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 pushs 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 efficientes, 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.

Created 2025-02-03 17:51:16 UTC by Nicolas
Updated 2025-02-03 18:09:18 UTC by Nicolas