et une planification efficaces des tâches CI/CD.

- `create` : Pour planifier un nouveau pipeline. Cette option permet de configurer des pipelines pour s'exécuter automatiquement à des moments spécifiés.
- `delete` : Pour supprimer un pipeline planifié en utilisant son ID. Cela est utile pour annuler les pipelines qui ne sont plus nécessaires.
- `list` : Pour obtenir la liste des pipelines planifiés. Cette commande donne un aperçu de tous les pipelines programmés dans un projet.
- `run` : Pour exécuter manuellement un pipeline planifié. Cela permet de déclencher un pipeline spécifique sans attendre sa prochaine exécution programmée.

Ces commandes offrent une gestion complète des pipelines planifiés, de leur création à leur suppression, en passant par leur exécution et leur surveillance.

## Cas d'Usage de `glab`

L'utilisation de `glab` dans ces contextes permet d'optimiser les workflows, d'augmenter l'efficacité et de réduire les délais de développement et de déploiement.

## Utilisation sur le Poste de Travail

Sur le poste de travail d'un développeur ou d'un administrateur système, `glab` permet une interaction rapide et efficace avec GitLab.

- **Gestion des Repositories** : Clonage, création et gestion des repositories directement depuis le terminal.

- **Suivi des Incidents et Issues** : Gestion rapide des incidents et issues sans passer par l'interface web.
- **Requêtes API Personnalisées** : Utilisation de `glab api` pour des requêtes spécifiques, facilitant l'intégration avec d'autres outils ou scripts.

## Utilisation dans les Pipelines CI/CD

`glab` est particulièrement utile dans les pipelines CI/CD pour automatiser diverses tâches liées à GitLab.

- **Automatisation des Merge Requests** : Création automatique de merge requests suite à des commits ou des changements dans les branches.
- **Gestion des Issues** : Ouverture et mise à jour des issues basées sur les événements de pipeline.
- **Reporting** : Envoi de rapports de build ou de tests directement depuis les pipelines.