

GitLab CI/CD, maîtriser la gestion du cycle de vie de vos développements logiciels

Formateur

M. Brahim Hamdi

Consultant Devops/Cloud

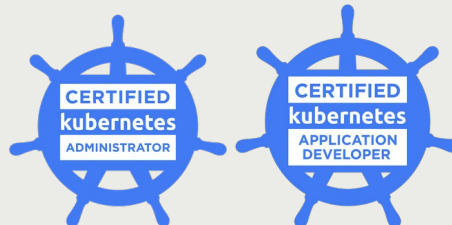
brahim.hamdi.consult@gmail.com

Novembre 2023

Formateur

Brahim Hamdi

- Consultant/formateur
- Expert DevOps & Cloud



Objectifs pédagogiques

- Connaître l'offre GitLab
- Pratiquer la gestion de versions avec Git et collaborer avec GitLab
- Mettre en place l'intégration continue (CI) et le déploiement continu (CD) avec GitLab
- Appréhender les éléments constitutifs d'une usine logicielle DevOps

Public concerné

- Développeurs,
- Chefs de projet,
- Administrateurs systèmes,
- Architectes

Prérequis

- Connaissances de base du système Linux.
- Connaissances de base de la gestion de versions avec Git.

Le plan de formation

- Présentation de GitLab
- Git et GitLab
- GitLab CI/CD
- Plus loin dans l'utilisation des runners
- Fonctionnalités complémentaires de GitLab

Présentation de GitLab

Qu'est ce que GitLab ?

- Outil **open source** de gestion de projets **git** (licence MIT)
- Application Web développée en langage Ruby par GitLab Inc
 - Dépôt : gitlab.com/gitlab-org/gitlab
- Dernière version : 16.5 (22 Octobre 2023)
- Les principales fonctionnalités :
 - Gérer le cycle de vie de projets Git
 - Gérer les participants aux projets et leurs droits (rôles, groupes, etc ...)
 - Déposer des Issues pour lister les bugs
 - Gérer la communication entre ces participants
 - Proposer des Merges Requests pour fusionner les branches
 - Lancer des pipelines d'intégration et de déploiement continu via GitLab CI/CD
 - Fournir un Wiki pour la documentation

L'offre GitLab



- GitLab SaaS (<https://gitlab.com/>) : l'offre de logiciel en tant que service de GitLab. Vous n'avez rien besoin d'installer pour utiliser GitLab SaaS, il vous suffit de vous inscrire et de commencer à utiliser GitLab immédiatement.
- GitLab Dedicated : un service SaaS à locataire unique pour les grandes entreprises hautement réglementées.
- GitLab autogéré : installez, administrez et gérez votre propre instance GitLab.

Les différentes distributions (autogéré)

- GitLab Community Edition (CE) : version libre
- GitLab Enterprise Edition (EE) : version payante

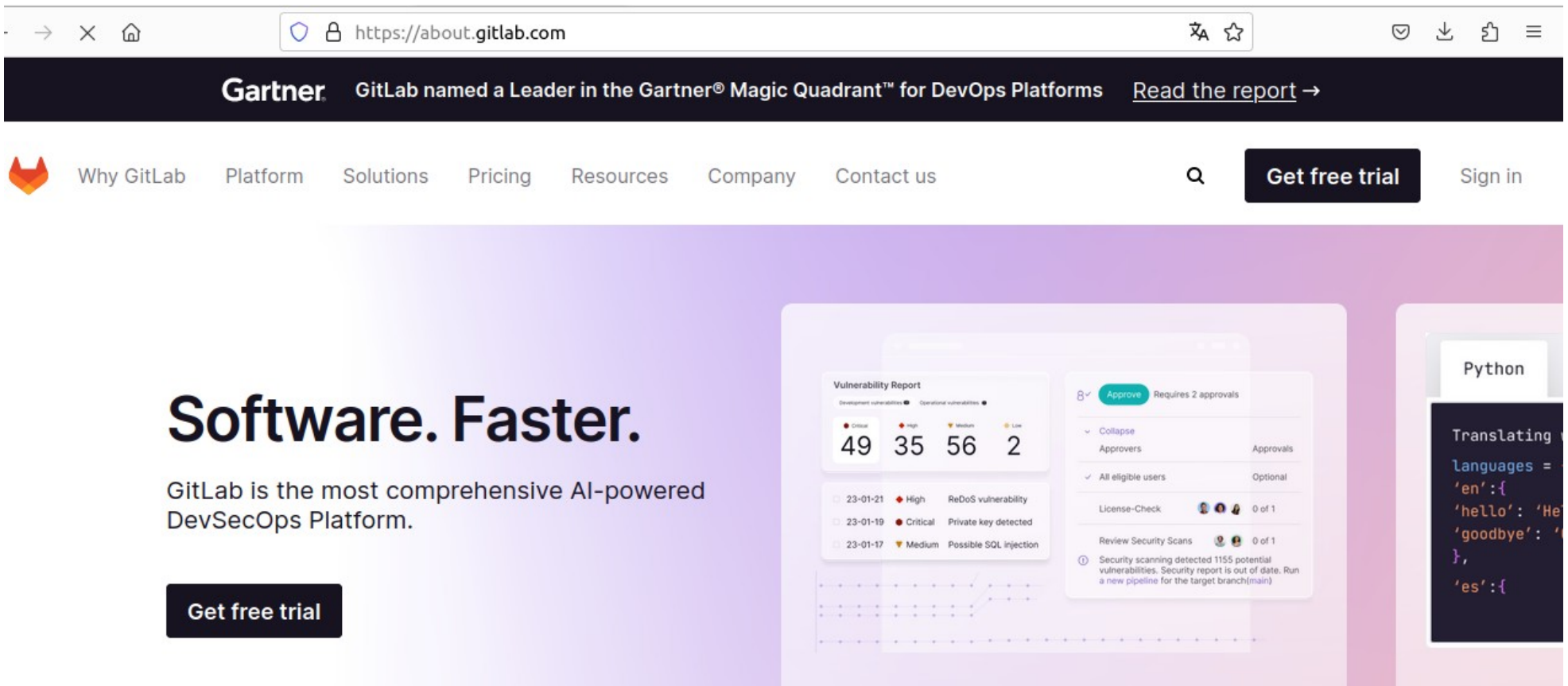
Feature	CE	EE
file manager, issues, wiki	✓	✓
Online code changes	✓	✓
GitHub import	✓	✓
LDAP/AD authentication	✓	✓
CI and Docker support	✓	✓
Support	×	✓
Kerberos authentication	×	✓
Merge Request Approvals	×	✓
Issues/Merge Requests templates	×	✓

GitLab vs GitHub

 GitHub	 GitLab
Les issues peuvent être suivies dans plusieurs repositories	Les issues ne peuvent pas être suivies dans plusieurs repositories
Repositories privés payants	Repositories privés gratuits
Pas d'hébergement gratuit sur un serveur privé	Hébergement gratuit possible sur un serveur privé
Intégration continue uniquement avec des outils tiers (Travis CI, CircleCI, etc.)	Intégration continue gratuite incluse
Aucune plateforme de déploiement intégrée	Déploiement logiciel avec Kubernetes
Suivi détaillé des commentaires	Pas de suivi des commentaires
Impossible d'exporter les issues au format CSV	Exportation possible des issues au format CSV par e-mail
Tableau de bord personnel pour suivre les issues et pull requests	Tableau de bord analytique pour planifier et surveiller le projet

Inscription sur gitlab.com

- Créer un compte GitLab et authentifiez-vous avec votre login.



The screenshot shows the GitLab website homepage. At the top, there's a dark blue header with the text "Gartner GitLab named a Leader in the Gartner® Magic Quadrant™ for DevOps Platforms" and a link "Read the report →". Below this is a navigation bar with the GitLab logo, links for "Why GitLab", "Platform", "Solutions", "Pricing", "Resources", "Company", and "Contact us", a search icon, a "Get free trial" button, and a "Sign in" link. The main content area has a purple background. On the left, it says "Software. Faster." and "GitLab is the most comprehensive AI-powered DevSecOps Platform." with a "Get free trial" button. On the right, there are three overlapping cards: a "Vulnerability Report" card showing counts for Critical (49), High (35), Medium (56), and Low (2) vulnerabilities, a "Merge Request" card showing an "Approve" button and "Requires 2 approvals", and a "Python" card showing a code snippet for translating languages.

Software. Faster.

GitLab is the most comprehensive AI-powered DevSecOps Platform.

[Get free trial](#)

Vulnerability Report

Critical	High	Medium	Low
49	35	56	2

Approve Requires 2 approvals

Python

```
Translating  
languages =  
'en': {  
'hello': 'He  
'goodbye': '  
},  
'es': {
```

Créer un nouveau projet

- Cliquez sur **Create a projet** puis sur **Create blank Projectct.**



- Indiquez un **Project Name : Car assembly line**
- Cochez qu'il s'agit d'un projet **Private**, puis validez en cliquant sur **Create Project.**

The screenshot shows the 'Create new project' form in GitLab. The form has the following fields and options:

- Project name:** A text input field containing 'Car assembly line'.
- Project URL:** A dropdown menu showing 'https://gitlab.com/' and 'leadingit.consult'.
- Project slug:** A text input field containing 'car-assembly-line'.
- Project deployment target (optional):** A dropdown menu showing 'Select the deployment target'.
- Visibility Level:** Radio buttons for 'Private' (selected) and 'Public'.
- Project Configuration:** Checkboxes for 'Initialize repository with a README' (checked) and 'Enable Static Application Security Testing (SAST)' (unchecked).

At the bottom of the form are two buttons: 'Create project' and 'Cancel'.

Page d'accueil du projet

Accès rapide aux merges requests, issues et tasks

Accès à la page d'accueil (projets, groupes, ...)

Gestion des paramètres du compte

Infos générales du projet

La branche en cours de consultation

Dernier commit sur la branche sélectionnée

Clone du dépôt

Accès à tous les outils du projet

Liste des fichiers de la branche sélectionnée

The screenshot shows the GitLab project homepage for 'Car assembly line'. The left sidebar contains navigation links: Project, Pinned, Issues, Merge requests, Manage, Plan, Code, Build, Secure, Deploy, Operate, Monitor, and Help. The main content area displays project information: 'Car assembly line' (Project ID: 52099193), 1 Commit, 1 Branch, 0 Tags, and 3 KiB Project Storage. Below this, the 'main' branch is selected, showing the 'Initial commit' by 'leadingit consult' just now. A table lists the files in the selected branch: 'README.md' (Initial commit, just now). The 'Clone' button is highlighted in blue.

Name	Last commit	Last update
README.md	Initial commit	just now

Git et GitLab

Créer un Repository git

- Configuration du compte **git**

```
git config --global user.name "monNom"
```

```
git config --global user.email "mon@email"
```

- Pour désactiver la coloration dans la console (par défaut activée)

```
git config --global color.ui false
```

- Pour consulter la liste de configurations (et vérifier les modifications)

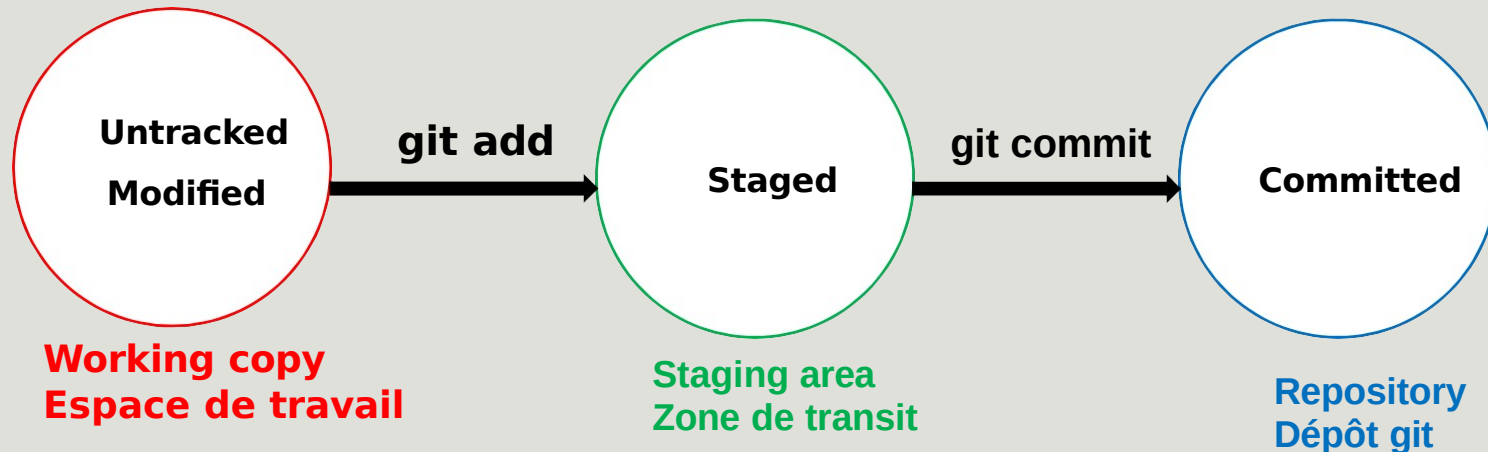
```
git config --list
```

- Pour vérifier la valeur d'une propriété de configuration

```
git config user.name
```

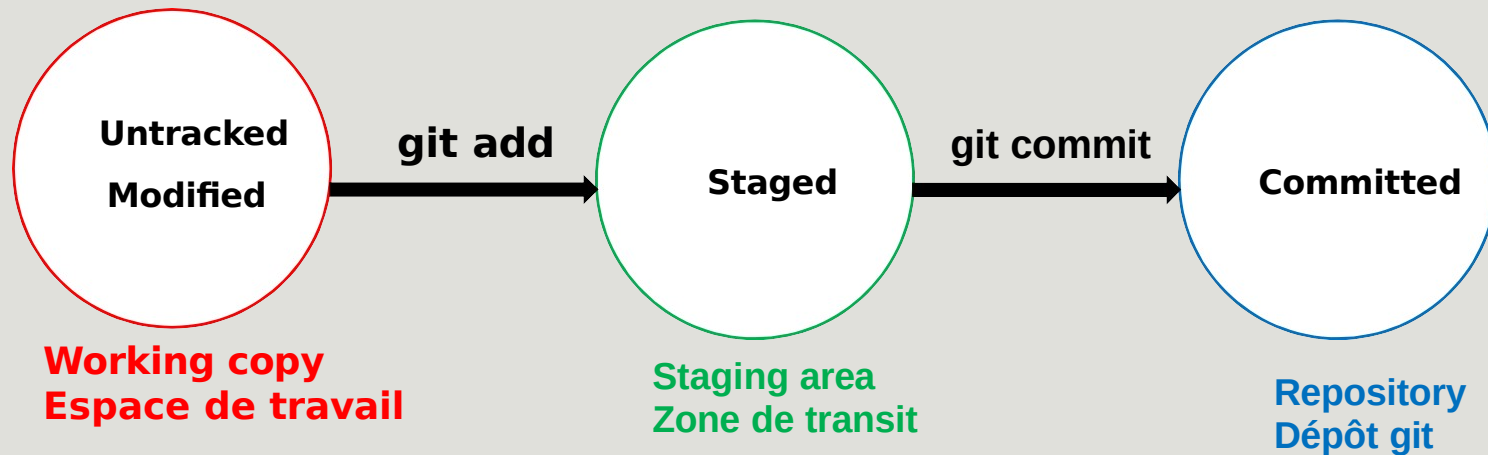

États des fichiers

☒ Un fichier doit être explicitement ajouté au dépôt Git



- **Commit** : ensemble cohérent de modifications
- **Working copy** (ou copie de travail) : contient les modifications en cours (c'est le répertoire courant)
- **Staging area** (ou index) : Zone de transit qui contient la liste des modifications effectuées dans la working copy qu'on veut inclure dans le prochain commit
- **Repository** : ensemble des commits du projet (et les branches, les tags ou libellés)

États des fichiers



- **Untracked (Fichiers non suivis) /Modified**
 - Nouveaux fichiers ou fichiers modifiés
 - Pas pris en compte pour le prochain commit
- **Staged (Fichiers Indexés)**
 - Fichiers ajoutés, modifiés, supprimés ou déplacés
 - Pris en compte pour le prochain commit
- **Unmodified/Committed**
 - Aucune modification pour le prochain commit

Premier Commit

○x Vérifions le contenu de notre dépôt

```
git status
```

○x Commençons par créer un fichier **file.txt**

(État : **untracked**)

```
echo "code source 1" > file.txt
```

○x Vérifions le contenu de notre dépôt

```
git status
```

○x Ajout du fichier dans la zone de transit

(État : **staged**)

```
git add file.txt
```

ou bien

```
git add .
```

ou aussi

```
git add --all
```

○x Vérifions le contenu de notre dépôt

```
git status
```

○x Création du commit

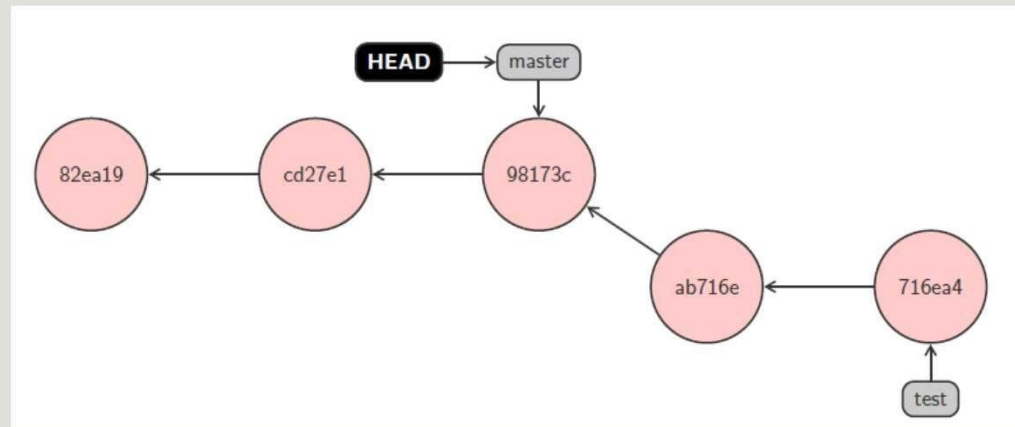
(État : **committed**)

```
git commit -m "mon premier commit"
```

Commit sur une branche

- Un commit va toujours se faire sur la branche courante

```
git checkout test
echo "code source 5">>>file.txt
git commit -am " mon cinquième commit "
echo "code source 6">>>file.txt
git commit -am " mon sixième commit "
git log --oneline
git checkout master
git log --oneline
```



Remarques

- ✓ En créant une branche, cette dernière pointe sur le commit à partir duquel elle a été créée
- ✓ En faisant un Commit à partir de la branche créée, cette dernière dévie de la branche principale

Opérations de base sur une branche

- On peut renommer une branche avec l'option -m

```
git branch -m dev alternative
```

- On peut supprimer une branche vide (ou fusionnée) avec l'option -d

```
git branch -d alternative
```

- Pour forcer la suppression d'une branche, on peut utiliser l'option -D

Fusion de branches

Problématique

- Lors de l'élaboration d'un projet, plusieurs branches seront créées, chacune pour une tâche bien particulière

Solution

- ✓ Fusionner les branches et rapatrier les modifications d'une branche dans une autre

- On peut fusionner deux branches pour en combiner les modifications
- La fusion se fait vers la branche courante
- Deux cas possibles de fusion :

- **sans conflit :**

- *fast forward : sans commit de fusion*
- *non fast forward (avec l'option `-no-ff`) : avec un commit de merge*

- **avec conflit :** avec un commit de merge

Annuler un Commit

❑ Avant cela, sur la branche **test** :

- ✓ Créer un deuxième fichier **file2.txt** contenant une ligne « *creating fle2* » et faire un Commit avec le message « *creating fle2.txt* »
- ✓ Ajouter une ligne « *updating fle2* » dans file2.txt et faire un Commit avec le message « *updating fle2.txt* »

■ Comment annuler le commit ayant comme message « *updating fle2.txt* »

```
git revert idCommit
```

■ Ensuite, modifier le message par "création du file2.txt". Cliquer sur **Echap** puis saisir **:wq** et cliquer sur **Entree** pour quitter.

🔍 Vérifions l'annulation avec

```
git log --oneline cat file2.txt
```

Supprimer des modifications

☐ Trois possibilités

- ✓ mode **mixed** (par défaut) : annuler le commit et garder les modifications dans le working directory
- ✓ mode **soft** : annuler le commit et garder les modifications dans le staging area
- ✓ mode **hard** : annuler le commit et ne pas garder les modifications

■ Ajouter une ligne « *updating fle2* » dans file2.txt et faire un Commit avec le message « *updating fle2.txt* »

■ Supprimer le commit ayant comme message « *updating fle2* » en mode **soft**

```
git reset --soft idCommit
```

ou

```
git reset --soft HEAD^
```

◦x

Vérifions l'historique

```
git log --oneline
```

◦x

Vérifions le contenu de notre dépôt

```
git status
```

◦x

Supprimer le commit ayant comme message « mon quatrième commit » en mode **hard**

```
git reset --hard idCommit
```


Les tags

Problématique

- ✓ Pour accéder à un commit qui présente une version importante de notre projet
- ✓ Il faut chercher le commit en question en lisant les messages de tous les Commits, et ensuite faire *git checkout*

Solution : utiliser les étiquettes (tags)

```
git tag -a nom-tag -m "message"
```

■ Une étiquette :

- ✓ permet de marquer un Commit/une version de notre application
- ✓ référence vers un Commit

○x Sur la branche principale master, créer un tag sur le Commit actuel

```
git tag -a v1 -m "premiere version du projet"
```

○x Création d'un tag sur un commit en utilisant l'identifiant de message «

```
git tag -a v0 idCommit -m "tag pour second"
```

La recherche

- Pour lister les tags

```
git tag --list
```

- On peut aussi se positionner sur un tag

```
git checkout v0
```

```
git log --oneline
```

- Pour supprimer un tags

```
git tag v0 --delete
```

Le fichier .gitignore

- Git peut ignorer des fichiers du répertoire de travail en utilisant le fichier *.gitignore*
- Créons le fichier *.gitignore* pour contenir les fichiers à **ignorer**

```
echo informatique.txt >> .gitignore
```

```
echo *.html >> .gitignore
```

```
echo view/* >> .gitignore
```

```
echo java >> informatique.txt
```

- En faisant *git status*, aucun fichier à indexer à l'exception de *.gitignore*

```
git status
```

- Tous les fichiers avec l'extension html sont ignorés
- Aussi, tous les fichiers du répertoire *view*

Dépôt distant

■ Dépôt distant ?

- ✓ Dépôt nu, site hébergeur : **GitHub**, **Bitbucket** et **GitLab**

■ A partir de notre premier dépôt (firstGit), créer un premier dépôt distant

```
cd ..  
cd firstGit/  
git checkout master  
git remote add origin ../firstGitBare
```

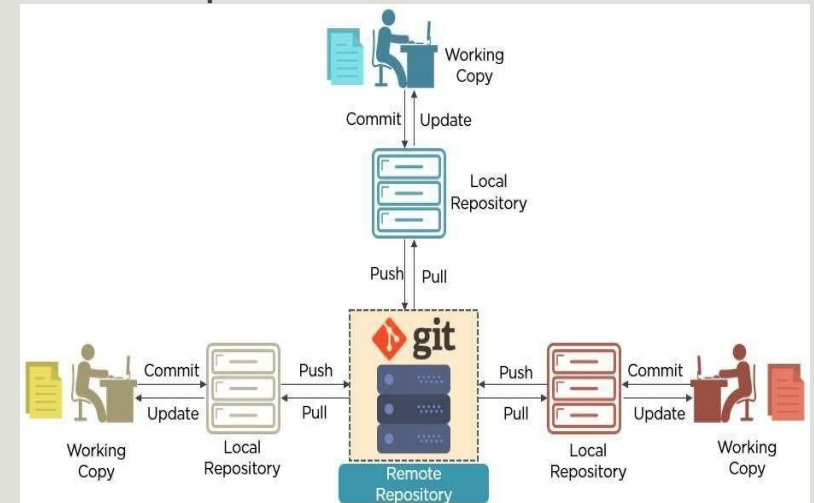
■ Afficher la liste des dépôts distants

```
git remote
```

○ Afficher les branches distantes

```
git branch -r
```

//aucun



Dépôt distant

- Envoyer (publier) la branche master sur le dépôt distant

```
git push origin master
```

- Afficher les branches distantes

```
git branch -r
```

Ou aussi

```
git branch -a
```

- Vérifier la réception de la branche à partir du *firstGitBare*

```
cd ..  
cd firstGitBare/  
git branch
```

- Afficher tous les Commit

```
git log --oneline
```

- ✎ Pour supprimer un remote

```
git remote remove nomRemote
```

- ✎ Pour renommer un remote

```
git remote rename oldName newName
```

Cloner un dépôt distant

- Se placer dans le parent du dépôt courant

```
cd ..
```

- Cloner le dépôt *firstGitBare* dans *firstGitClone*

```
git clone firstGitBare firstGitClone
```

- Se placer dans le répertoire cloné

```
cd firstGitClone
```

- Vérifier le dépôt distant

```
git remote -v
```

- Vérifier les Commits

```
git log --oneline
```

Mémo Git

- Gestionnaire de sources décentralisées
- Très populaire et performant
- Souplesse au niveau du poste du développeur
- Permet de travailler en mode déconnecté
- Favorise l'expérimentation et le test

CONFIGURATION DES OUTILS

Configurer les informations de l'utilisateur pour tous les dépôts locaux

```
$ git config --global user.name "[nom]"
```

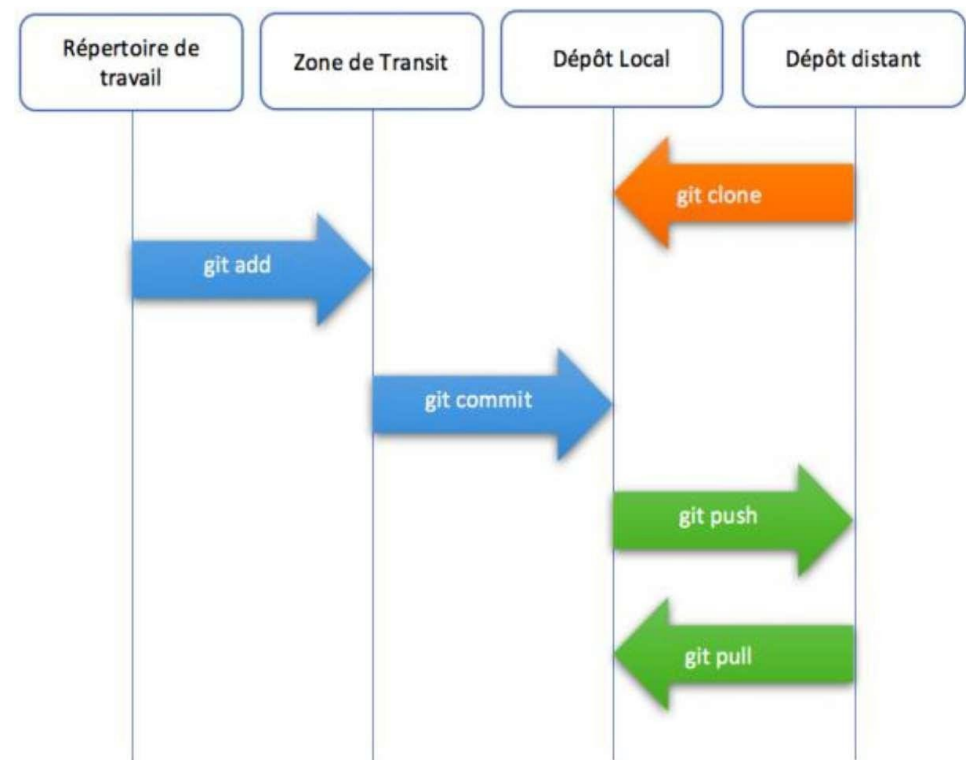
Définit le nom que vous voulez associer à toutes vos opérations de commit.

```
$ git config --global user.email "[adresse email]"
```

Définit l'email que vous voulez associer à toutes vos opérations de commit

```
$ git config --global color.ui auto
```

Active la colorisation de la sortie en ligne de commande



Mémo Git

CRÉER DES DÉPÔTS

Démarrer un nouveau dépôt ou en obtenir un depuis une URL existante

```
$ git init [nom-du-projet]
```

Crée un dépôt local à partir du nom spécifié

```
$ git clone [url]
```

Télécharge un projet et tout son historique de versions

EFFECTUER DES CHANGEMENTS

Consulter les modifications et effectuer une opération de commit

```
$ git status
```

Liste tous les nouveaux fichiers et les fichiers modifiés à commiter

```
$ git diff
```

Montre les modifications de fichier qui ne sont pas encore indexées

```
$ git add [fichier]
```

Ajoute un instantané du fichier, en préparation pour le suivi de version

```
$ git diff --staged
```

Montre les différences de fichier entre la version indexée et la dernière version

```
$ git reset [fichier]
```

Enlève le fichier de l'index, mais conserve son contenu

```
$ git commit -m "[message descriptif]"
```

Enregistre des instantanés de fichiers de façon permanente dans l'historique des versions

GROUPER DES CHANGEMENTS

Nommer une série de commits et combiner les résultats de travaux terminés

```
$ git branch
```

Liste toutes les branches locales dans le dépôt courant

```
$ git branch [nom-de-brancher]
```

Crée une nouvelle branche

```
$ git checkout [nom-de-brancher]
```

Bascule sur la branche spécifiée et met à jour le répertoire de travail

```
$ git merge [nom-de-brancher]
```

Combine dans la branche courante l'historique de la branche spécifiée

SYNCHRONISER LES CHANGEMENTS

Référencer un dépôt distant et synchroniser l'historique de versions

```
$ git fetch [nom-de-depot]
```

Récupère tout l'historique du dépôt nommé

```
$ git merge [nom-de-depot]/[branche]
```

Fusionne la branche du dépôt dans la branche locale courante

```
$ git push [alias] [branche]
```

Envoie tous les commits de la branche locale vers GitHub

```
$ git pull
```

Récupère tout l'historique du dépôt nommé et incorpore les modifications

VÉRIFIER L'HISTORIQUE DES VERSIONS

Suivre et inspecter l'évolution des fichiers du projet

```
$ git log
```

Montre l'historique des versions pour la branche courante

```
$ git log --follow [fichier]
```

Montre l'historique des versions, y compris les actions de renommage, pour le fichier spécifié

```
$ git diff [premiere-brancher]...[deuxieme-brancher]
```

Montre les différences de contenu entre deux branches

```
$ git show [commit]
```

Montre les modifications de métadonnées et de contenu incluses dans le commit spécifié

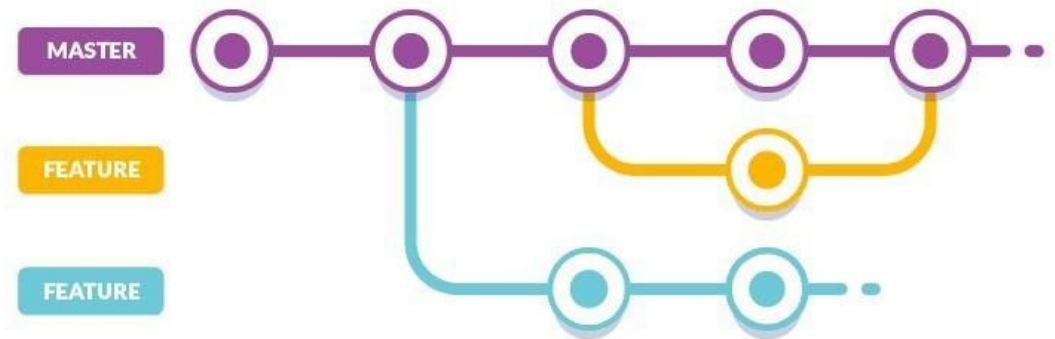
Les workflows

- **Un dépôt Git peut vite se transformer en arbre de Noël avec des guirlandes dans tous les sens**
- **Solution :** un workflow de développement
 - ✓ Pour unifier les pratiques au sein d'une même équipe
 - ✓ Pour simplifier la gestion du dépôt
 - ✓ Pour connaître en temps réel l'état de son dépôt (les fonctionnalités en cours de développement, les branches pouvant être supprimées, ...)
- **Exemples de workflow de développement**
 - ✓ **Feature Branch Workflow:** la simplicité, penser petit et synchronisation régulière
 - ✓ **GitFlow:** une solution **robuste** s'adaptant à tous les contextes



Feature Branch Workflow

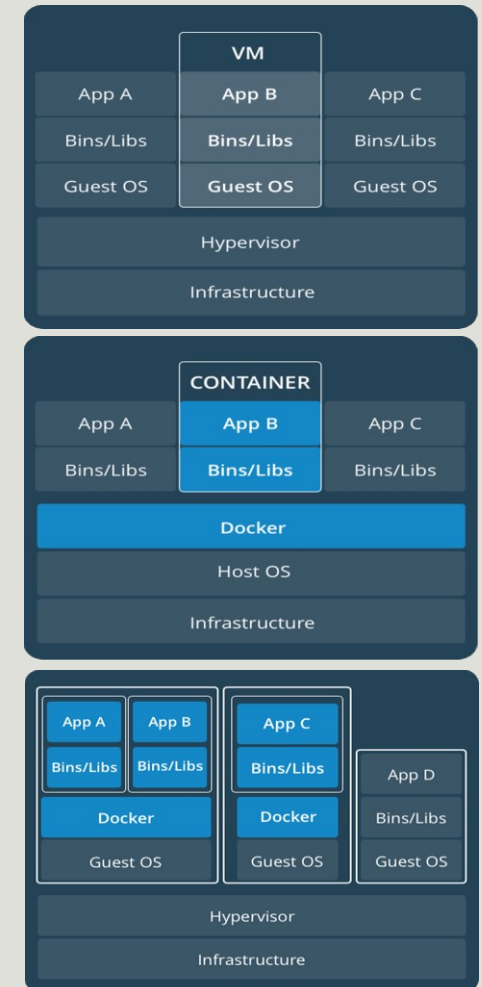
- **Workflow idéal lorsqu'il n'y a pas besoin de gérer des releases ou des versions**
- **Fonctionne surtout en équipe réduite**
- **Concept**
 - Une seule branche principale : **master**
 - Chaque **fonctionnalité** fait l'objet d'une branche **feature** tirée de la master
- **Principes**
 - Tout ce qui est sur master est **déployable**
 - Chaque branche doit avoir un nom significatif
 - Commit et push réguliers
 - **Merge request (Pull request pour Github)** à la fin d'une feature
 - Déployer directement après **merge**



GitLab CI/CD

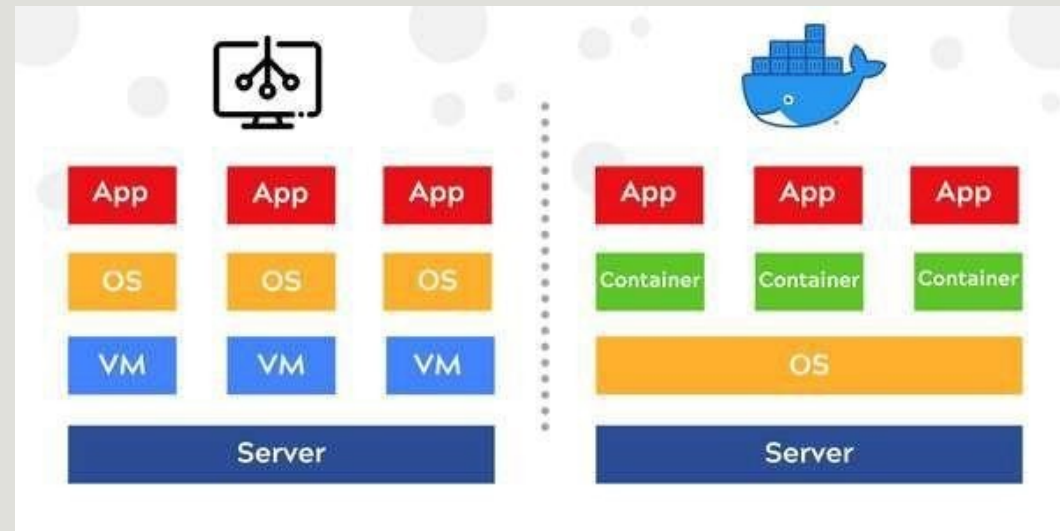
De la virtualisation à la conteneurisation

- Les machines virtuelles partagent un même serveur physique, alors que les **conteneurs** partagent le même système d'exploitation.
- Contrairement aux VMs, les **conteneurs Docker** n'incluent pas d'OS, mais permettent d'isoler des applications dans des contextes.
- La virtualisation système et applicative sont deux concepts différents, mais qui peuvent être complémentaires
 - Il est possible de contenir des conteneurs dans un système virtualisé.



Les avantages des conteneurs

- **Cohérence et Cohésion**
 - 1 conteneur = 1 microservices
- **Remplace les machines virtuels (VM)**
- **Couplage faible**
 - Chaque conteneur est indépendant
- **Liberté de déploiement**
 - Seulement besoin de Docker
 - Indépendance Technique



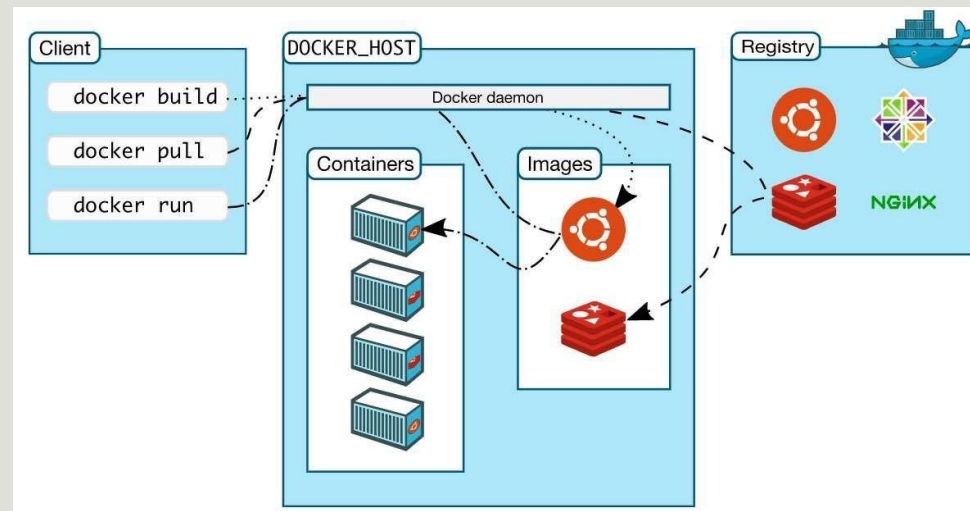
Docker

- Logiciel sous licence Apache 2.0, qui a été distribué en tant que projet **open source** à partir de mars 2013.
- **Docker** permet **d'automatiser le déploiement d'applications** dans des conteneurs logiciels.
- Il peut empaqueter une application et ses dépendances dans un conteneur isolé, qui pourra être exécuté sur n'importe quel serveur.
- Cela va nous permettre de changer l'architecture des applications monolithiques vers une architecture en microservices qui favorise l'agilité lors de la phase la plus coûteuse du cycle de vie des applicatifs : c'est-à-dire le déploiement



Container et image

- Une **image** est une collection ordonnée de modifications de layer Linux avec les paramètres d'exécution correspondants qui seront utilisés pour lancer un container.
- Les images sont toujours **Read-Only**.
- Un **container** est une instantiation dynamique d'une image active ou inactive si elle est terminée.



Pipeline et YAML

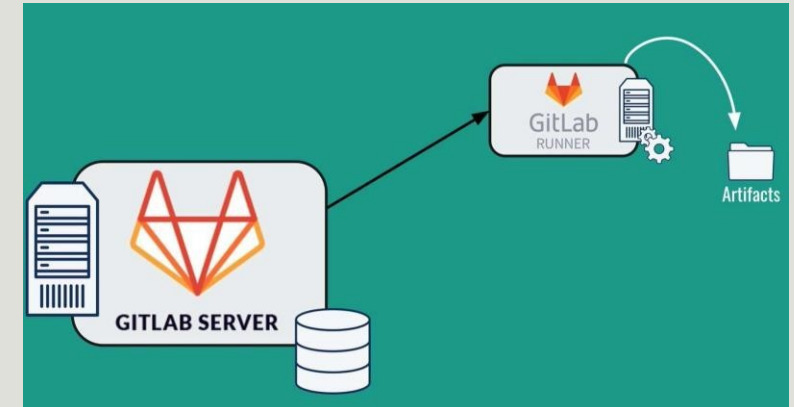
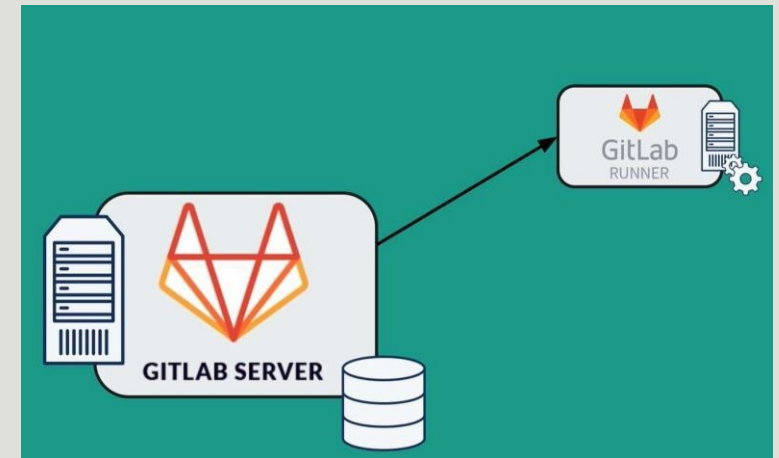
- La gestion de la pipeline CD/CD dans GitLab se fait simplement par l'ajout d'un fichier **YAML** «**.gitlab-ci.yml**» dans la racine de votre projet git concerné .
- **YAML** est un langage de sérialisation de données utilisé, par exemple, pour le déploiement de configurations par **Ansible** ou pour la configuration d'applications multi-containers via **Docker Compose**.

```
1  ---
2  #Blog about YAML
3
4  title: YAML Ain't Markup Language
5  author:
6    first_name: Lauren
7    last_name: Malhoit
8    twitter: "@Malhoit"
9  learn:
10    - Basic Data Structures
11    - Commenting
12    - When and How
```

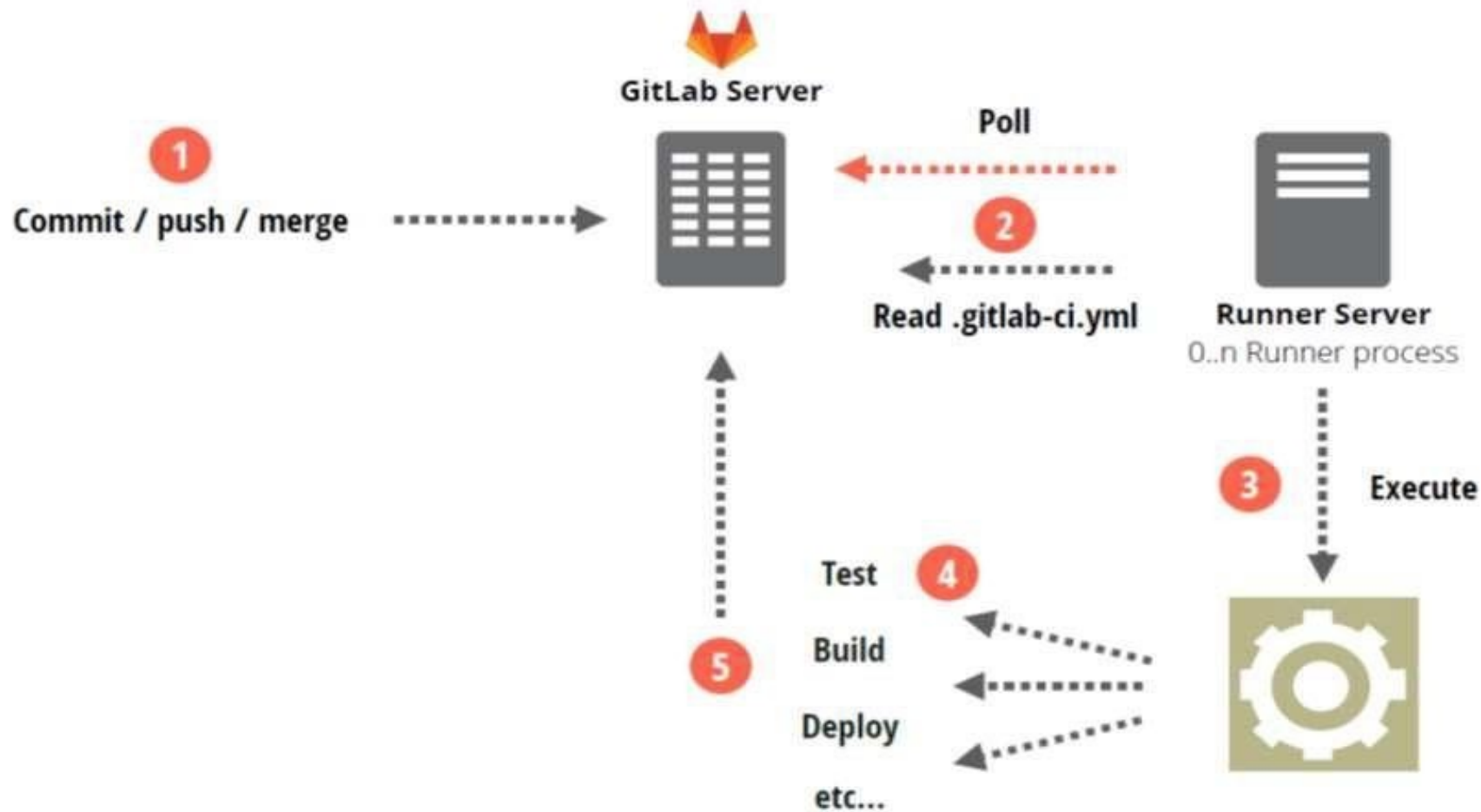
- Pour plus de détails sur YAML:
 - <https://opensharing.fr/yaml-memo-bases>
 - https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html

Architecture Gitlab: Serveur et Runner

- Vous avez besoin d'au moins d'un **serveur Gitlab** pour installer l'interface Web, **vos dépôts Git** (référentiels) et un **Runner**.
- Pour que le serveur Gitlab n'exécute pas les travaux (Jobs) et pour avoir une architecture **évolutive**, facile à déployer et **scalable**, l'exécution de ces Jobs sera déléguée au **Runner**
- Si le Job a été exécuté avec succès, nous pouvons sauvegarder les résultats (les fichiers et/ou dossiers) dans des **Artifacts**. Ces derniers vont être stockés au sein des pipelines pour être utilisés par d'autres Jobs.

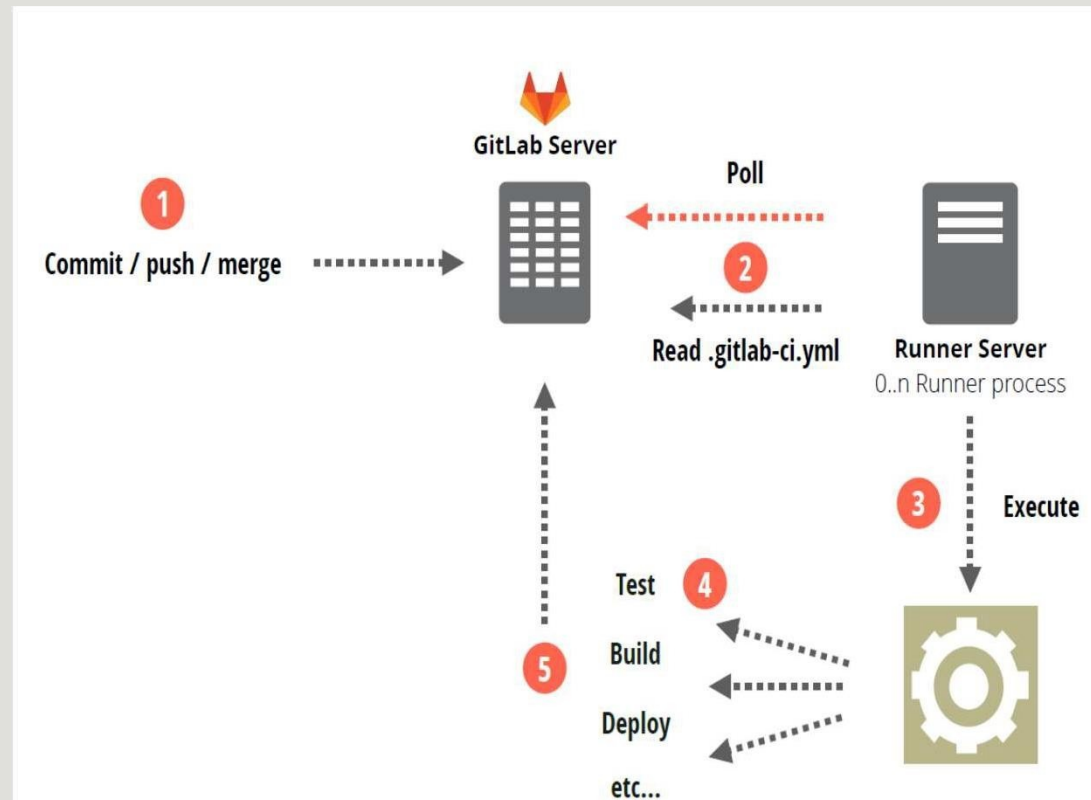


Fonctionnement de Gitlab



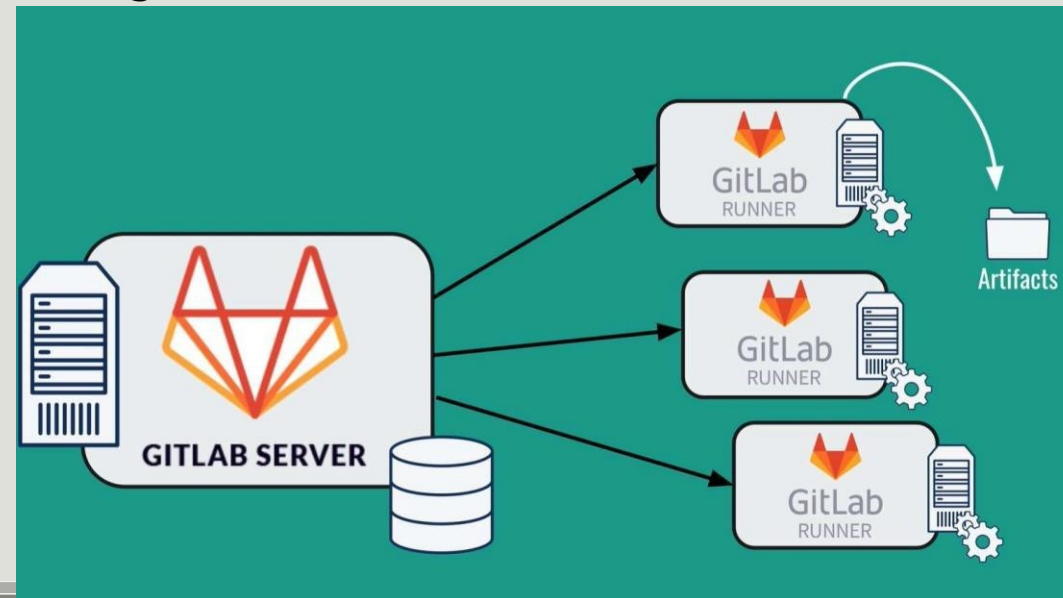
Le manifeste .gitlab-ci.yml

- Pour que la CI/CD sur GitLab fonctionne il vous faut un **manifeste .gitlab-ci.yml** à la racine de votre projet
- Dans ce manifeste vous allez pouvoir définir des **stages**, des **jobs**, des **variables**, etc.
- Le pipeline est déclenché à chaque **commit** ou **push** et s'exécute dans le Runner
- Et .gitlab-ci.yml explique au(x) Runner(s) ce qu'il faut faire



Gitlab Runner

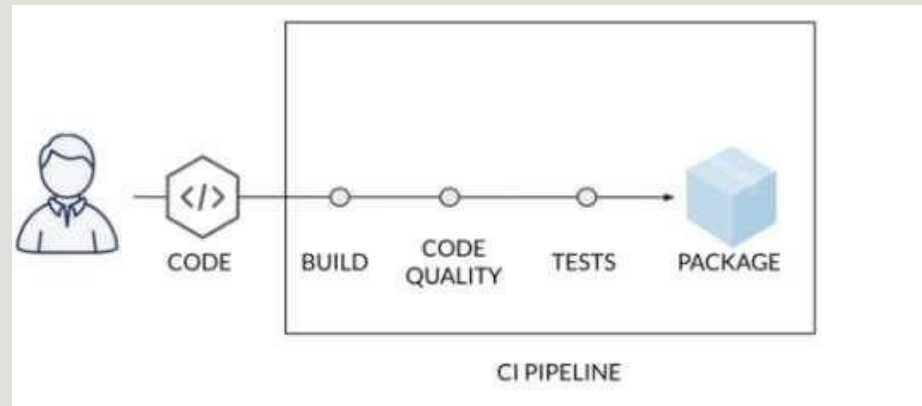
- Un Runner Gitlab-CI est un simple démon qui attend les Jobs comme vus dans le diagramme précédent.
 - Une fois un Job reçu celui-ci va demander à « un exécuteur » de traiter la demande.
 - Les **exécuteurs** sont des sous-processus qui vont se charger de faire les commandes (scripts) que vous avez définies dans votre gitlab-ci.
 - Gitlab-CI est capable de fonctionner de différente manière : SSH, Shell, Parallels, VirtualBox, Docker, Docker Machine (auto-scaling), Kubernetes, Personnalisé (Custom)
-
- Vous pouvez créer vos propres Runners
 - sur votre infrastructure en fonction de vos besoins.
 - Selon le nombre de projets que vous avez ou de l'activité que vous avez sur un projet, il vous faudra davantage de Runners ou beaucoup de patience.



Continuous Integration CI

■ Intégration continue (CI = Continuous Integration)

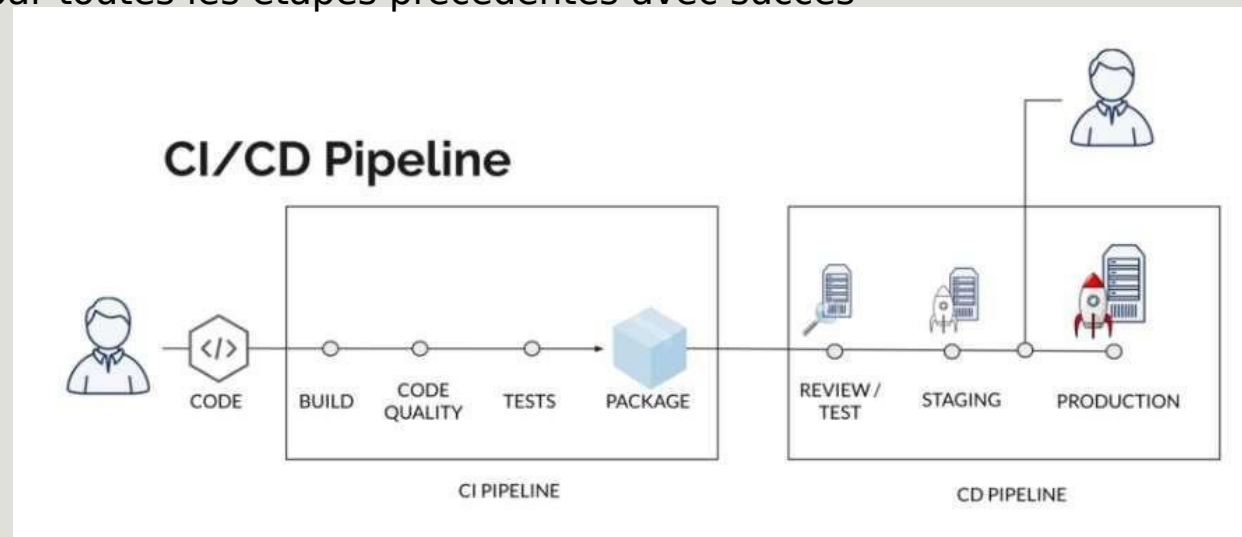
- ✓ CI est une pratique qui permet d'intégrer le code avec d'autres développeurs
- ✓ Le plus souvent, l'intégration du code est de vérifier si l'étape de construction (build stage) est toujours fonctionnelle
- ✓ Une pratique courante consiste à vérifier également si les tests unitaires (Unit Testing stage) sont toujours fonctionnels
- ✓ L'objectif du pipeline CI est de construire un **package** qui sera déployé par la suite



Continuous Delivery CD

■ Livraison continue (CD = Continuous Delivery)

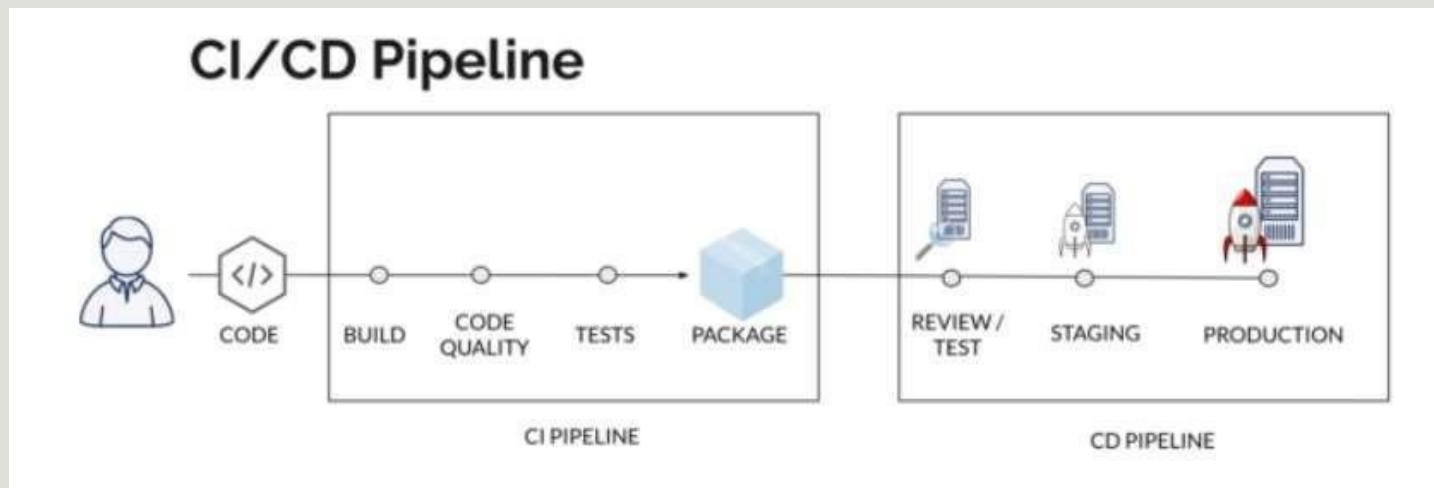
- ✓ C'est une extension de l'intégration continue
- ✓ L'objectif du CD est de prendre le package qui a été créé par le pipeline de CI et de tester son déploiement dans un **environnement de test (REVIEW et STAGING)**
- ✓ En ajoutant cette étape de pré-production et en exécutant certains tests, cela nous permet d'exécuter différents types de tests qui nécessitent la réponse de l'ensemble du système (généralement appelé tests d'acceptation)
- ✓ L'étape de déploiement en production (STAGE DEPLOY) est lancée **manuellement** si et seulement si le package a passé par toutes les étapes précédentes avec succès



Continuous Deployment CD

- **Déploiement continu (CD = Continuous Deployment)**

- Le déploiement continu est la pratique d'automatisation complète de l'ensemble des processus du pipeline CI/CD dans un environnement de **production**.
- Le package doit d'abord passer par toutes les étapes précédentes avec succès
- Aucune intervention manuelle n'est requise: **c'est automatique.**



LES JOBS

- Dans le manifeste de GitLab CI/CD vous pouvez définir un nombre **illimité** de jobs, avec des contraintes indiquant s'ils doivent être exécutés ou non.
- Voici comment déclarer un job le plus simplement possible :

```
job:1:  
  script: echo 'my first job'  
  
job:2:  
  script: echo 'my second job'
```

- Les noms des jobs doivent être uniques et ne doivent pas faire parti des mots réservés :*image, service, stages, types, before_script, after_script, variables, cache...*
- Dans la définition d'un job seule la déclaration **script** est obligatoire.

SCRIPT

- La déclaration script est donc la seule obligatoire dans un job. Cette déclaration est le cœur du job car c'est ici que vous indiquerez les actions à effectuer.
- Il peut appeler un ou plusieurs script(s) de votre projet, voire exécuter une ou plusieurs ligne(s) de commande.

```
job:script:  
  script: ./bin/script/my-script.sh ## Appel d'un script de votre projet  
  
job:scripts:  
  script: ## Appel de deux scripts de votre projet  
    - ./bin/script/my-script-1.sh  
    - ./bin/script/my-script-2.sh  
  
job:command:  
  script: printenv # Exécution d'une commande  
  
job:commands:  
  script: # Exécution de deux commandes  
    - printenv  
    - echo $USER'
```

BEFORE_SCRIPT ET AFTER_SCRIPT

- Ces déclarations permettront d'exécuter des actions avant et après votre script principal.
- Ceci peut être intéressant pour bien diviser les actions à faire lors des jobs, ou bien appeler ou exécuter une action avant et après chaque job.

```
before_script: # Exécution d'une commande avant chaque `job`  
- echo 'start jobs'
```

```
after_script: # Exécution d'une commande après chaque `job`  
- echo 'end jobs'
```

```
job:no_overwrite:
```

```
# Ici le job exécutera les action du `before_script` et `after_script` par défaut
```

```
script:  
- echo 'script'
```

```
job:overwrite:before_script:
```

```
before_script:  
- echo 'overwrite' # N'exécutera pas l'action définie dans le `before_script` par défaut
```

```
script:  
- echo 'script'
```

```
job:overwrite:after_script:
```

```
script:  
- echo 'script'
```

```
after_script:  
- echo 'overwrite' # N'exécutera pas l'action définie dans le `after_script` par défaut
```

IMAGE

- Cette déclaration est simplement l'image docker qui sera utilisée lors d'un job ou lors de tous les jobs.

```
image: alpine # Image utilisée par tous les `jobs`, ce sera l'image par défaut
```

```
job:node: # Job utilisant l'image node
```

```
  image: node
```

```
  script: yarn install
```

```
job:alpine: # Job utilisant l'image par défaut
```

```
  script: echo $USER
```

STAGES

- Cette déclaration permet de **grouper** des jobs en **étapes**.
- Par exemple on peut faire une étape de build, de test, de deployment,
- Si vous n'avez pas défini à quel stage un job appartient, il sera automatiquement attribué au stage **Test**.



stages: # Ici on déclare toutes nos étapes

- build
- test
- deploy

job:build:

stage: build #Déclare que ce `job` fait partie de l'étape build
script: make build

job:test:functional:

stage: test #Déclare que ce `job` fait partie de l'étape test
script: make test-functional

job:test:unit:

stage: test #Déclare que ce `job` fait partie de l'étape test
script: make test-unit

job:deploy:

stage: deploy #Déclare que ce `job` fait partie de l'étape deploy
script: make deploy

ONLY ET EXCEPT

- Ces deux directives permettent de mettre en place des contraintes sur l'exécution d'une tâche.
- Vous pouvez dire qu'une tâche s'exécutera uniquement sur l'événement d'un push sur master ou s'exécutera sur chaque push d'une branche sauf master.

```
job:only:master:
```

```
  script: make deploy
```

```
  only:
```

```
    - master # Le job sera effectué uniquement lors d'un événement sur la branche master
```

```
job:except:master:
```

```
  script: make test
```

```
  except:master:
```

```
    - master # Le job sera effectué sur toutes les branches lors d'un événement sauf sur la branche master
```

ONLY avec schedules

- Pour l'utilisation de **schedules** il faut dans un premier temps définir des règles dans l'interface web.
- On peut les configurer dans l'interface web de Gitlab : **CI/CD -> Schedules** et remplir le formulaire.

Schedule a new pipeline

Description

Test schedule

Interval Pattern

☒ Custom (Cron syntax) ☐ Every day (at 4:00am) ☐ Every week (Sundays at 4:00am) ☐ Every month (on the 1st at 4:00am)

0 20 * * *

Cron Timezone

Paris

Target Branch

master

Variables

RELEASE staging

Input variable key Input variable value

Activated

☒ Active

WHEN

- Comme pour les directives `only` et `except`, la directive `when` est une contrainte sur l'exécution de la tâche. Il y a quatre modes possibles :

- **on_success** : le job sera exécuté uniquement si tous les jobs du stage précédent sont passés
- **on_failure** : le job sera exécuté uniquement si un job est en échec
- **always** : le job s'exécutera quoi qu'il se passe (même en cas d'échec)
- **manual** : le job s'exécutera uniquement par une action manuelle

stages:

- build
- test
- report

job:build:

- stage: build
- script:
- make build

job:test:

- stage: test
- script:
- make test

when: on_success #s'exécutera uniquement si le job `job:build` pass

job:report:

- stage: report
- script:
- make report

when: on_failure #s'exécutera si le job `job:build` ou `job:test` ne pa

ALLOW_FAILURE

- Cette directive permet d'accepter qu'un job échoue sans faire échouer la pipeline.

```
stages:  
  - build  
  - test  
  - report  
  - clean  
  
...  
  
stage: clean  
  script:  
    - make clean  
  when: always  
  allow_failure: true # Ne fera pas échouer la pipeline  
  
...
```


TAGS

- Avec GitLab Runner vous pouvez héberger vos propres runners sur un serveur ce qui peut être utile dans le cas de configuration spécifique.
- Chaque runner que vous définissez sur votre serveur à un nom, si vous mettez le nom du runner en tags, alors ce runner sera exécuté.

```
job:tag:  
  script: yarn install  
  tags:  
    - shell # Le runner ayant le nom `shell` sera lancé
```

SERVICES

- Cette déclaration permet d'ajouter des services (container docker) de base pour vous aider dans vos jobs.
- Par exemple si vous voulez utiliser une base de données pour tester votre application c'est dans services que vous le demanderez.

```
test:functional:
  image: registry.gitlab.com/username/project/php:test
  services:
    - postgres # On appelle le service `postgres` comme base de données
  before_script:
    - composer install -n
  script:
    - codecept run functional
```

ENVIRONMENT

- Cette déclaration permet de définir un environnement spécifique au déploiement
- Il est possible de spécifier :
 - ✓ un **name**,
 - ✓ une **url**,
 - ✓ une condition **on_stop**,
 - ✓ une **action** en réponse de la condition précédente.

```
deploy:demo:
  stage: deploy
  environment: demo # Déclaration simple de l'environnement
  script:
    - make deploy

deploy:production:
  environment: # Déclaration étendue de l'environnement
    name: production
    url: 'https://blog.eleven-labs/fr/gitlab-ci/' # Url de l'application
  script:
    - make deploy
```

VARIABLES

- Cette déclaration permet de définir des variables pour tous les jobs ou pour un job précis.
- Ceci revient à déclarer des variables d'environnement.

```
variables: # Déclaration de variables pour tous les `job`  
  SYMFONY_ENV: prod  
  
build:  
  script: echo ${SYMFONY_ENV} # Affichera "prod"  
  
test:  
  variables: #Déclaration et réécriture de variables globales pour ce `job`  
    SYMFONY_ENV: dev  
    DB_URL: '127.0.0.1'  
  script: echo ${SYMFONY_ENV} ${DB_URL} # Affichera "dev 127.0.0.1"
```

- Il est aussi possible de déclarer des variables depuis l'interface web de GitLab **Settings > CI/CD > Variables** et de leur spécifier un environnement.

Variables ⓘ

Variables are applied to environments via the runner. They can be protected by only exposing them to protected branches or tags. You can use variables for passwords, secret keys, or whatever you want.

BD_PASSWORD	db_password_production	Protected	<input checked="" type="checkbox"/>	production	⊞
DB_PASSWORD	db_password_demo	Protected	<input checked="" type="checkbox"/>	demo	⊞
DB_PASSWORD	db_password_no_env	Protected	<input type="checkbox"/>	All environments	⊞
Input variable key	Input variable value	Protected	<input type="checkbox"/>	All environments	⊞

CACHE

- Le cache est intéressant pour spécifier une liste de fichiers et de répertoires à mettre en cache tout le long de votre pipeline. Une fois la pipeline terminée le cache sera détruit.
- Plusieurs sous-directives sont possibles :
 - ✓ **paths** : obligatoire, elle permet de spécifier la liste de fichiers et/ou répertoires à mettre en cache
 - ✓ **policy** : facultative, elle permet spécifier que le cache doit être récupéré ou sauvegardé lors d'un job (push ou pull).

```
job:build:
  stage: build
  image: node:8-alpine
  script: yarn install && yarn build cache:
    paths:
      - build # répertoire mis en cache
  policy: push # le cache sera juste sauvegardé, pas de récupération d'un cache existant
```

```
job:deploy:
  stage: deploy script: make deploy
  cache:
    paths:
      - build
  policy: pull # récupération du cache
```

ARTIFACTS

- Les artefacts sont un peu comme du cache mais ils peuvent être récupérés depuis une autre pipeline. Comme pour le cache il faut définir une liste de fichiers ou/et répertoires qui seront sauvegardés par GitLab. Les fichiers sont sauvegardés uniquement si le job réussit.
- Nous y retrouvons cinq sous-directives possibles :
 - ✓ **paths** : obligatoire, elle permet de spécifier la liste des fichiers et/ou dossiers à mettre en artifact
 - ✓ **name** : facultative, elle permet de donner un nom à l'artifact. Par défaut elle sera nommée artifacts.zip
 - ✓ **untracked** : facultative, elle permet d'ignorer les fichiers définis dans le fichier .gitignore
 - ✓ **when** : facultative, elle permet de définir quand l'artifact doit être créé. Trois choix possibles on_success, on_failure, always. La valeur on_success est la valeur par défaut.
 - ✓ **expire_in** : facultative, elle permet de définir un temps d'expiration

```
job:  
  script: make build  
  artifacts:  
    paths:  
      - dist  
    name: artifact:build  
  when: on_success  
  expire_in: 1 weeks
```

RETRY

- Cette déclaration permet de **ré-exécuter** le job en cas d'échec.
- Il faut indiquer le nombre de fois où vous voulez ré-exécuter le job

```
job:retry:  
  script: echo 'retry'  
  retry: 5
```

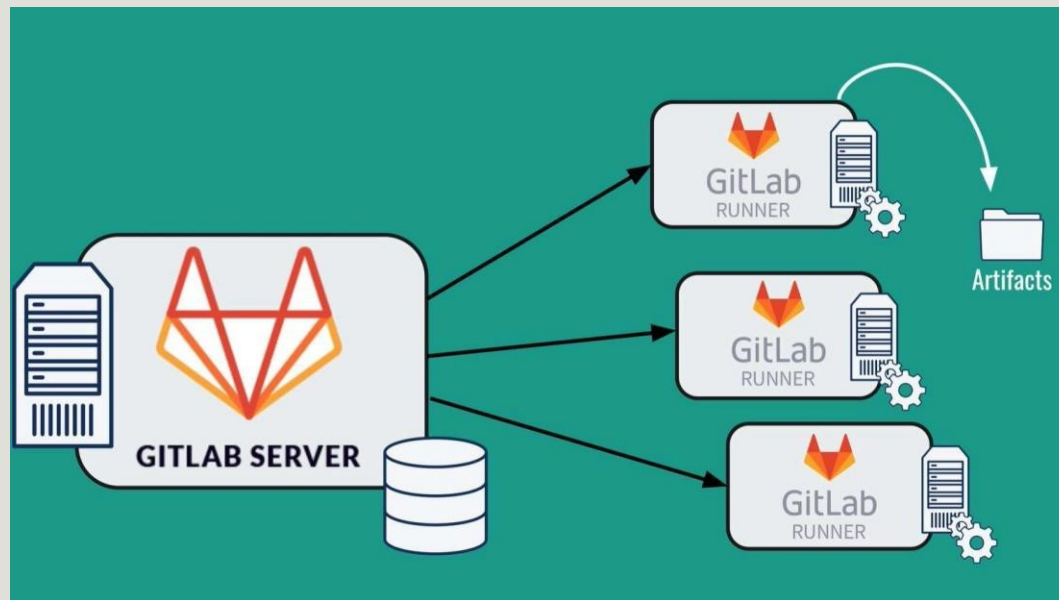
Plus loin dans l'utilisation des runners

Type des Runners

- Un Runner Gitlab-CI est un simple démon qui attend les Jobs comme vus dans le diagramme précédent.
- Une fois un Job reçu celui-ci va demander à « un exécuteur » de traiter la demande.
- Les **exécuteurs** sont des sous-processus qui vont se charger de faire les commandes (scripts) que vous avez définies dans votre gitlab-ci.
- Gitlab-CI est capable de fonctionner de différente manière :
 - ✓ SSH,
 - ✓ Shell,
 - ✓ Parallels,
 - ✓ VirtualBox,
 - ✓ Docker,
 - ✓ Docker Machine (auto-scaling),
 - ✓ Kubernetes
 - ✓ Personnalisé (Custom)

Type des Runners

- Dans Gitlab, vous pouvez créer vos propres Runners sur votre propre infrastructure en fonction de vos besoins.
- Selon le nombre de projets que vous avez ou de l'activité que vous avez sur un projet, il vous faudra davantage des Runners ou beaucoup de patience.



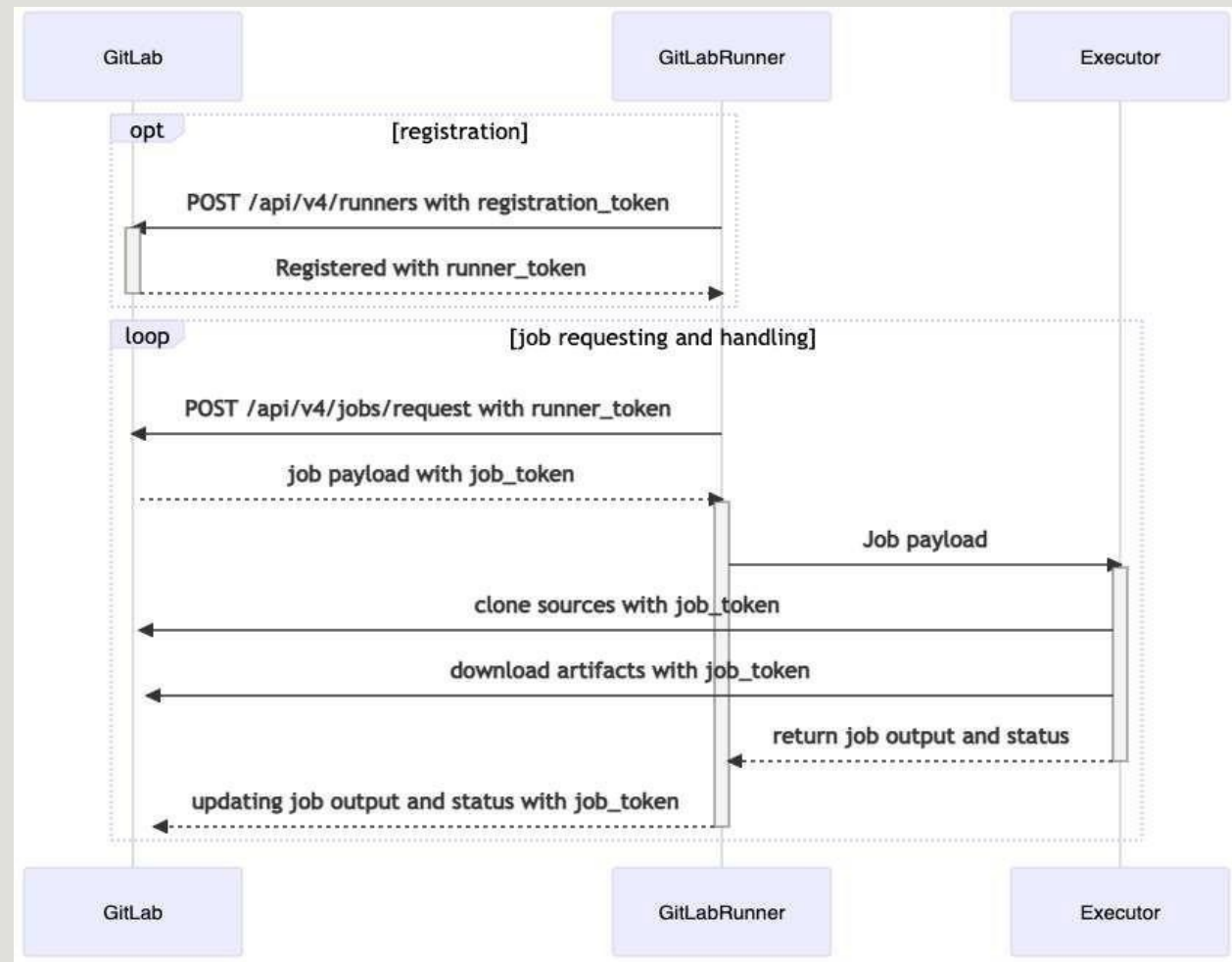
Gitlab Runner : Executor

Executor	SSH	Shell	VirtualBox	Parallels	Docker	Kubernetes	Custom
Nettoyer l'environnement de build (pour chaque build)	✗	✗	✓	✓	✓	✓	conditional
Réutiliser le clone précédent s'il existe	✓	✓	✗	✗	✓	✗	conditional
Protéger l'accès au système de fichiers du Runner	✓	✗	✓	✓	✓	✓	conditional
Migrer la machine de Runner	✗	✗	partial	partial	✓	✓	✓
Prise en charge de zéro-configuration pour les builds simultanés	✗	✗	✓	✓	✓	✓	conditional
Environnements de construction très compliqués	✗	✗	✓	✓	✓	✓	✓
Débogage des problèmes de build	easy	easy	hard	hard	medium	medium	medium

<https://docs.gitlab.com/runner/executors/>

Gitlab Runner : Executor

- L'avantage est double :
 - ✓ Pas de limite en nombre de compilation.
 - ✓ Accès à vos ressources locales pour le déploiement.



Comment choisir un exécuteur ?

- **Shell**
 - C'est le plus simple de tous.
 - Vos scripts seront lancés sur la machine qui possède le Runner.
- **Parallels, VirtualBox**
 - Le Runner va créer (ou utiliser) une machine virtuelle pour exécuter les scripts.
 - Pratique pour avoir un environnement spécifique (exemple macOS)
- **Docker**
 - Utilise Docker pour créer / exécuter vos scripts et traitement (en fonction de la configuration de votre .gitlab-ci.yml)
 - Solution la plus simple et à privilégier
- **Docker Machine (auto-scaling)**
 - Identique à docker, mais dans un environnement Docker multi-machine avec auto-scaling.
- **Kubernetes**
 - Lance vos builds dans un cluster Kubernetes.
 - Très similaire à Docker-Machine
- **SSH**
 - À ne pas utiliser. Il existe, car il permet à Gitlab-CI de gérer l'ensemble des configurations possibles.

Installation de Gitlab Server

Les recommandations de <https://docs.gitlab.com/ce/install/requirements.html>

■ Les OS supportés

- ✓ Ubuntu (16.04/18.04/20.04/22.04)
- ✓ Debian (9/10/11)
- ✓ CentOS (7/8/9)
- ✓ openSUSE Leap (15.1/15.2)
- ✓ SUSE Linux Enterprise Server (12 SP2/12 SP5)
- ✓ Red Hat Enterprise Linux (please use the CentOS packages and instructions)
- ✓ Scientific Linux (please use the CentOS packages and instructions)
- ✓ Oracle Linux (please use the CentOS packages and instructions)

■ Les OS non supportés

- X Arch Linux
- X Fedora
- X FreeBSD
- X Gentoo
- X macOS
- X Windows

Installation de Gitlab Server

Les recommandations de <https://docs.gitlab.com/ce/install/requirements.html>

- **La configuration hardware**
 - **CPU:**
 - **4 cores** est le nombre minimal de cœurs recommandé pour supporter jusqu'à 500 utilisateurs
 - 8 cores pour supporter jusqu'à 1000 utilisateurs
 - **Memory**
 - **4GB RAM** est la taille minimum de mémoire recommandé pour supporter jusqu'à 500 utilisateurs
 - 8GB RAM pour supporter jusqu'à 1000 utilisateurs
- **Prérequis**
 - Ruby 2.7 versions
 - Go 1.13 versions
 - Git 2.31.x and later is required
 - Node.js 14.x versions
 - Redis 4.0 versions

Installation de Gitlab Server

Les recommandations de <https://docs.gitlab.com/ce/install/requirements.html>

■ Les méthodes d'installation

Installation method	Description
Linux package	The official deb/rpm packages (also known as Omnibus GitLab) that contains a bundle of GitLab and the components it depends on, including PostgreSQL, Redis, and Sidekiq.
Helm charts	The cloud native Helm chart for installing GitLab and all of its components on Kubernetes.
Docker	The GitLab packages, Dockerized.
Source	Install GitLab and all of its components from scratch.
GitLab Environment Toolkit (GET)	The GitLab Environment toolkit provides a set of automation tools to deploy a reference architecture on most major cloud providers.

■ GitLab sur les fournisseurs de cloud

Cloud provider	Description
AWS (HA)	Install GitLab on AWS using the community AMIs provided by GitLab.
Google Cloud Platform (GCP)	Install GitLab on a VM in GCP.
Azure	Install GitLab from Azure Marketplace.
DigitalOcean	Install GitLab on DigitalOcean. You can also test GitLab on DigitalOcean using Docker Machine .

Installation de Gitlab Runner

Les recommandations de <https://docs.gitlab.com/runner/install/>

■ Les OS supportés

- ✓ CentOS,
- ✓ Debian
- ✓ Ubuntu
- ✓ RHEL
- ✓ Fedora
- ✓ Mint
- ✓ **Windows**
- ✓ **macOS**
- ✓ FreeBSD

Repositories

- [Install using the GitLab repository for Debian/Ubuntu/CentOS/RedHat](#)

Binaries

- [Install on GNU/Linux](#)
- [Install on macOS](#)
- [Install on Windows](#)
- [Install on FreeBSD](#)
- [Install nightly builds](#)

Containers

- [Install as a Docker service](#)
- [Install on Kubernetes](#)
- [Install using the Kubernetes Agent](#)
- [Install on OpenShift](#)

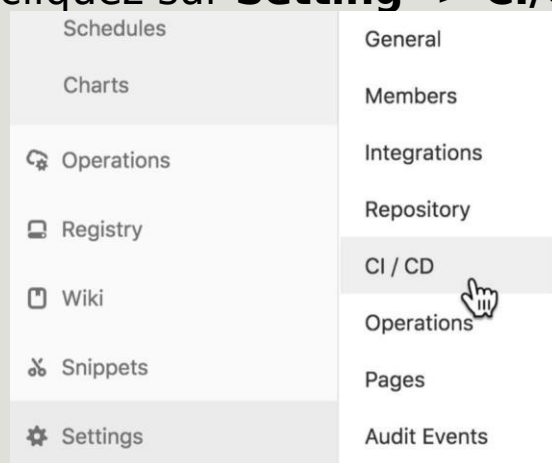
Autoscale

- [Install in autoscaling mode using Docker machine](#)
- [Install the registry and cache servers](#)

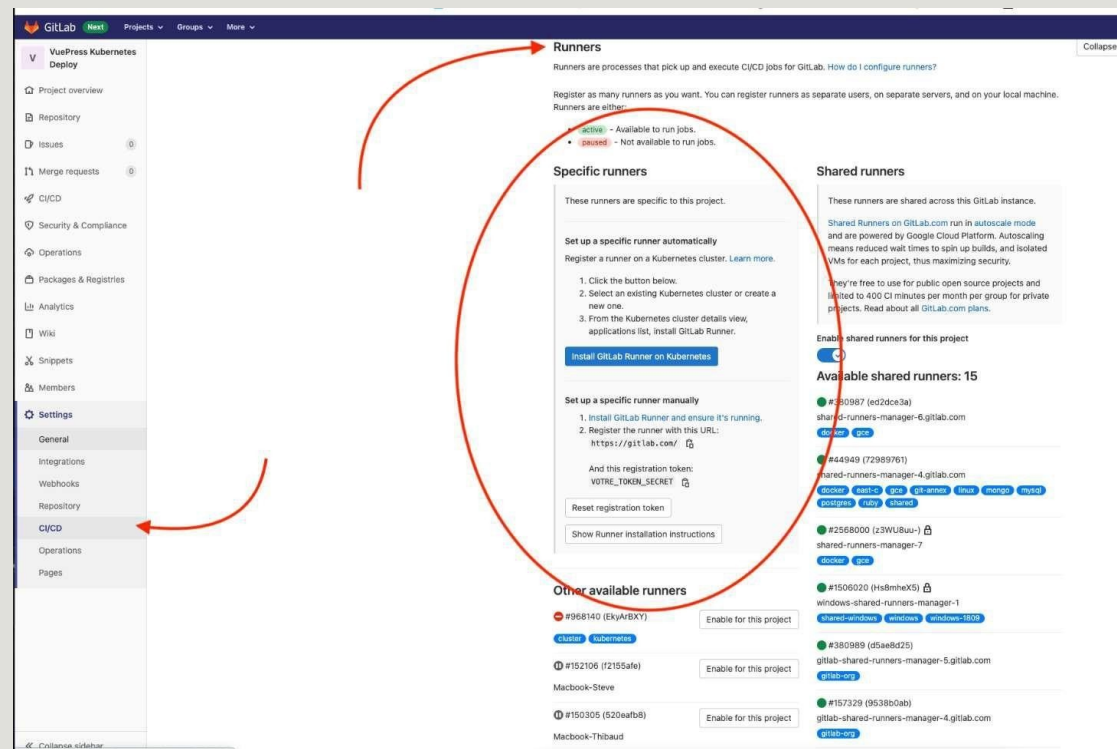
Installation d'un runner avec Docker

■ Étape 1 : Récupérer les informations des runners spécifiques (Specific runners) de votre projet

0 Dans le projet **Car assembly line**, cliquez sur **Setting -> CI/CD**



0 Dans l'interface **Runners**, cliquez sur le bouton **Expand**



Runners

Register and see your runners for this project.





Installation d'un runner avec Docker

■ Étape 2 : Enregistrement du Runner avec Gitlab-CI

- L'étape d'enregistrement n'est à réaliser qu'une seule fois. Elle a pour but d'autoriser Gitlab à communiquer avec votre runner, elle s'assure aussi que seuls vos jobs vont être lancés sur votre Runner.
- Pour enregistrer un Runner tapez la commande suivante:

```
docker run --rm -it -v $(pwd)/config:/etc/gitlab-runner gitlab/gitlab-runner register
```

```
stage@kubernetes-worker:~$ docker run --rm -it -v $(pwd)/config:/etc/gitlab-runner gitlab/gitlab-runner register
Runtime platform                                arch=amd64 os=linux pid=7 revision=4b9e985a version=14.4.0
Running in system-mode.

Enter the GitLab instance URL (for example, https://gitlab.com/):
https://gitlab.com/
Enter the registration token:
fUbjSxyFexFwKy9rBSq
Enter a description for the runner:
[100d8abcc0c0]: Démo Docker runner
Enter tags for the runner (comma-separated):

Registering runner... succeeded runner=fUbjSxy
Enter an executor: docker-ssh, shell, ssh, virtualbox, docker-ssh+machine, kubernetes, custom, docker, parallels, docker+machine:
docker
Enter the default Docker image (for example, ruby:2.6):
alpine
Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!
```



Installation d'un runner avec Docker

■ Étape 3 : Lancer le runner

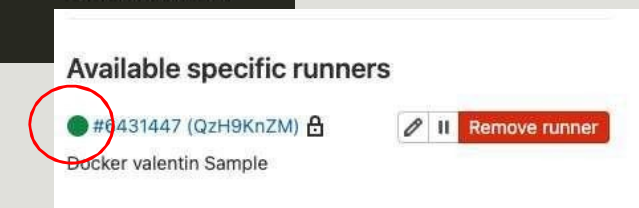
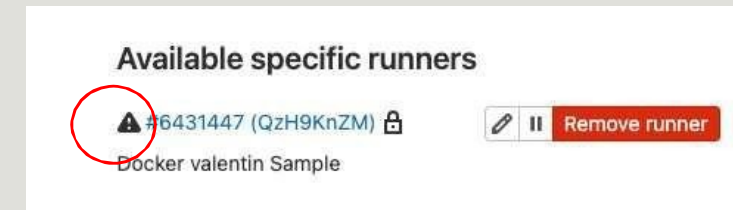
- Notre runner est maintenant connu de Gitlab, il n'est pour l'instant par contre pas encore en fonction.
- Pour le lancer on réutilise évidemment Docker, via la commande suivante :

```
docker run -d --name gitlab-runner --restart always \
-v $(pwd)/config:/etc/gitlab-runner \
-v /var/run/docker.sock:/var/run/docker.sock \
gitlab/gitlab-runner:latest
```

- Cette action lance un Container Docker visible via la commande docker ps :

```
runner ~ docker ps
CONTAINER ID   IMAGE                     COMMAND                  CREATED        STATUS        PORTS       NAMES
61569fcfdda5   gitlab/gitlab-runner:latest  "/usr/bin/dumb-init ..."  7 seconds ago  Up 6 seconds  -          gitlab-runner
runner ~
```

- Votre runner est maintenant actif sur Gitlab-CI :





Installation d'un runner avec Docker

- **Étape 4 : Configuration du runner**

- Dans la dernière version des runners, il existe un Bug au niveau de la configuration des volumes.
- Pour corriger ce Bug, éditez le fichier « **config/config.toml** »

```
sudo nano  
config/config.toml
```

- Ajouter dans la déclaration volumes la valeur suivante: **`"/var/run/docker.sock:/var/run/docker.sock"`**

```
[runners.docker]  
...  
volumes = ["/cache", "/var/run/docker.sock:/var/run/docker.sock"]
```

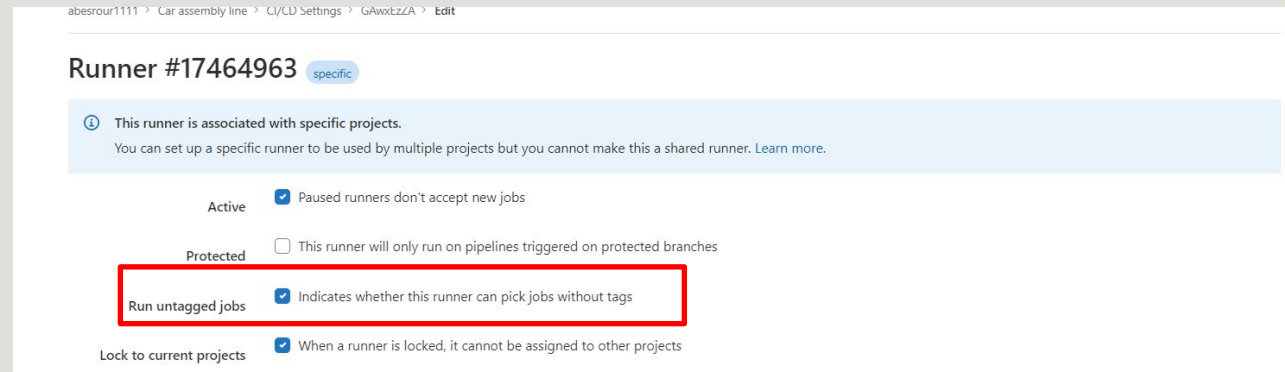
- Redémarrez le conteneur avec la commande:

```
docker restart gitlab-  
runner
```

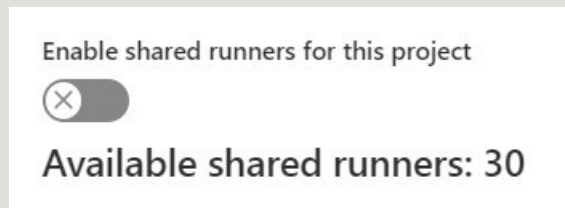


Installation d'un runner avec Docker

- **Étape 5 : Configuration du runner depuis gitlab**
- En dernier lieu, certaines configurations doivent être terminées dans le site gitlab:
 - Activer la prise en compte des jobs non taggués par le runner:



- Désactiver les runners partagés (shared runners) depuis l'interface des settings CI/CD



Merci pour votre attention

