

GitLab CI / CD

m2iformation.fr





Introduction

Qu'est ce que GitLab CI / CD ?

GitLab CI / CD est une utilisation supplémentaire que l'on peut avoir d'un dépôt Git sur GitLab. Son objectif est de permettre aux développeurs d'automatiser une série de tâches en rapport avec leur code afin d'alléger leur charge de travail. Via GitLab CI, il est possible de permettre l'exécution de tests et potentiellement d'un déploiement de façon automatisée ou planifiée, que cela soit sur toutes les branches, sur certaines branches, lors de certains évènements ou encore manuellement.

Installer notre environnement

Pour utiliser GitLab CI / CD, il est possible de ne rien avoir besoin de faire d'autre que de posséder un compte sur GitLab et de créer le bon type de fichier au bon endroit. Cependant, les limites imposées par GitLab concernant l'exécution de pipelines est de 400 min d'utilisation de processus pour tout le compte et tous les dépôts, peu importe la date. Cette limite est présente afin de permettre un essai du mécanisme de création de pipeline, mais est bien souvent insufisante lorsqu'il s'agit de faire une utilisation complète et dans la durée de GitLab CI / CD.

GitLab On-Premise

L'une des solution est déjà de procéder à l'installation de GitLab sur notre propre machine. De la sorte, on disposera de notre propre système de dépôt Git privatisé. Il est ensuite possible d'administrer notre instance de GitLab et d'en gérer certains paramètres tels que d'ajouter des runners disponible pour l'ensemble des projets et des groupes. Un runner est un service qui sera souvent lancé en tâche de fond sur une machine dans le but de récupérer le code présent dans les dépots et de procéder à l'exécution du script fourni sous la forme d'un fichier `.gitlab-ci.yml` qui doit se trouver à la racine du dépôt Git.

Windows ou Linux ?

Pour pouvoir utiliser GitLab On-Premise, il est nécessaire de posséder un environnement de travail utilisant comme système d'exploitation une distribution de Linux telle qu'Ubuntu ou Debian. Ceci est dû au fait que GitLab ne supporte pas réellement l'utilisation de On-Premise sur des machines fonctionnant sous Windows.

Pour installer GitLab On-Premise sur nos machines, il faut passer par ce [lien](#). Les instruction d'installation sont disponibles sur la page relative à la distribution choisie.

Windows ou Linux ?

Il est cependant tout à fait possible de nous servir de <https://gitlab.com> pour la réalisation de nos pipelines, mais il faut garder en tête que l'on n'aura pas la capacité d'administrer GitLab directement (nous ne serront après tous que de simples utilisateurs à ce niveau là).

Utiliser le site officiel de GitLab pour lancer l'exécution de nos pipeline aura aussi comme particularité que de rendre disponible le code source présent sur nos dépôts via le réseau internet, ce qui est particulièrement utile dans le cas où l'on voudrait faire usage de runner utilisant Docker.

Runner local ?

Ce qui est par contre nécessaire lors de l'exécution de pipelines gratuites sans limite de temps et de minutes d'exécution, c'est de faire appel à notre propre runner. Pour installer un runner, il nous faut aller sur ce [lien](#). Un runner peut être installé sur Windows au besoin, mais l'exécution de jobs nécessitant des services peut poser problème à cause du mécanisme de fonctionnement de Docker Desktop. Il est donc plutôt conseillé de faire appel à une machine virtuelle sur laquelle on procédera à l'installation du runner.

Ajout du runner

Pour ajouter un runner dans l'ensemble des projets GitLab que l'on voudrait créer, le plus simple est de créer un runner de type Group Runner.

Il est en effet possible de créer un runner disponible à plusieurs niveaux hiérarchiques:

- Runner d'instance (Ensemble de GitLab)
- Runner de groupe
- Runner de projet

Pour créer un runner de groupe, il faut donc dans un premier temps créer un nouveau groupe sur GitLab.

Ajout du runner

L'ajout du runner dans le groupe se fait dans les paramètres de groupe.

Il est possible de retrouver la liste des runners au niveau de **Build > Runners**. Leur ajout se fera ensuite via un bouton situé en haut à droite de l'interface web.

Lors de l'ajout du runner, il faudra choisir plusieurs informations telles que:

- Les Tags
- La Configuration

Les Tags

Un tag au niveau des runner est simplement un moyen ultérieur de cibler tel ou tel runner. Dans le cas où l'on serait un client payant de GitLab, on pourrait par exemple se servir des tags pour faire appel à tel ou tel runner disponible pour l'ensemble des clients et profiter ainsi d'un environnement d'exécution de notre pipeline adapté à ses objectifs. Avoir plusieurs runners, c'est ainsi avoir la capacité de permettre:

- Exécution de plusieurs pipelines simultanément
- Optimiser les pipelines demandant un environnement particulier

Runner sans tags

Il est d'ailleurs possible de cocher l'option **Run untagged jobs** de sorte à permettre aussi à ce runner de récupérer les jobs n'en possédant pas. Ceci peut être intéressant par exemple si l'on veut avoir un runner par défaut sur lequel l'ensemble des jobs vont s'effectuer, mais également par exemple un autre runner disposant par défaut d'un environnement tournant dans un conteneur Docker avec une image de PHP.

Configuration du Runner

Vient ensuite la question de la configuration. L'option **Maximum job timeout** est de son côté obligatoire et il faudra définir une valeur de temps maximal possible pour l'exécution d'un job supérieure à 600ms. Le caractère **Protected** d'un runner peut également être intéressant de sorte à réserver l'exécution de runner uniquement sur des branches de type production. Quand au paramètre **Paused**, il sert simplement à éviter de lancer directement le runner si l'on ne veut pas, par exemple, qu'il se mette à récupérer des jobs en attente de runner dès son arrivée dans GitLab.

Configuration locale

Une fois la configuration au niveau de GitLab faite, il devient possible d'ajouter (enregistrer) un runner. Pour ce faire, une ligne de commande est fournie après avoir choisir notre système d'exploitation et le type de runner:

```
gitlab-runner register \
--url https://gitlab.com \
--token <token>
```

Cette commande va permettre à **GitLab-Runner** d'intéroger GitLab via notre token de sorte à relier notre runner local avec l'instance de GitLab (Self-managed ou non).

Exécuteur ?

Lorsque l'on va entrer la ligne de commande pour enregistrer le runner, une série de question nous sera posée. Dans cette série de question va se trouver à un moment donné la demande d'un type d'**exéiteur**. En fonction de ce qui est installé localement ou non, on aura à notre disposition plus ou moins de suggestions. Ces suggestions servent simplement à nous donner les valeurs possibles de l'exéiteur.

Exécuteur Shell

Si l'on choisi **shell**, alors on va demander à créer un runner qui fera tourner les pipelines localement, en se servant de notre propre terminal et donc des commandes disponibles sur notre machines.

L'utilisation de l'exécuteur Shell permet de ne pas avoir à installer certaines dépendances telles que Node.js si l'on les possède déjà localement. Elle permet aussi, au besoin, de permettre un réalisation plus facile de pipeline. Cependant, faire tourner les pipelines sur notre propre machine veut également dire que l'on va avoir une pléthore de dossier et de fichiers qui vont se créer et se peupler à chaque job. Le nettoyage n'est pas automatique.

Exécuteur Docker

Une autre solution serait par exemple de faire appel à l'exécuteur **Docker**, qui comme son nom l'indique va se servir de Docker pour fonctionner. En réalité, il s'agit de créer à chaque fois qu'un job est nécessaire à être réalisé, de procéder à la création d'un conteneur dans lequel l'ensemble du job va être réalisé. Une fois le job terminé, le conteneur sera automatiquement supprimé, ce qui nettoiera donc également le clone du dépôt et l'ensemble des tâches effectuées par le job.



Bonnes pratiques sur GitLab

Sécuriser les branches

Il est important de procéder à l'ajout de sécurité dans notre environnement GitLab. De sorte à éviter de pousser du code sur les branches relatives à la production, il peut être de bon ton de les protéger. Pour cela, il faut se rendre dans les paramètres d'un projet, puis dans la section **Settings > Repository > Protected branches**. Dans cette section, il faudra ensuite changer les paramètres pour la branche **main / master** tel que **Allow to push and merge** (Permission de pousser ou de merge) soit porté à **No one** (Personne).

Modifier les règles de Merge Request

Autre modification intéressante que l'on peut faire dans l'environnement GitLab, le comportement d'une Merge Request dans **Settings > Merge Requests**. Il peut être intéressant, par soucis de clareté dans notre hiérarchie de commits, de passer la méthode de merging à **Fast-forward merge** pour avoir plus de clareté sur la branche. On peut également encourager la création d'un commit unique dans le cas du rapprochement de plusieurs commits via **Squash commit when merging** passé à **Encourage**. Enfin, pour s'entraîner à la résolution de merge request, on peut cocher les options **Pipelines must succeed** et **All threads must be resolved**.

Git Workflow

Le **Git Workflow** désigne l'ensemble des pratiques et conventions adoptées par une équipe pour organiser le travail collaboratif autour d'un dépôt Git. Un workflow bien défini permet de structurer le développement, de faciliter la collaboration et de sécuriser les livraisons.

Principaux workflows

- **Feature Branch Workflow** : chaque nouvelle fonctionnalité ou correction de bug est développée dans une branche dédiée, généralement issue de la branche principale (`main` ou `master`). Une fois terminée, la branche est fusionnée via une Merge Request.

Principaux workflows

- **Git Flow** : propose une organisation stricte avec des branches dédiées pour le développement (`develop`), la production (`main`), les fonctionnalités (`feature/*`), les correctifs (`hotfix/*`) et les releases (`release/*`).
- **Forking Workflow** : chaque contributeur travaille sur sa propre copie (`fork`) du dépôt, puis propose ses modifications via des Merge Requests.

Bonnes pratiques

- Toujours créer une branche pour chaque tâche ou fonctionnalité.
- Utiliser des noms de branches explicites (ex : `feature/login-form`, `bugfix/header-alignment`).
- Fusionner les branches via des Merge Requests pour permettre la revue de code et l'exécution automatique des pipelines CI/CD.
- Garder la branche principale protégée et à jour.

Un workflow adapté à votre équipe et à votre projet permet d'améliorer la qualité du code, de réduire les conflits et de fluidifier les déploiements continus avec GitLab CI/CD.



Les bases des Pipelines

Qu'est ce qu'un pipeline ?

Un **pipeline** dans GitLab CI/CD est un ensemble de tâches automatisées qui s'exécutent à la suite d'un changement dans le dépôt (push, merge request, etc.). Il permet d'automatiser des étapes comme la compilation, les tests, l'analyse de code ou le déploiement. Un pipeline est composé de plusieurs **stages** (étapes), eux-mêmes constitués de **jobs** (tâches unitaires).

Chaque pipeline est défini dans un fichier de configuration spécifique et s'exécute selon les règles que vous définissez, garantissant ainsi la qualité et la fiabilité du code avant sa mise en production.

Fichier de pipeline

Le fichier de configuration d'un pipeline GitLab s'appelle `.gitlab-ci.yml` et doit être placé à la racine du dépôt. Ce fichier décrit les jobs à exécuter, leur organisation en stages, les variables d'environnement, les conditions d'exécution, les images Docker à utiliser, etc.

Fichier de pipeline

```
stages:  
  - build  
  - test  
  
build_job:  
  stage: build  
  script:  
    - echo "Compilation..."  
  
test_job:  
  stage: test  
  script:  
    - echo "Tests..."
```

Les Jobs

Un **job** est une tâche unitaire à exécuter dans le pipeline. Chaque job contient un ou plusieurs scripts à lancer, et peut être associé à un stage, des variables, des conditions, des images Docker, etc.

Exemple de job :

```
lint:  
  stage: test  
  script:  
    - npm run lint
```

Les jobs d'un même stage peuvent s'exécuter en parallèle, tandis que les stages s'exécutent séquentiellement.

Les Stages

Les **stages** (étapes) définissent l'ordre d'exécution des jobs dans le pipeline. Chaque stage regroupe un ou plusieurs jobs. Les stages courants sont : `build`, `test`, `deploy`, mais vous pouvez en définir d'autres selon vos besoins.

Exemple :

```
stages:  
  - build  
  - test  
  - deploy
```

Les jobs du stage `build` s'exécutent avant ceux du stage `test`, etc.

Les Artifacts

Les **artifacts** sont des fichiers générés par un job et transmis aux jobs suivants ou conservés après l'exécution du pipeline (par exemple, des binaires, des rapports de tests, etc.). Ils permettent de partager des résultats entre jobs ou de les télécharger depuis l'interface GitLab.

```
build:  
  stage: build  
  script:  
    - make build  
  artifacts:  
    paths:  
      - dist/
```

Ici, le dossier `dist/` sera conservé comme artifact à la fin du job `build`



Continuous Integration

Choisir l'image Docker

Lorsque l'on veut réaliser un job, il peut être intéressant de passer par l'utilisation d'une image personnalisée. Ce choix se fait via la propriété `image` dans la description d'un job:

```
build_website:  
  image: node:22-alpine  
  script:  
    - npm ci  
    - npm run build  
  artifacts:  
    paths:  
      - build/
```

Choisir l'image Docker

En choisissant une image adaptée à nos besoin, il sera plus aisément de faire la mise en place des dépendances pour notre job car celle-ci pourrait être déjà porteuse d'une majorité d'entre elles. Par exemple, on pourrait utiliser une image de Python pour avoir directement Python d'installé sur notre machine.

Il serait d'ailleurs également possible de publier en amont une image Docker contenant l'ensemble des dépendances nécessaire au job. Ce dernier n'aura alors plus qu'à l'exploiter de sorte à ne pas nous demander de recréer la roue à chaque fois.

Dépendance entre Stages

Dans le cas où l'on ferait usage des artefacts, il vaut mieux alléger les jobs ultérieurs de sorte à spécifier une liste de dépendance. En effet, sans ajout de cette propriété et écriture de la liste, l'ensemble des jobs créant des artefacts à des stades antérieurs serviront à alimenter le téléchargement des dépendances.

Pour modifier les dépendances, il suffit d'ajouter la propriété:

```
build_website:  
  image: node:22-alpine  
  dependencies: []
```

Ici, on indique que l'on ne veut aucune dépendance.

Dépendance entre Stages

```
deploy_website:  
  image: aws-cli  
  dependencies:  
    - build_website  
  script:  
    - aws s3 cp build/index.html s3://monbucket/index.html
```

Dans ce cas de figure, on indique que l'on a besoin de récupérer les artefacts issus du job `build_website` de sorte à obtenir le dossier de build, mais pas de télécharger l'ensemble des autres artefacts.

Réaliser les tests

Au cours du pipeline, il est généralement de bon ton de procéder à la réalisation de tests. Plusieurs types de tests existent et il convient par exemple de faire l'ensemble des tests unitaires dès le début du pipeline. Pour cela, il suffit, dans le cadre d'une application de type Node.js utilisant Vite, d'avoir un job tel que:

```
unit_tests:  
  image: node:alpine  
  dependencies: []  
  script:  
    - npm ci  
    - npm run test
```

Publier des rapports de tests

La réalisation de tests unitaires va souvent amener à la création d'un ou de plusieurs fichiers de couverture (**coverage files**). Ces fichiers peuvent, si l'on le veut, être également rendus disponibles à l'issue d'un job via l'ajout sous forme d'artefact:

```
unit_tests:  
  image: node:alpine  
  ...  
  artifacts:  
    paths:  
      - reports/
```

JUnit

Dans le cas où l'on aurait un framework de tests supportant l'export d'un fichier de couverture sous la forme d'un fichier .xml utilisant la syntaxe des rapports JUnit, alors il est même possible de l'ajouter à notre pipeline pour en observer directement le contenu dans l'onglet des tests fourni par GitLab.

Pour cela, il va falloir ajouter l'export de rapports de couverture. Pour ce faire, on va devoir utiliser une nouvelle propriété `artifacts:reports`.

JUnit

On obtient donc un job dont la structure pourrait être telle que:

```
unit_tests:  
  image: node-alpine  
  dependencies: []  
  script:  
    - npm ci  
    - npm run test  
  artifacts:  
    reports:  
      junit: reports/junit-coverage.xml
```

Code Quality

Un autre type de rapport qu'il est possible d'ajouter dans le cadre d'une application disposant d'un linter est le rapport de qualité du code. Pour cela, dans le cadre d'une application Javascript, on va avoir:

```
unit_tests:  
  image: node-alpine  
dependencies: []  
script:  
  - npm ci  
  - npm run lint  
artifacts:  
  reports:  
    codequality: gl-codequality.json
```

beforeScript

Dans le cas où l'on souhaiterai faire l'exécution d'un ensemble d'instruction en amont d'un job (de sorte à ce que cela ne fasse pas forcément partie du suivi du job en soit), alors on peut utiliser deux propriétés supplémentaires dans le contenu d'un job. La première d'entre elles est **beforeScript**. Comme son nom l'indique, son objectif est de faire s'exécuter une série d'instruction en amont du job.

beforeScript

```
build_website:  
  image: node-alpine  
  dependencies: []  
  before_script:  
    - apk install curl  
    - npm ci  
  script:  
    - npm run build
```

afterScript

La propriété `after_script` permet de définir une ou plusieurs commandes qui seront exécutées après le script principal du job, que celui-ci réussisse ou échoue. Elle est utile pour effectuer des actions de nettoyage, de notification ou de collecte de logs, indépendamment du résultat du job.

afterScript

```
build_website:  
  image: node:alpine  
  dependencies: []  
  script:  
    - npm run build  
  after_script:  
    - echo "Nettoyage des fichiers temporaires"  
    - rm -rf /tmp/build-cache
```

Ici, les commandes définies dans `after_script` seront toujours exécutées à la fin du job, même si la commande principale échoue.

Stages fournis par GitLab

GitLab ne fournit pas de stages prédéfinis dans les pipelines. C'est à vous de définir les stages nécessaires à votre workflow dans le fichier `.gitlab-ci.yml` via la clé `stages`. Cependant, il existe des conventions courantes pour nommer les stages, telles que : `build`, `test`, `deploy`, `review`, etc.

Stages fournis par GitLab

```
stages:  
  - build  
  - test  
  - deploy
```

Chaque job doit être associé à un stage existant. L'ordre des stages dans la liste détermine l'ordre d'exécution dans le pipeline. Vous pouvez adapter et nommer les stages selon les besoins de votre projet.

Variables fournies par GitLab

GitLab met à disposition de nombreuses variables d'environnement automatiquement lors de l'exécution d'un pipeline. Ces variables permettent d'accéder à des informations sur le projet, le pipeline, le commit, l'utilisateur, etc.

Variables fournies par GitLab

Quelques exemples courants :

- `CI_COMMIT_REF_NAME` : nom de la branche ou du tag déclencheur du pipeline.
- `CI_COMMIT_SHA` : hash du commit courant.
- `CI_PIPELINE_ID` : identifiant unique du pipeline.
- `CI_PROJECT_PATH` : chemin du projet (namespace/projet).
- `CI_JOB_STAGE` : nom du stage en cours d'exécution.
- `CI_RUNNER_DESCRIPTION` : description du runner utilisé.

Variables fournies par GitLab

Vous pouvez utiliser ces variables dans vos scripts ou dans le fichier `.gitlab-ci.yml` :

```
deploy:  
  stage: deploy  
  script:  
    - echo "Déploiement de la branche $CI_COMMIT_REF_NAME"  
    - echo "Pipeline ID : $CI_PIPELINE_ID"
```

La liste complète des variables fournies par GitLab est disponible dans la [documentation officielle](#).

Variables personnalisées

Dans le cas où l'ensemble des variables proposées par GitLab n'est pas suffisante pour nos besoins, on peut créer nous même de nouvelles variables. Pour cela, on peut se rendre dans l'interface web et aller dans **Settings > CI / CD > Variables**. Cette section permet de voir et de créer de nouvelles variables. Plusieurs options sont possible lors de la création de variables personnalisées.

Variables personnalisées

- Le type de la variable permet de créer au besoin une variable référençant un fichier (dans le cas où l'on aurait besoin d'accéder à un ensemble de données placée dans un fichier texte, par exemple).
- La visibilité va permettre de décider du caractère caché ou non d'une variable. On peut s'en servir pour créer ce que l'on appelle des secrets. Un secret permet par exemple de stocker une variable d'environnement contenant des accréditations de façon sécurisée.

Variables personnalisées

- On peut ensuite choisir s'il s'agit d'une variable disponible uniquement sur les branches protégées ou non, ainsi que si la variable doit permettre, dans sa valeur, de traiter d'autres variables via \$

Enfin, on va devoir choisir la clé et la valeur de notre variable d'environnement. Attention, dans le cas où l'on souhaiterai utiliser un secret, la valeur doit être exempte de tout caractère d'espacement. Il est également possible de donner un nom à notre variable, pour plus de lisibilité.

Pipeline de branche

Lorsque l'on veut créer des pipelines et des jobs, il peut être intéressant de spécifier si un job doit s'exécuter en permanence ou s'il doit être spécifique à une série de conditions telle que le fait qu'il soit lancé sur une branche particulière, si tel individu a déclenché le pipeline, etc...

Pour spécifier si un job doit avoir lieu sur une branche particulière, on peut utiliser la propriété `only`:

Pipeline de branche

```
deploy:  
  stage: deploy  
  only:  
    - main  
  script:  
    - echo "Déploiement de la branche $CI_COMMIT_REF_NAME"  
    - echo "Pipeline ID : $CI_PIPELINE_ID"
```

Pipeline de branche

Il est possible de procéder en sens inverse, via l'exclusion de branche pour l'exécution d'un job. Pour cela, on va se servir à la place de `except`:

```
deploy:  
  stage: build  
  except:  
    - main  
  script:  
    - echo "Je suis lancé sur les branches autre que 'main'"
```

Pipeline de branche

`only` et `except` sont cependant désormais dépréciés. GitLab a fourni une nouvelle propriété et une nouvelle logique permettant la réalisation de sélecteurs de branches. Cette nouvelle méthodologie demande de s'y connaître un peu plus en structure conditionnelle et va demander l'ajout de `rules`:

```
deploy:  
  stage: build  
  rules:  
    - if: '$CI_COMMIT_REF_NAME == main'  
  script:  
    - echo "Je suis lancé sur 'main' uniquement"
```

Pipeline de MergeRequest

Via l'utilisation des règles et des variables d'environnement fournies par GitLab, il est même possible de réaliser l'exécution de pipeline uniquement lors de certains types d'évènement, tel qu'un merge request. Pour cela, il faut encore une fois utiliser les règles. Malheureusement, l'ajout de règles d'exclusion de jobs et l'utilisation de merge request va causer un problème au niveau du fonctionnement de GitLab. En réalité, il n'est pas possible de lancer un job tout seul. Il faut qu'il soit relié à une pipeline. Dans le cas où l'on souhaiterai séparer les jobs, il va donc falloir trouver une autre solution pour ne pas multiplier nos pipelines.

Créer un Workflow

Un **workflow** dans GitLab CI/CD correspond à la manière dont les pipelines sont déclenchés et structurés selon différents événements ou conditions. Il permet de contrôler précisément quand et comment les jobs et pipelines s'exécutent.

Créer un Workflow

Par défaut, chaque push ou merge request déclenche un pipeline. Mais il est possible de personnaliser ce comportement avec la clé `workflow` dans le fichier `.gitlab-ci.yml` :

```
workflow:  
  rules:  
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'  
    - if: '$CI_COMMIT_BRANCH == "main"'
```

Ici, le pipeline ne sera déclenché que lors d'une merge request ou sur la branche `main`.

Créer un Workflow

- Limiter l'exécution aux branches principales :

```
workflow:  
  rules:  
    - if: '$CI_COMMIT_BRANCH == "main" || $CI_COMMIT_BRANCH == "develop"'
```

- Déclencher uniquement sur tag :

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

Créer un Workflow

Utiliser un workflow possède plusieurs avantages:

- Réduire le nombre de pipelines inutiles.
- Adapter l'exécution selon le type d'événement (push, MR, tag, etc.).
- Optimiser les ressources et le temps d'exécution.

Pour aller plus loin, consultez la [documentation officielle GitLab sur les workflows.](#)

Scheduled Pipeline

Un **Scheduled Pipeline** (pipeline planifié) permet d'exécuter automatiquement un pipeline à une date et une fréquence définies, indépendamment des commits ou des merge requests. Cela est utile pour lancer des tâches récurrentes comme des sauvegardes, des tests de nuit, des mises à jour de dépendances, ou des déploiements réguliers.

Scheduled Pipeline

Pour créer un pipeline planifié :

1. Rendez-vous dans votre projet GitLab.
2. Accédez à **CI/CD > Schedules** dans le menu latéral.
3. Cliquez sur **New schedule**.
4. Définissez la fréquence (via une expression cron), la branche cible, et éventuellement des variables spécifiques.

Scheduled Pipeline

Exemple d'expression cron pour lancer un pipeline tous les jours à minuit :

```
0 0 * * *
```

Vous pouvez également utiliser la clé `only: schedules` ou une règle dédiée dans vos jobs pour qu'ils ne s'exécutent que lors d'un pipeline planifié :

```
nightly_job:
  script: echo "Tâche planifiée"
  rules:
    - if: '$CI_PIPELINE_SOURCE == "schedule"
```



Continuous Delivery

Environnements de déploiement

Un **environnement** dans GitLab CI/CD représente une cible de déploiement pour votre application, comme `development`, `staging` ou `production`. Les environnements permettent de suivre les déploiements, d'effectuer des rollbacks et de visualiser l'état de chaque version déployée.

Environnements de déploiement

Pour définir un environnement dans un job :

```
deploy_staging:  
  stage: deploy  
  script:  
    - echo "Déploiement sur staging"  
environment:  
  name: staging
```

Vous pouvez configurer plusieurs environnements selon vos besoins (préproduction, recette, etc.), et GitLab affichera l'historique des déploiements dans l'interface du projet.

Manipuler les données sensibles

Pour manipuler des données sensibles (clés API, mots de passe, tokens...), il est recommandé d'utiliser les **variables d'environnement protégées** dans GitLab. Ne stockez jamais de secrets directement dans le code ou le fichier `.gitlab-ci.yml`.

Ajoutez vos secrets dans **Settings > CI/CD > Variables** et cochez l'option **Protected** si nécessaire. Utilisez-les ensuite dans vos scripts :

```
deploy:  
  stage: deploy  
  script:  
    - echo "$PRODUCTION_API_KEY" > api_key.txt
```

Environnement de Production

L'environnement de production est la cible finale de vos déploiements. Il est essentiel de sécuriser les déploiements sur cet environnement :

- Protégez la branche de production (`main`/`master`).
- Limitez les permissions de déploiement.
- Utilisez des jobs avec des règles strictes ou des approbations manuelles (`when: manual`).

Environnement de Production

```
deploy_prod:  
  stage: deploy  
  script:  
    - ./deploy_prod.sh  
environment:  
  name: production  
when: manual  
only:  
  - main
```

Smoke test ?

Un **smoke test** est un test rapide exécuté après le déploiement pour vérifier que l'application fonctionne correctement dans l'environnement cible. Il permet de détecter rapidement les problèmes majeurs.

Smoke test ?

```
smoke_test:  
  stage: test  
  script:  
    - curl -f https://mon-app.example.com/health  
environment:  
  name: production  
rules:  
  - if: '$CI_COMMIT_BRANCH == "main"'
```

Environnement de Staging

L'environnement de **staging** (préproduction) est utilisé pour valider les changements avant la mise en production. Il doit être aussi proche que possible de la production (mêmes configurations, mêmes services).

Déployez automatiquement sur staging après validation des tests, puis effectuez des tests manuels ou automatisés avant de promouvoir vers la production.

Réaliser les tests E2E

Les **tests E2E** (end-to-end) valident le fonctionnement global de l'application dans un environnement complet. Ils sont souvent exécutés sur l'environnement de staging ou de review.

```
e2e_tests:  
  stage: test  
  script:  
    - npm run e2e  
environment:  
  name: staging  
rules:  
  - if: '$CI_COMMIT_BRANCH == "develop"'
```

Environnement de Review

Les **environnements de review** sont créés dynamiquement pour chaque merge request, permettant de visualiser les changements avant la fusion. Ils facilitent la validation par les équipes métier ou QA.

Environnement de Review

```
review:  
  stage: deploy  
  script:  
    - ./deploy_review.sh  
environment:  
  name: review/$CI_COMMIT_REF_NAME  
  url: https://review-$CI_COMMIT_REF_SLUG.example.com  
rules:  
  - if: '$CI_MERGE_REQUEST_ID'
```

Construire des images

Pour déployer des applications conteneurisées, il est courant de construire une image Docker dans le pipeline :

```
build_image:  
  stage: build  
  image: docker:latest  
  services:  
    - docker:dind  
  script:  
    - docker build -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA .  
    - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA  
  only:  
    - main
```

Registre de conteneur GitLab

GitLab propose un **registre de conteneur** intégré pour stocker et distribuer vos images Docker. Utilisez les variables prédéfinies (`$CI_REGISTRY`, `$CI_REGISTRY_IMAGE`) pour vous connecter et pousser vos images.

Pour utiliser une image depuis le registre dans un job :

```
deploy:  
  image: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA  
  script:  
    - ./deploy.sh
```

Utilisation de Services

Les **services** permettent de lancer des conteneurs auxiliaires (ex : base de données, cache) pour vos jobs. Pratique pour les tests nécessitant un environnement complet.

Exemple :

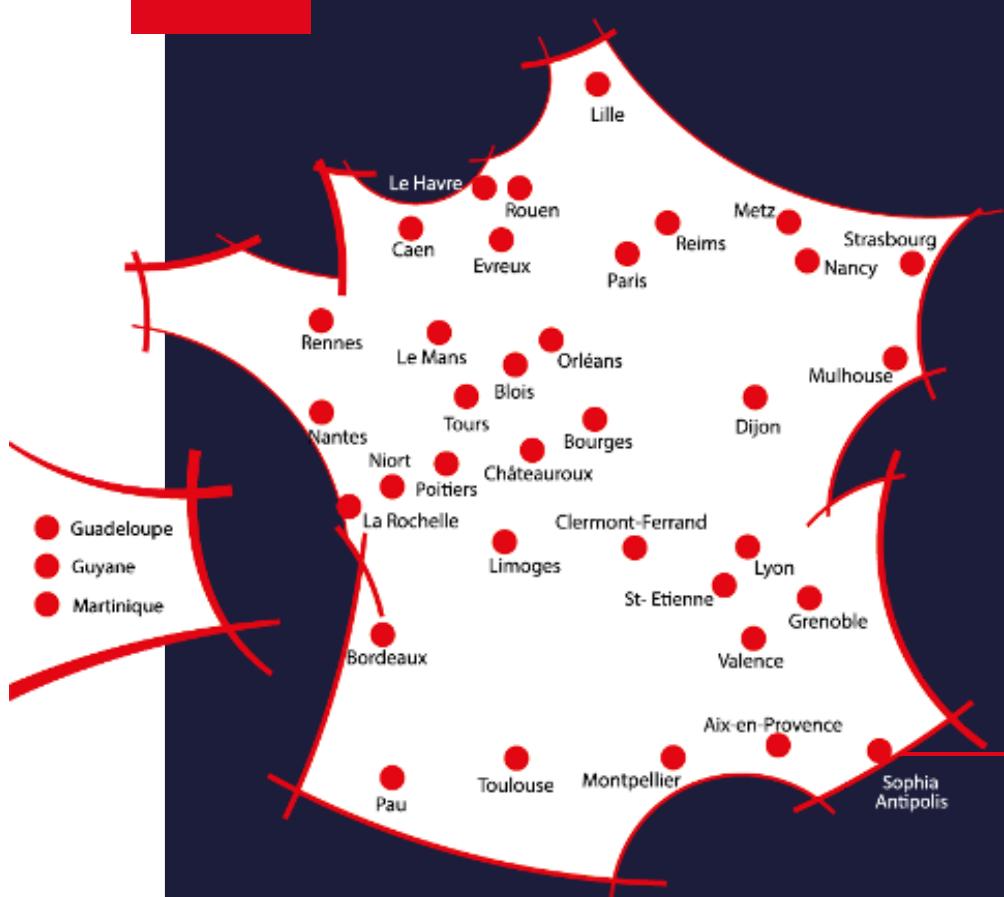
```
test:  
  image: node:alpine  
  services:  
    - name: postgres:15  
      alias: db  
  script:  
    - npm run test
```

Publier via SSH

Pour déployer sur un serveur distant via SSH, ajoutez votre clé privée comme variable protégée, puis utilisez `ssh` dans votre script :

```
deploy_ssh:  
  stage: deploy  
  script:  
    - mkdir -p ~/.ssh  
    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' > ~/.ssh/id_rsa  
    - chmod 600 ~/.ssh/id_rsa  
    - ssh -o StrictHostKeyChecking=no user@host "cd /app && git pull && ./restart.sh"  
only:  
  - main
```

Veillez à sécuriser vos clés et à limiter l'accès aux jobs de production.



Découvrez également
l'ensemble des stages à votre disposition
sur notre site m2iformation.fr

m2iformation.fr

