



# GitLab CI/CD

---

David THIBAU - 2024  
david.thibau@gmail.com



# Agenda

---

## **Introduction**

- DevOps et CI/CD
- La plateforme Gitlab

## **Workflow de collaboration**

- Projets et membres
- Planification et suivi
- Repository Gitlab
- Les Merge Request
- Déclinaisons de GitlabFlow

## **Concepts Gitlab-CI**

- Introduction
- Jobs et Runners
- UI pipeline

## **Syntaxe .gitlab-ci.yml**

- Basiques Pipelines
- Principales directives
- Réutilisation
- Intégration docker
- Environnements et déploiements
- Packaging et Releasing



# Introduction

---

**DevOps et CI/CD**  
La plateforme Gitlab



# Objectif DevOps

---

- Déployer souvent et rapidement
- Automatisation complète
- Zero-downtime des services
- Possibilité d'effectuer des roll-backs
- Fiabilité constante de tous les environnements
- Possibilité de scaler sans effort
- Créer des systèmes résilients, capable de se reprendre en cas de défaillance ou erreurs



# Approche en continu

---

**A chaque ajout de valeur** dans le dépôt de source (*push*), l'intégralité des tâches nécessaires à la mise en service d'un logiciel sont essayées.

- La majeure partie des tâches sont des tests
- Des tâches de déploiement sont incluses.

En fonction de leurs succès, l'application est déployée dans les différents environnements

- Des dépôts d'artefacts
- Des environnements d'exécution (intégration, staging, production, ...)



# Pipelines

---

Les tâches de construction sont donc séquencées dans une **pipeline**.

- Une étape est exécutée seulement si les étapes précédentes ont réussi.
- Les plate-formes CI/CD ont pour rôle de
  - Démarrer les pipelines
  - Observer leur exécution
  - Rassembler les résultats des constructions (Résultat des tests, métriques)



# Distinction CI/CD

---





# Pipeline et les containers

---

Les containers même si ils ne sont pas indispensables, jouent un rôle important dans le DevOps :

- Utiliser des images pour exécuter les builds  
=> **Facilite énormément l'exploitation de la plateforme CI/CD**
- Construire et pousser des images pendant l'exécution d'une pipeline  
=> **Permet les déploiements immuables**
- Utiliser des images pour exécuter des services nécessaires à une étape de build  
=> **Test d'intégration nécessitant les services de support (BD, Broker, ...)**





# Introduction

---

DevOps et CI/CD  
**La plateforme Gitlab**



# Introduction

---

Gitlab se définit comme une plateforme DevOps complète qui inclut :

- La gestion des codes sources
- Le pilotage de projet agile
- L'exécution de pipelines de CI/CD
- La gestion des dépôts artefacts
- La gestion des environnements et infrastructure de déploiement
- La mise à disposition des bonnes pratiques DevOps



# Community vs Enterprise

---

Les 2 éditions ont le même cœur, l' *enterprise edition* ajoute du code propriétaire.

Le code propriétaire peut devenir gratuit au fur et à mesure des évolutions

Les versions payantes apportent généralement :

- Des fonctionnalités innovantes
- Des fonctionnalités avancées (Scanners de sécurité par exemple)
- Des fonctionnalités transverses au projet
- Des facilités d'intégration avec des outils
- Une installation en HA
- Du support 24h/24



# Interface utilisateur

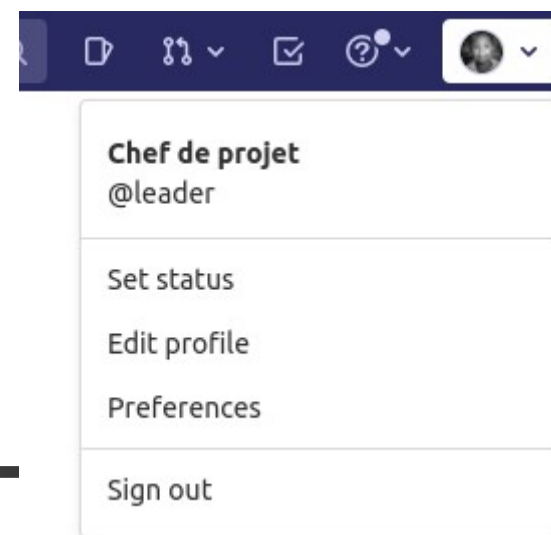
---

2 grand profils utilisateurs accèdent à la plateforme :

- Administrateur : Permet de configurer la plateforme, de gérer les utilisateurs, de configurer les runners disponibles et de configurer de façon transverse certains aspects des projets
- Utilisateur :
  - Permet de gérer son compte (infos, créidentiels, notifications, préférences)
  - Permet d'accéder à ses projets



# Menu *User Settings*



**Profile** : Édition du profil utilisateur

**Account** : Gestion de l'authentification (Possibilité d'activité le 2 factors)

**Applications** : Se connecter avec un fournisseur OAuth2

**Chat** : Mattermost si l'administrateur l'a configuré pour la plate-forme

**Personal Access Token** : Jeton représentant l'utilisateur pouvant être utilisé pour accéder à l'API Gitlab

**Emails** : Possibilité d'associer plusieurs emails au compte

**Password** : Modification mot de passe

**Notifications** : Configurer le niveau de notifications de Gitlab

**SSH Keys** : Pouvoir accéder au dépôt en ssh et sans mot de passe

**GPG Keys** : Pouvoir signer des tags

**Preferences** : Personnalisation de l'UI

**Active Sessions** : Les sessions actives (Navigateur loggés avec le compte)

**Authentication Log** : Journal des authentifications



# Mise en place clés ssh

---

La mise en place des clés *ssh* permet de pouvoir interagir avec le dépôt de source sans avoir à fournir de mot de passe.

2 étapes :

- Créer une paire de clé privé/publique
- Fournir la clé publique à Gitlab via l'interface web



# Mise en place

---

- Environnement Linux :

```
ssh-keygen -t ed25519 -C "email@example.com"
```

Ou

```
ssh-keygen -o -t rsa -b 4096 -C "email@example.com"
```

- Copier le contenu de la clé publique (\*.pub) dans l'interface Gitlab
- Tester avec :

```
ssh -T git@gitlab.com
```



# Workflows de collaboration

---

## **Projets et Membres**

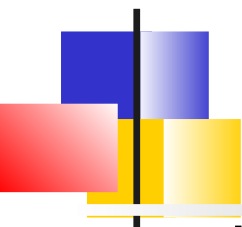
Planification et suivi

Repository Gitlab

Les MergeRequest

GitlabFlow et ses déclinaisons





# Projets

Un projet *Gitlab* a vocation à être associé à un dépôt de source *Git*

Par défaut, tous les utilisateurs *Gitlab* peuvent créer un projet

3 visibilité sont possibles pour un projet :

- **Public** : Le projet peut être cloné sans authentification.
- **Interne** : Peut être cloné par tout utilisateur authentifié.
- **Privé** : Ne peut être cloné et visible seulement par ses membres  
Les projets d'entreprise sont en général privé



# Membres

---

Les utilisateurs peuvent être affectés à des projets, ils en deviennent **membres**

Un membre a un rôle qui lui donne des permissions sur le projet :

- **Guest** : Créer un ticket
- **Reporter** : Obtenir le code source
- **Developer** : Push/Merge/Delete sur les branches non protégée, Merge request sur les autres branches
- **Maintainer** : Administration de l'équipe, Gestion des branches protégés ou non, Labels,
- **Owner** : Créateur du projet, a le droit de le supprimer



# Groupes

---

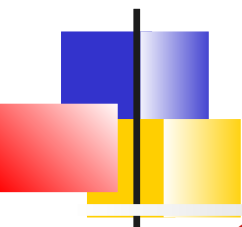
Afin de faciliter la gestion des projets et des membres associés, il est possible de définir des **groupes de projets**.

Tous les projets du groupe hériteront des configurations (Visibilité, membres, ...)

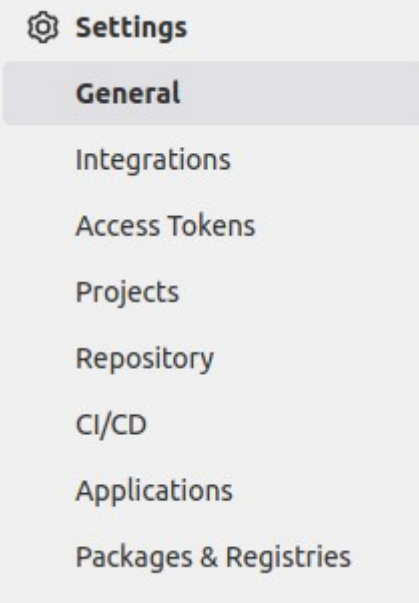
Il sera possible de visualiser toutes les issues et Merge Request des projets du groupe

Les groupes peuvent être hiérarchiques

*Attention : Il est dangereux de déplacer un projet existant dans un autre groupe*



# Menu Groupe Settings



**Group Information → Members** : Ajout de membres

**General** : Nom et visibilité

**Integration** : Intégration à des outils tierces (Slack, JIRA, ...)

**Access Token** : Jeton d'accès à l'API concernant les projets du groupe

**Projects** : Projets du groupe

**Repository** : Jetons permettant à des applications tierces de cloner le dépôt, récupérer des artefacts stockés dans Gitlab, nom de la branche par défaut

**CI/CD** : Définition de variables, de runners, activation/désactivation de AutoDevOps

**Applications** : Fournisseur OAuth2

**Package & Registries** : Définition de dépôts d'artefacts, de proxy des dépendances



# Création de projet

---

La création de projet peut se faire à partir de la home page ou de la page d'un groupe

Il peut s'agir :

- D'un projet vierge
- D'un projet à partir d'un gabarit contenant déjà certains fichiers
- En important un projet d'un autre dépôt Git

Lors de la création, il faut définir :

- Un nom
- Un *project slug* qui donnera lieu à une URL d'accès (pas de caractères spéciaux)
- La visibilité
- Si le dépôt Git associé au projet doit être initialisé avec un fichier README

# Création projet vierge à partir d'un groupe



## Create blank project

Create a blank project to house your files, plan your work, and collaborate on code, among other things.

New project › Create blank project

### Project name

### Project URL



### Project slug

Want to house several dependent projects under the same namespace? [Create a group](#).

### Project description (optional)

### Visibility Level

☒  Private

Project access must be granted explicitly to each user. If this project is part of a group, access will be granted to members of the group.

### Project Configuration

☒ Initialize repository with a README

Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.



# Menu Projet

**Projects** : Activité, Labels et membres

**Repository** : Navigateur de fichiers, Commits, branches, tags, historique, comparaison, statistiques sur les fichiers du projet

**Issues** : Gestion des issues, tableau de bord Kanban

**Merge requests** : Travaux en cours

**CI/CD** : Historique d'exécution des pipelines

**Security & compliance** : Rapports sur les détections de vulnérabilités

**Deployments** : Gestion des environnements de déploiement

**Monitor** : Information de surveillance du projet

**Infrastructure** : Cluster Kubernetes associés, Plateforme serverless, Historique des changements Terraform

**Packages et registries** : Accès aux dépôts d'artefacts

**Wiki** : Documentation annexe

**Snippets** : Bouts de code

**Settings** : Configuration

## D Delivery Service

Project information

Repository

Issues 0

Merge requests 0

CI/CD

Security & Compliance

Deployments

Monitor

Infrastructure

Packages & Registries

Analytics

Wiki

Snippets

Settings



# Menu Projet → Settings

## **General :**

- Nom, Classification Topic, Avatar,
- Visibilité projet, Configuration des features (menus accessibles)
- Merge request : Configuration des fusions de branches
- Badges,
- Service Desk : Utilisateurs pouvant envoyer des issues par mail
- Advanced : Suppression, déplacement de projet, ...

**Integrations** : Intégration application tierces

**Webhook** : Alternative à Integration. Permet d'envoyer un webhook à une application tierce

**Access Token** : Jeton d'accès pour l'API projet

**Repository** : Branche par défaut, branches et tags protégées, dépôt miroir, Clé et jetons permettant d'accéder aux dépôts et aux packages, Nettoyage du dépôt

**CI/CD** : Configuration général pipelines, AutoDevOps, Runners, politique de rétention des artefacts, Jeton de déclenchement, ...

**Monitor** : Configuration du monitoring

**Usage Quotas** : Définition de quotas de stockage





# Workflows de collaboration

---

Projets et Membres  
**Planification et suivi**  
Repository Gitlab  
Les MergeRequest  
GitlabFlow et ses déclinaisons



# Issues et Milestones

Gitlab permet de s'adapter à chaque méthodologie agile via les **issues**, les **milestones** et les **epics** dans la version payante

Une **issue** peut ainsi représenter :

- Une user story
- Un demande d'évolution
- Une déclaration de bug
- Un idée d'amélioration
- Une tâche technique

Elles sont généralement affectés à des **milestones** qui peuvent représenter :

- Une release
- Un sprint
- Une date de livraison
- ....

Les **Epics** permettent de rassembler des issues provenant de différents projets

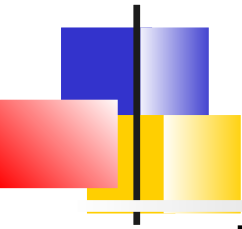


# Visualisation des issues

---

Les issues peuvent être visualisées via :

- Une **liste** : Toutes les issues du projet avec possibilité de filtrer ou faire des actions par lots (bulk)
- Un **board** : Tableau de bord façon Kanban, permettant de glisser/déposer les issues dans des colonnes représentant le statut de l'issue
- **Epic** : Vision transversale aux projets des issues partageant un thème, un milestone,



# Labels

---

Les labels jouent un rôle très important dans Gitlab

Ce sont des petits libellés colorés qui permettent de tagger les objets Gitlab :

Affecter des labels aux issues permet :

- De catégoriser les issues
- De filtrer les listes d'issues
- De créer les tableaux de bord (boards)



# Usage des Labels

---

Gitlab propose des labels par défaut mais il est possible de configurer ses propres labels

Un label peut être défini au niveau

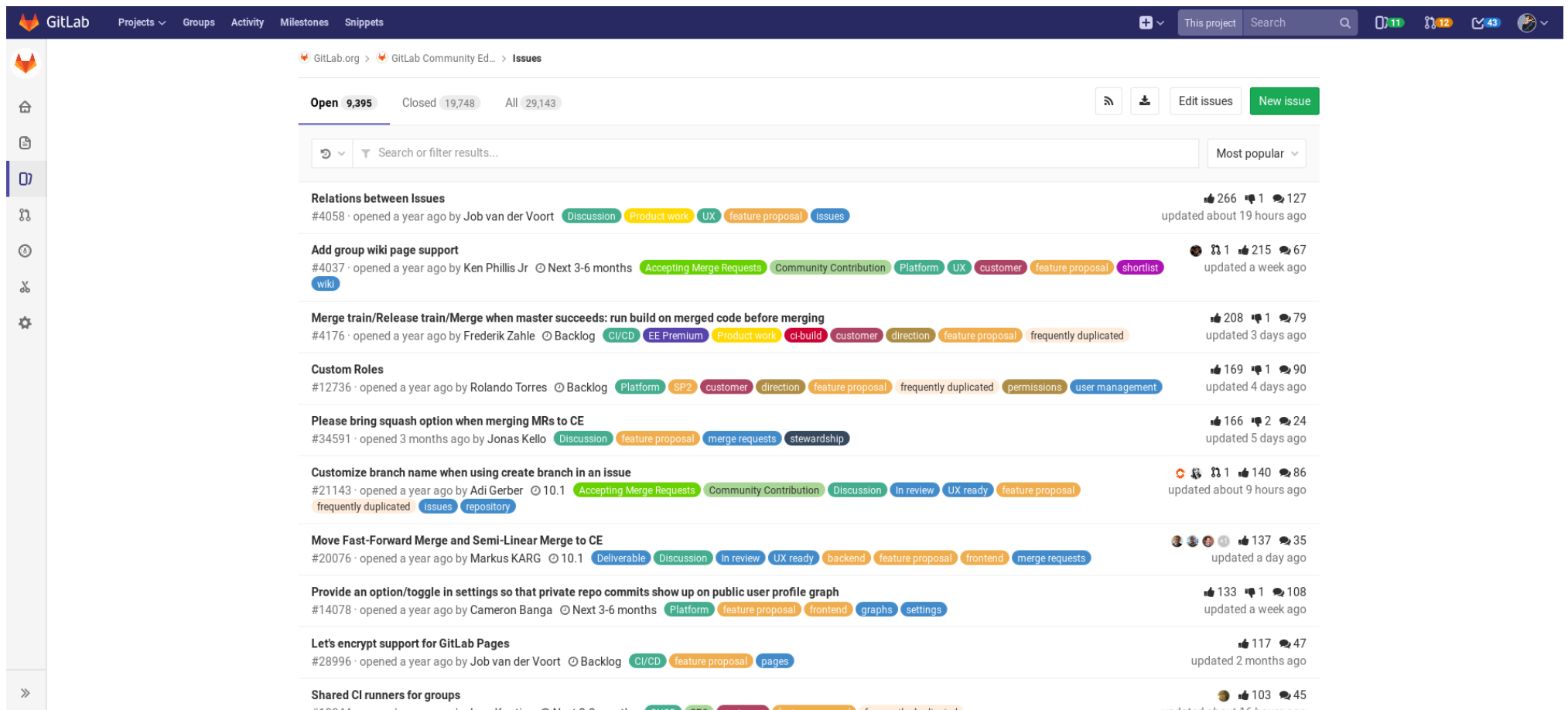
- Groupe : **Group information > Labels.**
- ou projet : **Project information > Labels.**

Plusieurs labels peuvent être associés à la même issue

On peut donc créer des labels permettant :

- De typer (Bug, Idée, RFC, User Story, ...)
- Indiquer le domaine concerné (Front-end, Back-end, CI/CD,...)
- Indiquer un statut (*Review*, *Duplicate*, ...)
- ...

# Exemple : Liste d'issues avec labels



The screenshot displays the GitLab web interface for the 'GitLab Community Edition' project. The top navigation bar includes links for Projects, Groups, Activity, Milestones, and Snippets. The main content area shows a list of issues, each with a title, description, and a set of labels. The issues are sorted by 'Most popular'.

**GitLab** Projects Groups Activity Milestones Snippets

GitLab.org > GitLab Community Ed... > Issues

Open 9,395 Closed 19,748 All 29,143

Search or filter results... Most popular

**Relations between Issues**

#4058 · opened a year ago by Job van der Voort · Discussion · Product work · UX · feature proposal · issues · 266 likes · 1 comment · 127 replies · updated about 19 hours ago

**Add group wiki page support**

#4037 · opened a year ago by Ken Phillis Jr · Next 3-6 months · Accepting Merge Requests · Community Contribution · Platform · UX · customer · feature proposal · shortlist · wiki · 1 assignee · 215 likes · 67 replies · updated a week ago

**Merge train/Release train/Merge when master succeeds: run build on merged code before merging**

#4176 · opened a year ago by Frederik Zahle · Backlog · CI/CD · EE Premium · Product work · ci-build · customer · direction · feature proposal · frequently duplicated · 208 likes · 1 comment · 79 replies · updated 3 days ago

**Custom Roles**

#12736 · opened a year ago by Rolando Torres · Backlog · Platform · SP2 · customer · direction · feature proposal · frequently duplicated · permissions · user management · 169 likes · 1 comment · 90 replies · updated 4 days ago

**Please bring squash option when merging MRs to CE**

#34591 · opened 3 months ago by Jonas Kello · Discussion · feature proposal · merge requests · stewardship · 166 likes · 2 comments · 24 replies · updated 5 days ago

**Customize branch name when using create branch in an issue**

#21143 · opened a year ago by Adi Gerber · 10.1 · Accepting Merge Requests · Community Contribution · Discussion · In review · UX ready · feature proposal · frequently duplicated · issues · repository · 1 assignee · 140 likes · 86 replies · updated about 9 hours ago

**Move Fast-Forward Merge and Semi-Linear Merge to CE**

#20076 · opened a year ago by Markus KARG · 10.1 · Deliverable · Discussion · In review · UX ready · backend · feature proposal · frontend · merge requests · 137 likes · 35 replies · updated a day ago

**Provide an option/toggle in settings so that private repo commits show up on public user profile graph**

#14078 · opened a year ago by Cameron Banga · Next 3-6 months · Platform · feature proposal · frontend · graphs · settings · 133 likes · 1 comment · 108 replies · updated a week ago

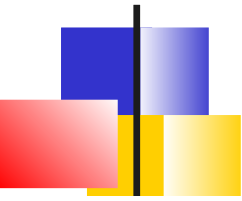
**Let's encrypt support for GitLab Pages**

#28996 · opened a year ago by Job van der Voort · Backlog · CI/CD · feature proposal · pages · 117 likes · 47 replies · updated 2 months ago

**Shared CI runners for groups**

#10244 · opened a year ago by Jean Kertier · Next 3-6 months · CI/CD · SP2 · customer · feature proposal · frequently duplicated · 103 likes · 45 replies · updated about 16 hours ago

# Exemple board



13 To do

- Suppression Formation  
#236 opened by dthibau  
bug
- Déployer sous forme de conteneur Docker  
#226 opened by dthibau
- Implémenter Cache SpringBoot  
#232 opened by dthibau  
enhancement
- Etude - plateforme Kubernetes d'OVH  
#81 opened by dthibau  
enhancement
- Utiliser OpenAPI pour simplifier les développements front-end  
#83 opened by dthibau  
enhancement Tek
- URL permettant de tester toutes les notifications  
#32 opened by dthibau
- Créer un système de tag dans PLBSI V1 et V2  
#61 opened by Vincent-PLB  
Reminder  
2.8
- Donner de la visibilité sur les cours CPF et action co sur la vue Intervenants Manager

2 In progress

- Inclure des tests automatisés des systèmes dépendants de la base plbsi dans la pipeline Jenkins  
#240 opened by dthibau  
enhancement Tek
- Partenaires : Ajouter un onglet "Contexte" (on le renommera ultérieurement si nécessaire)  
#251 opened by Vincent-PLB  
enhancement  
Partenaire 2.7.3

1 A corriger

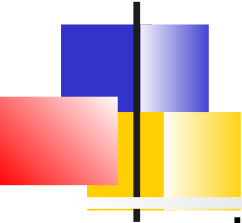
- Déplacer une formation trop loin dans une catégorie fait planter le module de réorganisation des formations  
#201 opened by Vincent-PLB  
bug

6 Recette

- Partenaires : Ajouter un encart "Délégation" pour Aurore  
#252 opened by Vincent-PLB  
enhancement  
Partenaire 2.7.3
- Les blocs Remarques du Commerce peuvent-elles afficher la date, l'heure et l'auteur de la dernière modification ?  
#95 opened by Vincent-PLB  
enhancement  
Partenaire 2.7.3
- Partenaires : Augmenter la limite de caractères sur les blocs de texte libre  
#249 opened by Vincent-PLB  
enhancement  
Partenaire 2.7.3
- Système de tooltip  
#255 opened by dthibau  
enhancement  
Partenaire 2.7.3
- Prochain Inter confirmé : ne plus prendre en compte les sessions à 1 participant  
#238 opened by Vincent-PLB  
Kibana  
Partenaire 2.7.3
- Ajouter les blocs remarques dans l'export excel

88 Done

- Certaines formation partenaire référencent des formations qui n'existent plus  
#254 opened by dthibau  
BD invalid Tek  
Partenaire 2.7.3
- Monitoring plbsi  
#225 opened by dthibau  
Tek  
2.7
- Upgrader la version de Java  
#49 opened by dthibau  
Tek  
2.7
- Créer les bases du Suivi des Offres dans PLBSI V2  
#120 opened by Vincent-PLB  
wontfix
- Mettre en place une rotation pour plbsi.log  
#67 opened by dthibau  
Tek
- Créer une maquette pour la page Historique  
#158 opened by Vincent-PLB  
duplicate  
2.5
- En consultation, le texte du champ



# Champs d'une issue à la création

---

Une issue comporte de nombreux champs qui sont pour la plupart optionnels. Les principaux sont :

- Titre : On peut forcer des gabarits
- Types : *Issue* ou *Incident*
- Description : Texte riche
- Assignee : Les personnes impliquées
- Due date
- Milestone
- Labels

Une issue peut être créée par tous les membres du projet et même par les utilisateurs si on active la fonctionnalité

***ServiceDesk***





# Collaboration autour de l'issue

---

De nombreuses fonctionnalités de collaboration sont proposées autour de l'issue :

- Threads de discussion et notifications/alertes
- Workflows (Changement de labels/statut)
- Association d'une issue à une Merge Request, et donc aux autres objets associés à la MR:
  - Aux modifications de code, aux commits
  - A la revue de code
  - Aux pipelines, aux résultats des tests automatisés
  - Aux environnements de déploiement pour réceptionner l'issue (Review Apps)



# Autres fonctionnalités

---

**Issues liées** : Permet d'associer une issue à une autre (Travail préliminaire, contexte, dépendance, doublon)

**Crosslinking** : Liens vers des objets référençant l'issue.  
(Commit, Autre Issue ou Merge Request)

- Par exemple un commit

`git commit -m "this is my commit message. Ref #xxx"`

Fermeture automatique : Possibilité de fermer les issues automatiquement après un merge request

Gabarits : Créer des issues à partir de gabarits

Édition en mode bulk

Import/Export d'issues

API Issues



# Workflows de collaboration

---

Projets et Membres

Planification et suivi

**Repository Gitlab**

Les MergeRequest

GitlabFlow et ses déclinaisons



# Particularités *Gitlab*

---

On peut interagir avec les dépôts GitLab via :

- **l'UI Gitlab en uploadant des fichiers par exemple**
- **Par l'éditeur Web en ligne** ou l'intégration VSCode
- en **ligne de commande.**

GitLab supporte des langages de **markup** pour les fichiers du dépôt (Extension .md) et certains champs de l'interface. n

Lorsqu'un fichier **README** ou index est présent, son contenu est immédiatement rendu (sans ouverture du fichier) lorsque l'on accède au projet. D'autres fichiers ont des particularités, exemple CONTRIBUTING.md

**Verrouillage** de fichier : Empêcher qu'un autre fasse des modifications sur le fichier pour éviter des conflits.

Gitlab utilise des hooks qui peuvent afficher des messages d'assistance

Accès aux données via **API**. Exemple :

```
GET /projects/:id/repository/tree
```



# Clonage d'un repo

---

Plusieurs options pour cloner un dépôt :

- Via la ligne de commande :
  - En https, peut nécessiter de saisir à chaque fois son username/mot de passe.
  - En ssh, après avoir déposé sa clé publique
- Via Gitlab UI
  - Ouverture automatique du projet dans Xcode, VisualCode ou IntelliJ IDEA



# Commits

---

- Messages :
  - **Skip pipelines**: Si le mot-clé **[ci skip]** est présent dans le commit, la pipeline de GitLab ne s'exécute pas.
  - **Cross-link issues/MR**: Si on mentionne une issue ou un MR dans un message de commit (#xxx), Un lien sera proposé par Gitlab.
- Lorsque c'est possible, Gitlab proposer d'effectuer via l'interface un *cherry-pick* ou un *revert* d'un commit particulier
- Possibilité de signer les commits via GPG



# Analytiques proposées

---

GitLab détecte les langages de programmation utilisé et affiche ces infos sur la page Projet

Dans le menu *Analyze*, il offre un graphique dédié aux projets :

- Langages de programmation détectés
- Statistiques sur les commits
- Certains graphiques peuvent y être ajouté par les pipelines. Ex : Couverture de code

Un graphique dédié aux contributeurs



# Vues proposées

---

*Settings → Contributors* : Les contributeurs au code

*Repository → Commits* : Historique des commits

*Repository → Branches/Tags* : Gestion des branches et des tags

*Repository → Graph* : Vue graphique des commits et merge

*Repository → Charts* : Affiche les langages détectés par Gitlab et des statistiques sur des commits





# Les branches

---

Dans Gitlab, les branches peuvent avoir des caractéristiques particulières :

- Peut être la branche par défaut
- Peut être une branche protégée
- Peut être une branche dont le nom répond à un pattern défini par le mainteneur

Le mainteneur est donc responsable de :

- Définir la branche par défaut
- Définir des règles sur le nom des branches et les protections associées



# Branche par défaut

---

A la création de projet, *GitLab* positionne ***main/master*** comme branche par défaut.

- Peut-être changé *Settings* → *Repository* (au niveau projet ou administrateur)

La branche par défaut a certaines particularités :

- Elle ne peut pas être détruite
- C'est une branche protégée
- C'est en général la branche cible des MergeRequest
- Lors de l'accès aux sources, c'est cette branche qui est affichée



# Branches protégées

Le mainteneur administre les branches protégées via le menu *Settings* → *Protected branches* ou *Settings* → *Branch rules*

Des permissions sont associées à une branche protégée :

- ***Allow to Merge*** : Qui peut y fusionner une autre branche.
- ***Allow to Push*** : Qui peut y faire un push
- ***Force Push*** : Les personnes ayant le droit push peuvent elles faire des force push<sup>1</sup>.

Des *wildcards* sont possibles pour protéger des branches en fonction de leurs noms *Ex* :

*\*-stable, production/\**

1. Pousser une branche locale sans être sur la même ligne que la branche distante



# Création de branche

---

Plusieurs façons de créer des branches avec Gitlab :

- A partir du menu (*Repository* → *Branches*), Il est possible d'indiquer la branche de départ
- **A partir d'une issue**, en créant une Merge Request  
Par défaut, la branche est créé à partir de la branche par défaut  
Elle est dédiée à la résolution de l'issue et est généralement supprimée lorsque l'issue est résolue
- En commande en ligne, en poussant une branche locale vers le dépôt



# Workflows de collaboration

---

Projets et Membres

Planification et suivi

Repository Gitlab

**Les MergeRequest**

GitlabFlow et ses déclinaisons



# Patterns de workflow

---

C'est à l'équipe de définir le workflow de collaboration adapté à son environnement.

Cependant, certains patterns de collaboration sont documentés :

- Projets OpenSource (Linux, Github, ...) : **Workflow avec intégrateur** basé sur les *pull-request*
- Éditeur logiciel avec maintenance concurrente de plusieurs releases : **Atlassian Gitflow**
- Projet DevOps avec déploiement continu : **GitlabFlow** basé sur les merge-request

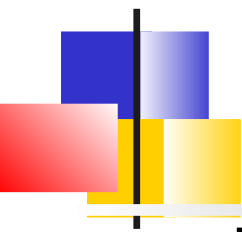


# Merge Request gitlab

Gitlab propose d'organiser le travail autour d'une **Merge Request**.

A chaque démarrage, d'une nouvelle tâche,

- 1) Le responsable de la tâche crée une Merge Request  
La Merge Request définit une branche source et une branche cible
- 2) Les collaborateurs effectuent des modifications de code.  
La merge request regroupe toutes les informations nécessaires à l'évaluation et à la réalisation de la tâche.
- 3) Une ou plusieurs personnes désignées sont responsables de déterminer quand la tâche est terminée.  
A la fin de la tâche, les travaux sont fusionnés dans la branche cible



# Création de Merge Request

---

Plusieurs façons pour créer une MR :

- A partir d'une issue, la branche source reprend le nom de l'issue.  
Par défaut, elle part de la branche par défaut et à vocation à être fusionné dans la branche par défaut.
- A partir d'une branche existante, la MR reprend le nom de la branche.  
Par défaut la branche cible est la branche par défaut
- Directement et dans ce cas, on choisit librement la branche source et la branche cible





# Cycle de vie d'une MR

---

1. Lors de sa création la MR a un statut **Draft** indiqué dans son titre.
2. Après un certain nombre de commits et de push, les responsables de la tâche jugent qu'ils ont terminés. Ils active le lien **Mark as Ready**
4. Le mainteneur est assisté par la MR pour juger de la fin réelle du travail. Il peut alors :
  - Accepter la MR : L'ensemble des commits sont alors fusionnés dans la branche cible. La MR a le statut **Merged**
  - Refuser la MR : Il peut indiquer les motifs de son refus. Les responsables de la tâche continuent leur travail
  - Fermer la MR : Cela équivaut à abandonner les travaux. La MR a le statut **Closed**



# Propriétés d'une MR

---

En dehors de son titre, une MR peut avoir défini :

- Une description rich text
- Une ou plusieurs<sup>u</sup> personnes assignées
- Un ou plusieurs<sup>u</sup> reviewers
- Un milestone
- Un ou plusieurs labels
- Des options de merge :




# Onglets d'une MR

---

L'accès à sa vue détaillé fait apparaître 4 onglets :

- *Activité* : Les commentaires et les threads. Les évènements comme les push ou les revues de code
- *Commits* : *L'accès aux commits et aux patches associés.*
- *Pipeline* : Les pipelines CI/CD et leurs résultats
- *Changes* : Les changements sur les fichiers résultants des commits.

# Resolve "Implémenter l'interaction avec banques"

[Edit](#)[Code](#) 

 Merged **mouhamadou bamba badjinka** requested to merge `6-implementer-l-interacti...` into `main` 5 days ago

[Overview](#) **0**[Commits](#) **6**[Pipelines](#) **7**[Changes](#) **17**[Add a to do](#)

Closes [#6 \(closed\)](#)



**Pipeline #1231758956 failed**

Pipeline failed for `18dc461e` on `6-implementer-l-interaction-avec-banques` 3 days ago



8✓

Approval is optional



Merged by  **David THIBAU** 3 days ago

[Revert](#)[Cherry-pick](#)

Merge details

- Changes merged into `main` with [2253fd14](#).
- Deleted the source branch.
- Closed [#6 \(closed\)](#)



**Pipeline #1231763196 failed**

Pipeline failed for `2253fd14` on `main` 3 days ago



## Activity

[All activity](#) 

- **mouhamadou bamba badjinka** requested review from [@badjinka](#) 5 days ago
- **mouhamadou bamba badjinka** assigned to [@dthibau](#) 5 days ago
- **mouhamadou bamba badjinka** added 1 commit 5 days ago
  - [af44fec](#) - inter action avec banque
  - [Compare with previous version](#)
- **mouhamadou bamba badjinka** added 1 commit 5 days ago
  - [5f8ab2cf](#) - inter action avec banque V2
  - [Compare with previous version](#)
- **David THIBAU** added 1 commit 5 days ago
  - [65d26fd4](#) - Configuration eureka et config
  - [Compare with previous version](#)


**Assignee**

[Edit](#)

**David THIBAU**

**Reviewer**

[Edit](#)

**mouhamadou bamba badjinka** 

**Labels**

[Edit](#)

None

**Milestone**

[Edit](#)

None

**Time tracking**



No estimate or time spent

**2 Participants**





# Commentaires et discussions

---

Des **commentaires** peuvent être associés aux MR

- Soit au niveau général
- Soit au niveau d'un commit particulier

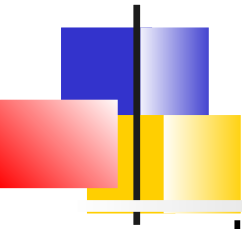
Un commentaire peut être transformé en **discussion/thread**.

Une discussion/thread regroupe plusieurs échanges et a un statut

- La discussion démarre avec un statut **unresolved**
- Elle se termine avec le statut **resolved**

Il est possible de

- voir toutes les discussions non résolues
- De déplacer les discussions non résolues vers une issue
- D'empêcher la fusion, si une discussion est non résolue  
(**Project → Settings → General → MR**)



# Revue de code

---

Une revue de code consiste à effectuer plusieurs commentaires liés à des lignes de code.

Lors d'une revue de code, le reviewer commence par créer des commentaires visibles uniquement par lui.

Lorsqu'il est prêt, il publie l'ensemble des commentaires en une fois.

- 1) Sélectionner l'onglet **Changes** de la MergeRequest
- 2) Sélectionner l'**icône de commentaire** en face du patch
- 3) Ecrire le 1<sup>er</sup> commentaire et activer le bouton **Start Review**
- 4) Faire d'autres commentaires et activer le bouton **Add to review**
- 5) A la fin, activer le bouton **Submit the review**

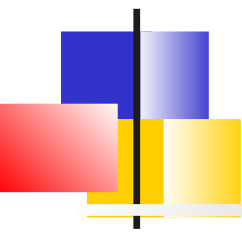


# Configuration des MR

---

Dans le menu ***Project → Settings → General***, le mainteneur peut configurer les merge request

- Méthode de fusion :
  - Commit de merge
  - Commit de merge avec possibilité de rebasing si conflit
  - Pas de commit merge seulement des fast-forward. Si conflit possibilité de rebasing
- Options de fusion : Résolution automatique des discussions, hooks, Suppression de la branche source cochée par défaut
- Squash des commits (perte de l'historique des commits de la branche source)
  - Autoriser, Favoriser ou empêcher
- Vérifications avant la fusion
  - La pipeline doit s'être exécutée avec succès
  - Tous les discussions doivent être résolues
- Gabarits des messages de Merge



# Workflows de collaboration

---

Projets et Membres

Planification et suivi

Repository Gitlab

Les *MergeRequest*

**GitlabFlow et ses déclinaisons**

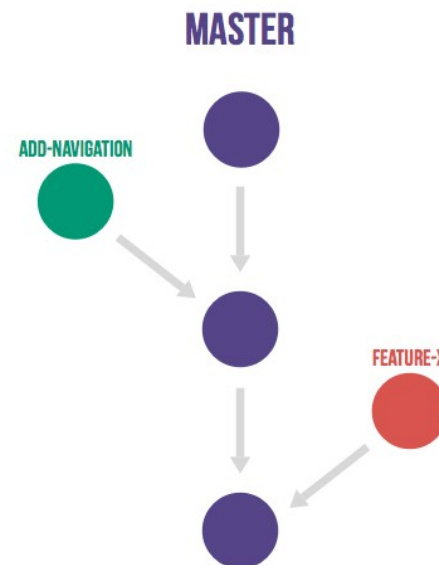




# Gitlab Flow

Dans sa configuration par défaut, Gitlab propose une workflow de collaboration simple orienté correction d'issue.

Ce type de workflow peut convenir à des projet DevOps simple





# Déclinaisons DevOps

---

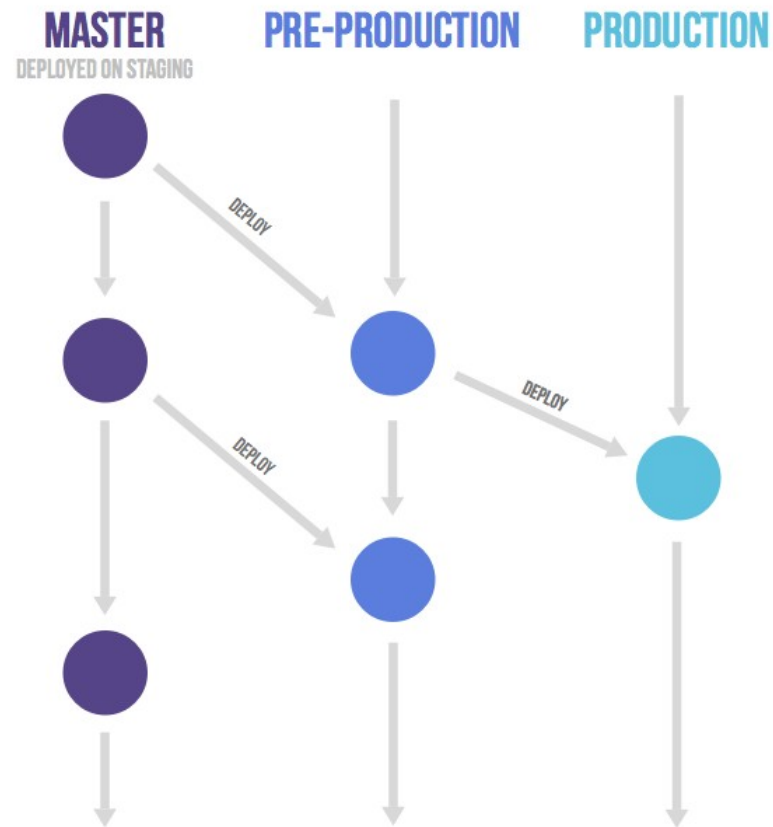
Il est cependant possible de modifier la configuration par défaut en créant d'autres branches et en identifiant les moyens de mettre à jour ces branches.

Dans un projet DevOps<sup>1</sup>, on peut introduire par exemple :

- **integration** : Branche protégée en amont de master qui sert aux déploiements dans un environnement d'intégration. Les branches de feature sont fusionnées dans intégration
- **qa/préprod** : Branche protégée dédiée à un environnement de recette.  
Quand le mainteneur (ou la pipeline CI) le décide la branche principale est intégrée dans cette branche et un déploiement s'effectue en recette
- **production** : Chaque merge à partir de la pré-prod ou de la branche par défaut est taggée et correspond à une livraison dans l'environnement de production

1. Un projet où l'on n'a pas besoin de maintenir les précédentes versions

# Déclinaison avec qa





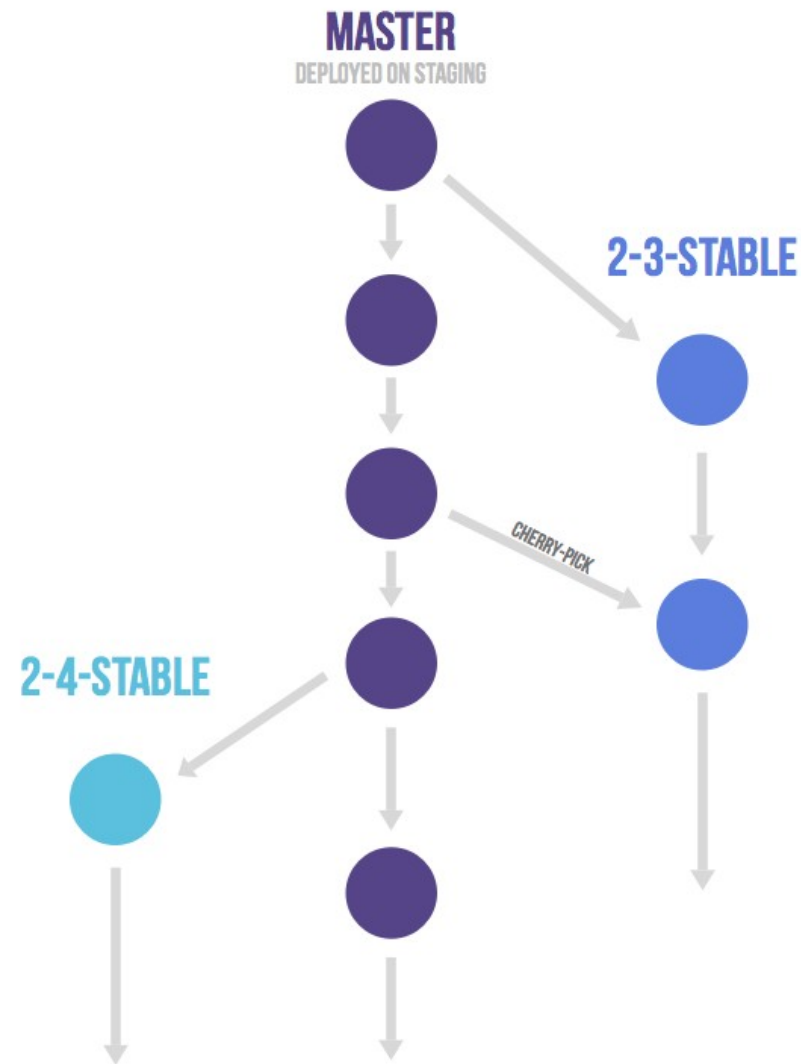
# Déclinaisons Release

---

Un workflow tel GitFlow d'Atlassian peut également être mis en place et faire apparaître d'autres branches

- ***release-candidate*** : Branche en amont d'une branche de release permettant de faire des commits préparant la release.  
La MR associé a comme cible une branche de release particulière
- ***release*** : Branche de release. Chaque merge est taggée et correspond à une distribution de release. Les Bug fixes à posteriori sont repris de *master* via des cherry-picks dans les branches de release impactées

# Branches de releases (Gitflow)





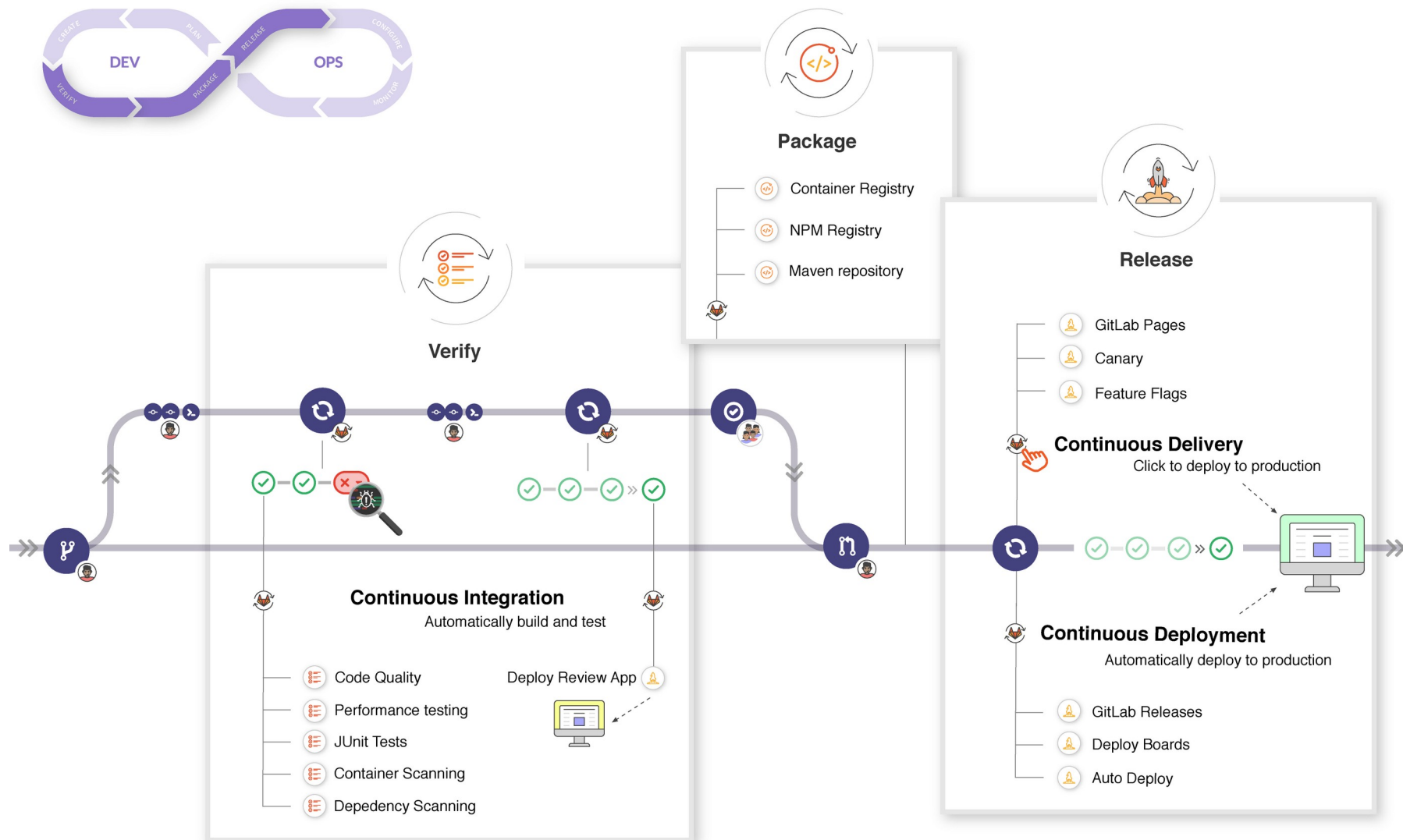
# Fusions entre branche

---

La fusion entre branches peut s'effectuer via :

- Les MRs : le code source est modifié.  
Ex :  
Feature → master  
RC → Release
- Les pipelines CI/CD. La fusion s'exécute après des tests
  - Automatiques
  - Ou manuels

# Exemple pipeline





# Concepts Gitlab-ci

---

## **Jobs et Runners** UI Pipelines





# Runner

---

Les jobs de builds sont exécutés via des **runners**

**GitLab Runner** est une application qui s'exécute sur des machines distinctes et qui communique avec Gitlab.

Un runner peut être

- dédié à un projet à un groupe de projet.  
Il est défini par le mainteneur de Projet
- ou peut être partagé par tous les projets.  
Il est alors défini par l'administrateur



# Type de Runners

---

Un **Runner** peut être une machine virtuelle, une machine physique, un conteneur docker ou un pod dans un cluster *Kubernetes*.

Le type de runner conditionne les pipelines qu'il peut exécuter

GitLab et les Runners communiquent via une API  
=> La machine du runner doit avoir un accès réseau au serveur Gitlab.

Pour disposer d'un runner :

- Il faut l'installer
- Puis l'enregistrer soit comme runner partagé (administrateur) soit comme runner dédié au projet



# Installation GitlabRunner

---

L'installation s'effectue :

- Via des packages  
Debian/Ubuntu/CentOS/RedHat
- Exécutable MacOS ou Windows
- Comme service Docker
- Auto-scaling avec Docker-machine
- Via Kubernetes



# Enregistrement

---

Pour enregistrer un runner, il faut obtenir un token via l'UI de gitlab

La commande ***gitlab-runner register*** exécutée dans l'environnement du runner démarre un assistant posant les question suivantes :

- L'URL de *gitlab-ci*
- Le token
- Une description
- Une liste de tags
- L'exécuteur (shell, docker, ...)
- Si docker, l'image par défaut pour construire les builds



# Exécuteurs

---

Les exécuteurs d'un runner ont une influence sur les jobs que le runner peut exécuter :

- **Shell** : Toutes les dépendances du projet doivent être pré-installées sur le runner (git, npm, jdk, ...)
- **Virtual Machine** : Nécessite Virtual Box ou Parallels. Les outils projet sont pré-installés sur la VM
- **Docker** : Permet d'exécuter des builds dans une image docker fournie par le projet.  
D'autres services docker peuvent être démarrés pendant le build, ex :  
BD pour des tests d'intégration
- **Docker-machine** : Des Vms avec docker installé sont créés à la demande et détruite après le job.
- **Kubernetes** : Utilisation d'un cluster Kubernetes. Via l'API, le runner crée des pods (machine de build + services)
- **ssh** : Peu recommandé, exécute le build via ssh sur une machine distante



# Affectation d'un runner et tags

---

Lorsqu'une pipeline doit être exécutée, Gitlab affecte un runner pour le job.

- Il choisit de préférence un runner dédié au projet
- Chaque runner peut également avoir une liste de tags et une pipeline peut définir également des tags  
=> Gitlab recherche alors le runner ayant les mêmes tags que le job

Si Gitlab ne trouve pas de runner adapté, la pipeline ne démarre pas (état stuck)



# Concepts Gitlab-ci

---

Jobs et Runners  
**UI Pipelines**



# Editeur

---

Un éditeur en ligne de *.gitlab-ci.yml* est disponible

Il permet une validation de la syntaxe

***Repository → Files → .gitlab-ci.yml → Pipeline Editor***

Des gabarits sont également disponibles pour la plupart des technologies

***Repository → New File → Apply Template → .gitlab-ci.yml  
→ <techno>***





# Exécution des pipelines

---

Les pipelines s'exécutent automatiquement à chaque push sur une branche

Dans le cas d'une MR, elle se déclenche à chaque modification de la MR

Elles peuvent être également planifiées pour s'exécuter à des intervalles réguliers via l'UI ou l'API

*Settings → CI/CD → Schedules*

Enfin, elles peuvent être démarrées manuellement par l'UI



# Tableau de bord d'exécution

Status	Pipeline	Triggerer	Stages	
<div>⏸ blocked</div> <div>⌚ 00:13:00</div>	<div>Adding probe</div> <div><a href="#">#123967683</a>  master  302815b0 </div> <div>latest Auto DevOps</div>		<div>✓ ✓ ! ✓ ⚙ ⏸</div>	<div>▶ ▼ ↺ ⏹ ⬇ ▼</div>
<div>✅ passed</div> <div>⌚ 00:21:12 📅 3 years ago</div>	<div>Bonjour</div> <div><a href="#">#123967122</a>  1-changer-hello-en-bonjour  d9111678 </div> <div>latest Auto DevOps</div>		<div>✓ ✓ ✓ ✓ ! ✓</div>	<div>▶ ▼ ↺ ⬇ ▼</div>

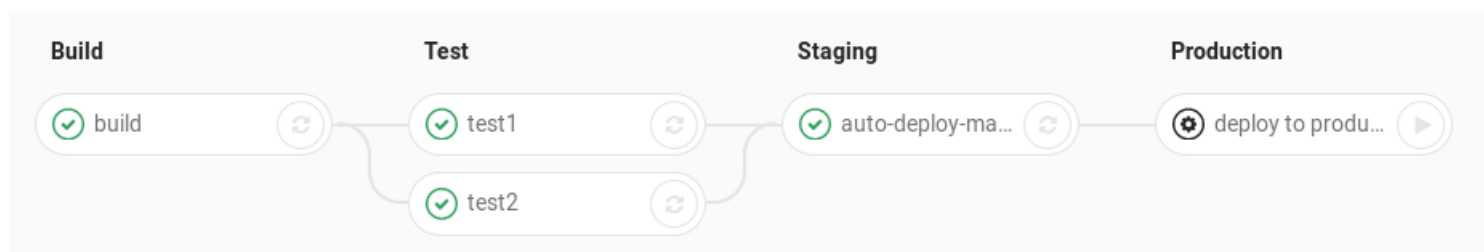
Un tableau de bord permet de voir les dernières exécutions de pipeline, leurs status, le commit, le déclenchement et l'exécution des tâches

Une barre de boutons permet de continuer ou redémarrer la pipeline et de télécharger les artefacts



# Visualisation d'une pipeline

Le détail d'une pipeline est affichée graphiquement.



Il est possible de visualiser la sortie standard de chaque tâche en la sélectionnant

De déclencher une tâche manuelle



# Syntaxe gitlab-ci.yml

---

## **Basiques *.gitlab-ci.yml***

Principales Directives

Réutilisation

Intégration docker

Environnements et déploiements

Packaging et Releasing



# Spécification de la pipeline

---

La spécification du job et de ses différentes phases peuvent être faits de différentes façons :

- **AutoDevOps** : Mode par défaut.  
Gitlab choisit la pipeline en fonction du projet.  
Nécessite des runners docker
- Fichier **.gitlab-ci.yml** à la racine du projet  
Des gabarits selon les piles technologies sont proposés par Gitlab



# *AutoDevOps*

---

*AutoDevOps* est une pipeline adapté à toutes les technologies.

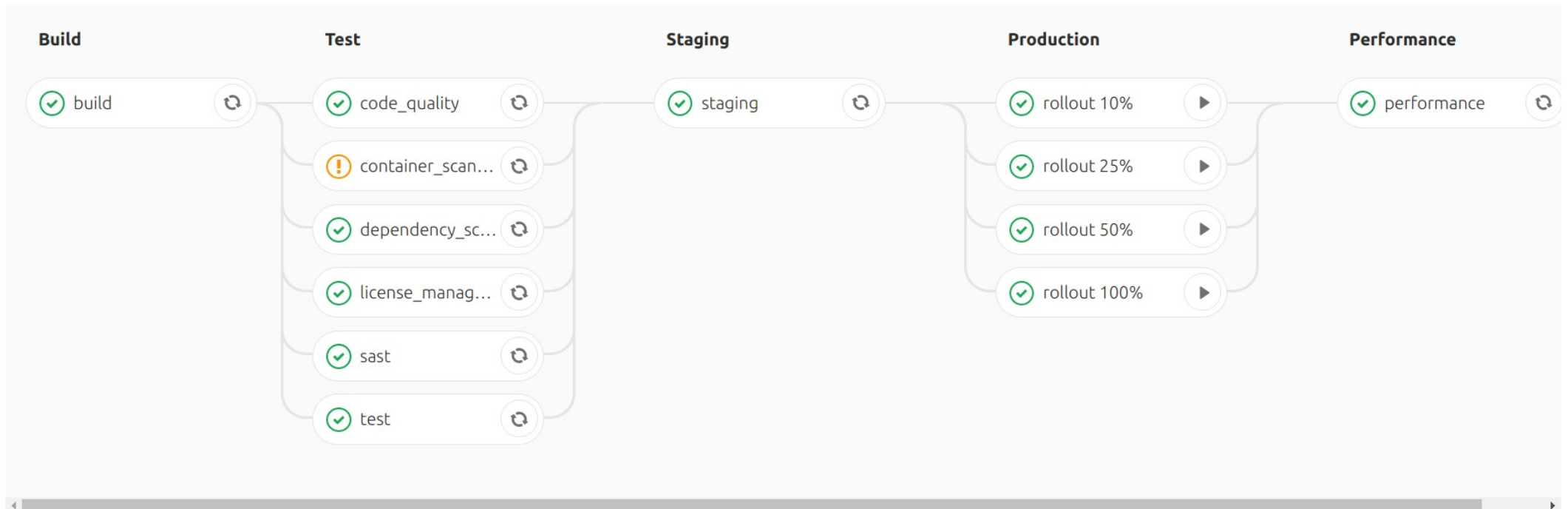
## PréRequis :

- Docker pour builder, tester construire le conteneur
- Kubernetes : Pour les déploiements

## Phases :

- Sur toutes les branches :
  - Build : Compilation, packaging
  - Test : Tests, Analyse qualité, Scan sécurité, licences
- Sur branche de feature
  - Review : Déploiement sur un environnement dédié à la branche
- Sur la branche par défaut
  - Staging : Déploiement dans un environnement de staging
  - Production : Roll-out manuel de la production
  - Performance : Test de performance en prod

# AutoDevops sur branche main





# Jobs / phases / tâches

---

Le fichier *.gitlab-ci.yml* définit des **jobs**.

Les jobs sont associés à des **phases** exécutées séquentiellement.

- Les jobs d'une même phase sont exécutés en parallèle
- Par défaut, si une phase échoue, les phases suivantes ne sont pas exécutées.

Un job est constitué d'une ou plusieurs **commandes shell** exécutées séquentiellement sur la machine de build (runner, image docker ou autre) .

Les *jobs* peuvent récupérer ou sauvegarder des résultats par le biais du serveur Gitlab





# stages

---

La directive ***stages*** permet de définir les phases séquentielles de la pipeline

Elle se place dans la partie globale de *.gitlab-ci.yml*

- Si elle n'est pas présente, les phases par défaut sont : *.pre, build, test, deploy, .post*

- Exemple :

stages:

- build
- test
- deploy



# Jobs

---

Chaque job est défini par un nom et est associé à un stage. (Si le stage n'est pas précisé, le job appartient au stage test)

Les tâches exécutés par le job sont définies par les directives :

- ***script*** : Décrit les commandes du job
- ***before-script***, ***after-script*** : Les commandes exécutées avant/après chaque job.



# Exemple

---

## **stages:**

- Build
- Test
- Staging
- Production

## build:

**stage: Build**

script: make build dependencies

## test1:

**stage: Test**

script: make build artifacts

## test2:

**stage: Test**

script: make test

## auto-deploy-ma:

**stage: Staging**

script: make deploy

## deploy-to-production:

**stage: Production**

script: make deploy



# Directive needs

---

Introduit dans Gitlab 12.2, la directive ***needs*** permet d'exprimer des dépendances entre jobs sans prendre en compte les stages.

Dans l'exemple suivant, le job *linux:rspec* s'exécute dès que le job *linux:build* est terminé (même si *mac:build* n'est pas terminé)

```
linux:build:
  stage: build
  script: echo "Building linux..."
mac:build:
  stage: build
  script: echo "Building mac..."
linux:rspec:
  stage: test
  needs: ["linux:build"]
  script: echo "Running rspec on linux..."
```



# Contexte des directives

---

En fonction de leur niveau (indentation *yml*), les directives s'appliquent à l'ensemble des jobs ou à une job particulier.

Les principales directives globales sont :

- **default** : Valeurs par défauts des jobs. Inclut entre autres :
  - **image** : L'image docker utilisée pour le build (Nécessite un runner docker)
  - **services** : Les services devant être démarrés avant le build
  - **tag** : Tags du jobs permettant de l'affecter au bon runner
  - **timeout**, **retry**, **cache**, ...
- **include** : Permet d'inclure un autre fichier yml
- **stages** : La définition des phases
- **workflow** : Permet de contrôler le comportement de la pipeline global (règles d'annulation, de création, ...)



# Variables

---

Le job peut accéder à un ensemble de variables :

- Fournies **systématiquement** par GitLab : Id d'issue, commit ID, branch ...
- **Définies par l'UI** au niveau transverse projet (administrateur), au niveau groupe ou au niveau projet.
- **Définies dans *.gitlab-ci.yml*** au niveau global ou job

Une variable peut être configurée comme étant masquée ou protégée<sup>1</sup>

L'accès se fait via la notation ***\${variable}***

Ex :

```
docker login -u "$CI_REGISTRY_USER" -p  
"$CI_REGISTRY_PASSWORD" $CI_REGISTRY
```

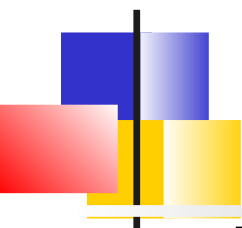
*1. Accessible seulement des branches protégées*



# Principales variables prédéfinies

---

- **`CI_PROJECT_*`** : Informations sur le projet
- **`CI_REPOSITORY_URL`** : L'url du dépôt
- **`CI_DEFAULT_BRANCH, CI_COMMIT_REF_PROTECTED`** : Branche par défaut, flag si la branche est protégée
- **`CI_COMMIT_AUTHOR, CI_COMMIT_BRANCH, CI_COMMIT_DESCRIPTION, CI_COMMIT_MESSAGE, CI_COMMIT_REF_SLUG, CI_COMMIT_SHA`** : Tout sur le commit
- **`CI_BUILDS_DIR`** : Le répertoire de build
- **`CI_RUNNER_*`** : Infos sur le runner
- **`CI_API_V4_URL, CI_API_GRAPHQL_URL`** : Urls des APIs gitlab
- **`CI_DEBUG_TRACE, CI_DEBUG_SERVICES`** : Flags de debug
- **`CI_JOB_NAME_SLUG, CI_JOB_STAGE, CI_JOB_STATUS, CI_JOB_ID, CI_JOB_IMAGE`** : Informations sur le Job



# Pipeline de Merge Request

---

Lorsqu'une Merge Request est créée ou mise à jour. Gitlab déclenche une pipeline qui fusionne les commits de la branche source avec la branche cible.

Cela permet d'anticiper les problèmes pouvant arriver lors de la fusion.

Ce ne sont pas les mêmes variables qui sont disponibles, en particulier :

- `GIT_MERGE_REQUEST_EVENT_TYPE` : Indique l'évènement lié à la MR
- La variable `GIT_COMMIT_BRANCH` n'est pas disponible car on est pas sur une branche





# Variables d'une MR

---

Les autres variables disponibles lors d'une MR :

- **`CI_MERGE_REQUEST_TITLE`, `CI_MERGE_REQUEST_ASSIGNEES`, `CI_MERGE_REQUEST_LABELS`, `CI_MERGE_REQUEST_MILESTONE`** : Infos sur la MR
- **`CI_MERGE_REQUEST_APPROVED`** : Flag si c'est une approbation
- **`CI_MERGE_REQUEST_SOURCE_BRANCH_NAME`, `CI_MERGE_REQUEST_SOURCE_BRANCH_PROTECTED`, `CI_MERGE_REQUEST_TARGET_BRANCH_NAME`, `CI_MERGE_REQUEST_TARGET_BRANCH_PROTECTED`** : Noms des branches et flag si elles sont protégées
- **`CI_MERGE_REQUEST_*_SHA`** : Les différents SHA1 des branches sources et cibles
- ...



# Syntaxe gitlab-ci.yml

---

Basiques *.gitlab-ci.yml*

**Principales Directives**

Réutilisation

Intégration docker

Environnements et déploiements

Packaging et Releasing



# cache

---

Cette directive à spécifier une liste de fichiers ou répertoires qui sera reprise entre 2 pipelines successives

Les caches sont :

- Partagés entre les pipelines et les jobs.
- Par défaut, non partagés entre les branches protégées et non protégées.
- Restaurés avant les artefacts. (Voir + loin)
- Limité à un maximum de quatre caches différents.



# Sous-directive de cache

---

Les sous-directives possibles sous cache sont :

- ***paths*** : Spécifie les chemins à cacher
- ***key*** : Fournit un identifiant pour le cache  
Tous les jobs qui utilisent la même clé de cache utilisent le même cache, y compris dans différents pipelines.
- ***untracked*** : Cache les fichiers non suivis par git
- ***unprotect*** : Permet le partage de cache entre branche protégée et non protégée
- ***when*** : Condition sur le cache
- ***policy*** : Permet de spécifier un cache en lecture seule



# Examples

---

```
cache:
  unprotect: true
  paths:
    - .m2/repository
---
cache-job:
  script:
    - echo "This job uses a cache definied for the branch."
  cache:
    key: binaries-cache-$CI_COMMIT_REF_SLUG
    paths:
      - binaries/
---
faster-test-job:
  stage: test
  cache:
    key: gems
    paths:
      - vendor/bundle
    policy: pull
  script:
    - echo "This job script uses the cache, but does not update it."
    - echo "Running tests..."
```



# Artifacts

---

Les ***artifacts*** sont une liste de fichiers et répertoires attachés à un job terminé.

Les artifacts sont uploadés sur le serveur à la fin du job.

Ils sont téléchargeable (tar.gz) via l'UI

Ils sont conservés 1 semaine (par défaut)

Ils sont téléchargés par défaut par les jobs en aval

pdf:

```
script: xelatex mycv.tex
```

```
artifacts:
```

```
  paths:
```

```
    - mycv.pdf
```

```
  expire_in: 1 week
```



# Sous-directives artifacts

---

Sous la directive artifacts peuvent être précisés :

- **paths** : Liste des chemins et fichiers à uploadés
- **exclude** : Pattern des fichiers à exclure de paths
- **expire\_in** : Surcharge le délai d'expiration par défaut
- **expose\_as** : Remonte l'artefact dans l'UI des MRs
- **name** : Surcharger le nom par défaut (*artifacts*)
- **access** : Détermine qui peut avoir accès à l'artefact  
(*all|developer|none*)
- **reports** : Permet d'indiquer le type d'artefact utilisé par des gabarits Gitlab.  
Exemple : *junit*
- **untracked** : Limite les artefacts aux fichiers non-suivis par git
- **when** : Condition d'upload de l'artefact.  
Par exemple : *on\_failure*



# Exemples

---

```
artifacts:
  expose_as: 'Exécutable'
  paths:
    - binaries/
    - .config
  exclude:
    - binaries/**/*.*
---
job:
  artifacts:
    access: 'developer'
---
```





# Publier les résultats de test

---

Gitlab permet de publier les rapports d'exécution des tests aux format JUnit

Il suffit de placer la directive ***artifacts:reports:junit*** et d'indiquer le chemin du rapport dans *.gitlab-ci.yml*

```
ruby:
  stage: test
  script:
    - bundle exec rspec --format progress --format RspecJunitFormatter --out rspec.xml
  artifacts:
    when: always
    paths:
      - rspec.xml
  reports:
    junit: rspec.xml
```



# Réutilisation des artefacts

---

La directive ***dependencies*** permet de contrôler les artefacts que l'on veut récupérer

- Soit elle indique le nom des jobs dont on veut récupérer les artefacts
- Soit elle indique une liste vide pour indiquer que l'on ne veut pas récupérer les artefacts

Si, les dépendances ne sont pas disponibles lors de l'exécution du job, il échoue.



# Réutilisation des artefacts (2)

---

```
build:osx:
  stage: build
  script: make build:osx
  artifacts:
    paths:
      - binaries/

build:linux:
  stage: build
  script: make build:linux
  artifacts:
    paths:
      - binaries/

test:osx:
  stage: test
  script: make test:osx
  dependencies:
    - build:osx
```



# ***GIT\_STRATEGY***

---

La variable ***GIT\_STRATEGY*** peut être positionnée dans la pipeline pour conditionner, l'interaction du runner avec le dépôt.

La variable peut prendre 3 valeurs :

- ***clone*** : Le dépôt est cloné par chaque job
- ***fetch*** : Réutilise le précédent workspace si il existe en se synchronisant ou effectue un clone
- ***none*** : N'effectue pas d'opération git, utiliser pour les taches de déploiement qui utilisent des artefacts précédemment construits



# ***GIT\_CHECKOUT***

---

La variable ***GIT\_CHECKOUT*** peut être utilisée lorsque *GIT\_STRATEGY* est définie à *clone* ou *fetch*

Elle spécifie si une extraction git doit être exécutée (true défaut)

Si false :

- ***fetch*** : Met à jour le dépôt et laisse la copie de travail sur la révision courante ,
- ***clone*** : Clone le dépôt et laisse la copie de travail sur la branche par défaut

variables:

```
GIT_STRATEGY: clone
```

```
GIT_CHECKOUT: "false"
```

script:

- git checkout -B master origin/master
- git merge \$CI\_COMMIT\_SHA



# Control Flow

---

***allow\_failure*** permet à une tâche d'échouer sans impacter le reste de la pipeline.

- La valeur par défaut est *false*, sauf pour les jobs manuels.

***retry*** permet de configurer le nombre de tentatives avant que le job soit en échec.

***tags*** : Liste de tags pour sélectionner un runner

***parallel*** : Nombre d'instances du jobs exécutés en parallèle

***trigger*** : Permet de déclencher une autre pipeline à la fin d'un job.



# Conditions

---

***when*** conditionne l'exécution d'un job principalement en fonction de son statut.

Les valeurs possibles sont :

- ***on\_success*** : Tous les jobs des phases précédentes ont réussi (défaut).
- ***on\_failure*** : Au moins un des jobs précédents a échoué
- ***always*** : Tout le temps
- ***manual*** : Exécution manuelle déclenchée par l'interface



# rules

---

La directives **rules** permet d'inclure ou d'exclure des jobs de la pipeline selon l'évaluation d'une expression.

- Les règles sont évaluées lors de la création du pipeline dans l'ordre de définition
- Lorsqu'une correspondance est trouvée, la tâche est incluse ou exclue du pipeline, selon la configuration.
- Si aucune règle match, le job n'est pas ajouté à la pipeline





# Directives rules

---

La directive **rules** accepte une liste de règle. Chacune ayant au moins un des mots-clés suivant :

- **if** : Les expressions sont évaluées en fonction de variables de la pipeline
- **changes** : Condition sur des changements sur des fichiers
- **exists** : Condition sur l'existence d'un fichier ou répertoire
- **when** : Expression sur le statut de la pipeline



# Exemple rules:if

---

job:

script: echo "Hello, Rules!"

rules:

# On interdit pour une MR d'une branche feature

# vers un autre branche que celle par défaut

- if: \$CI\_MERGE\_REQUEST\_SOURCE\_BRANCH\_NAME =~ /^feature/ &&  
\$CI\_MERGE\_REQUEST\_TARGET\_BRANCH\_NAME != \$CI\_DEFAULT\_BRANCH

when: never

# Si démarrage manuel, on autorise les échecs

- if: \$CI\_MERGE\_REQUEST\_SOURCE\_BRANCH\_NAME =~ /^feature/

when: manual

allow\_failure: true

# Le job s'exécute si c'est une fusion de MR

- if: \$CI\_MERGE\_REQUEST\_SOURCE\_BRANCH\_NAME



# Rules et variables

---

La directive rules peut également être utilisée pour positionner les valeurs de variables en fonction de condition :

```
job:
  variables:
    DEPLOY_VARIABLE: "default-deploy"
  rules:
    - if: $CI_COMMIT_REF_NAME == $CI_DEFAULT_BRANCH
      variables: # Surcharge DEPLOY_VARIABLE
        DEPLOY_VARIABLE: "deploy-production"
    - if: $CI_COMMIT_REF_NAME =~ /feature/
      variables:
        IS_A_FEATURE: "true" # Définition nouvelle variable.
  script:
    - echo "Run script with $DEPLOY_VARIABLE as an argument"
    - echo "Run another script if $IS_A_FEATURE exists"
```



# workflow

---

La directive globale **workflow** permet de contrôler l'exécution de la pipeline complète.

- **auto\_cancel** permet d'annuler une pipeline en cours d'exécution
  - **on\_new\_commit** : Annulation si un nouveau commit survient
  - **on\_job\_failure** : Nécessite une configuration de l'administrateur, permet d'annuler certains jobs de la pipeline lorsqu'un job est en échec
- **name** permet de nommer la pipeline
- **rules** permet de conditionner l'exécution en fonction des variables prédéfinies fournies par Gitlab



# Exemples

---

```
# Annule les jobs ayant la propriété interruptible à true  
# si un nouveau commit survient
```

```
workflow:
```

```
  auto_cancel:
```

```
    on_new_commit: interruptible
```

```
job1:
```

```
  interruptible: true
```

```
  script: sleep 60
```

```
job2:
```

```
  interruptible: false  # Default when not defined.
```

```
  script: sleep 60
```



# Syntaxe gitlab-ci.yml

---

Basiques *.gitlab-ci.yml*

Principales Directives

**Réutilisation**

Intégration docker

Environnements et déploiements

Packaging et Releasing



# Inclusion et Héritage

---

*Gitlab* fournit un support afin que les organisations puissent partager des pratiques entre projets.

- Il est possible d'inclure des fragments de pipeline pouvant être utilisés dans plusieurs projets
- Il est possible d'hériter de pipelines parentes et de compléter et surcharger le parent

Gitlab propose de nombreux fragments pouvant être inclus



# Inclusion

---

Le mot-clé ***include*** permet l'inclusion de fichiers YAML externes.

4 méthodes d'inclusions :

- ***local*** : Inclusion d'un fichier du dépôt
- ***file*** : Inclusion du fichier d'un autre projet
- ***template*** : Inclusion d'un template fourni par Gitlab. Le gabarit peut être surchargé
- ***remote*** : Inclusion d'un fichier accessible via URL





# Examples

---

include:

- remote:  
'https://gitlab.com/awesome-project/raw/master/.before-script-template.yml'
  - local: '/templates/.after-script-template.yml'
  - template: Auto-DevOps.gitlab-ci.yml
  - project: 'my-group/my-project'
- ref: master
- file: '/templates/.gitlab-ci-template.yml'



# Surcharge de gabarit

---

## Gabarit :

```
variables:
  POSTGRES_USER: user
  POSTGRES_PASSWORD: testing_password
  POSTGRES_DB: $CI_ENVIRONMENT_SLUG

production:
  stage: production
  script:
    - install_dependencies
    - deploy
  environment:
    name: production
    url: https://$CI_PROJECT_PATH_SLUG.
    $KUBE_INGRESS_BASE_DOMAIN
  only:
    - master
```

## Surcharge :

```
include: 'https://company.com/autodevops-
  template.yml'

image: alpine:latest

variables:
  POSTGRES_USER: root
  POSTGRES_PASSWORD: secure_password

stages:
  - build
  - test
  - production

production:
  environment:
    url: https://domain.com
```



# Extension

---

Le mot réservé ***extends*** permet à un job d'hériter d'un autre (ou plusieurs)

Le job peut surcharger des valeurs du parent. Ex :

```
tests:
  script: rake test
  stage: test
  only:
    refs:
      - branches
```

```
rspec:
  extends: .tests
  script: rake rspec
  only:
    variables:
      - $RSPEC
```



# Jobs standards

---

Gitlab fourni de nombreux jobs standards, il suffit en général d'inclure le template fourni par Gitlab.

Citons :

- Analyse statique de code  
template: Code-Quality.gitlab-ci.yml
- SAST : Static Application Security Testing  
template: Security/SAST.gitlab-ci.yml
- Secret Detection  
template: Security/Secret-Detection.gitlab-ci.yml
- ...



# Mise en place de l'analyse qualité

---

La mise en place nécessite des pré-requis :

- Une phase nommée **test** dans *.gitlab-ci.yml*
- Suffisamment d'espace de stockage

Inclure le gabarit de qualité dans *.gitlab-ci.yml*

include:

- template: Code-Quality.gitlab-ci.yml



# Syntaxe gitlab-ci.yml

---

Basiques *.gitlab-ci.yml*

Principales Directives

Réutilisation

**Intégration docker**

Environnements et déploiements

Packaging et Releasing



# Docker

---

Le mot-clé ***image*** spécifie l'image docker à utiliser pour exécuter les scripts.

Il peut être global à la pipeline ou spécifique à un job.

Par défaut, l'exécuteur utilise Docker Hub mais cela peut être configuré via *gitlab-runner/config.toml*



# Syntaxe image

---

2 syntaxes sont possibles pour image

- Si juste à spécifier le nom de l'image :  
`image: "registry.example.com/my/image:latest"`
- Si l'on veut passer d'autres options, il faut utiliser la clé `name`  
`image:`  
`name: "registry.example.com/my/image:latest"`  
`entrypoint: ["/bin/bash"]`  
`pull-policy : if-not-present`  
`docker :`  
`platform: arm64/v8`  
`user: dave`

La clé *entrypoint* équivalent à l'argument `--entrypoint` de la commande *docker*

La clé *docker* permet de passer des options à l'exécuteur *docker*





# Docker services

---

Le mot-clé ***services*** permet de démarrer d'autres containers durant le build.

Le build peut alors accéder au service via le nom de l'image (ou un alias)

services:

- tutum/wordpress:latest

alias : wordpress

Le service est accessible via *tutum-wordpress*, *tutum/wordpress*, *wordpress*



# Test du service

---

Lors de l'exécution du build, le Runner:

- Vérifie quels ports sont ouverts
- Démarre un autre conteneur qui attend que ces ports soient accessibles

Si ces tests échouent, un message apparaît dans la console :

```
*** WARNING: Service XYZ probably didn't start properly.
```



# Options pour service

---

4 options disponibles :

- ***name*** : Nom de l'image.  
Requis si l'on veut passer d'autres options
- ***entrypoint*** : L'argument *--entrypoint* de docker.  
Syntaxe équivalente à la directive *ENTRYPOINT* de docker
- ***command*** : Passer en argument de la commande docker.  
Syntaxe équivalente à la directive *CMD* de docker
- ***alias*** : Un alias d'accès dans le DNS



# Variables

---

Les variables définies dans le fichier YAML sont fournies au conteneur exécutant le service.

Exemple service Postgres :

```
services:
```

```
- postgres:latest
```

```
variables:
```

```
POSTGRES_DB: nice_marmot
```

```
POSTGRES_USER: runner
```

```
POSTGRES_PASSWORD: ""
```



# Construction d'image

---

Un scénario désormais classique du CI/CD est:

- 1) Créer une image applicative
- 2) Exécuter des tests sur cette image
- 3) Pousser l'image vers un registre distant
- 4) Déployer l'image vers un serveur

En commande docker :

```
docker build -t my-image dockerfiles/  
docker run my-image /script/to/run/tests  
docker tag my-image my-registry:5000/my-image  
docker push my-registry:5000/my-image
```



# Configuration du runner

---

3 possibilités afin de permettre l'exécution de commande *docker* :

- Avec l'exécuteur shell et une pré-installation de docker sur le runner
- Avec l'exécuteur docker et :
  - l'image docker (image contenant le client docker),
  - ainsi que le service *docker-in-docker* permettant de disposer d'un daemon docker
- Avec l'exécuteur docker, une pré-installation du démon docker sur le runner et une redirection de socket docker pour profiter du démon installé

Attention : Pour ces 3 techniques le serveur gitlab doit être accessible des containers



# Exécuteur shell

---

## 1. Enregistrer un exécuteur Shell sur le runner :

```
sudo gitlab-runner register -n \  
  --url https://gitlab.com/ \  
  --registration-token REGISTRATION_TOKEN \  
  --executor shell \  
  --description "My Runner"
```

## 2. Installer docker sur la machine hébergeant le runner

## 3. Ajouter l'utilisateur gitlab-runner au groupe docker

```
sudo usermod -aG docker gitlab-runner
```

## 4. Vérifier que gitlab-runner a accès à docker

```
sudo -u gitlab-runner -H docker info
```

## 5. Tester la pipeline :

```
before_script:
```

```
- docker info
```

```
build_image:
```

```
script:
```

```
- docker build -t my-docker-image .
```

```
- docker run my-docker-image /script/to/run/tests
```



# Docker in Docker (1)

---

Enregistrer un exécuteur docker en mode  
privilège

```
sudo gitlab-runner register -n \  
  --url https://gitlab.com/ \  
  --registration-token REGISTRATION_TOKEN \  
  --executor docker \  
  --description "My Docker Runner" \  
  --docker-image "docker:stable" \  
  --docker-privileged
```





# Docker in Docker (2)

---

Tester dans un *.gitlab-ci.yml*

image: **docker:stable**

variables:

DOCKER\_HOST: tcp://docker:2375/

DOCKER\_DRIVER: overlay2

**services:**

- **docker:dind**

before\_script:

- docker info

...



# Association de socket (1)

---

Enregistrer un runner avec une association de socket :

```
sudo gitlab-runner register -n \  
  --url https://gitlab.com/ \  
  --registration-token REGISTRATION_TOKEN \  
  --executor docker \  
  --description "My Docker Runner" \  
  --docker-image "docker:stable" \  
  --docker-volumes  
/var/run/docker.sock:/var/run/docker.sock
```



# Association de socket (2)

---

image: **docker:stable**

before\_script:

- docker info

build:

stage: build

script:

- docker build -t my-docker-image .
- docker run my-docker-image  
/script/to/run/tests



# Registre Gitlab

---

Une fois l'image construite, il est naturel de la pousser dans un registre

Gitlab dans sa version entreprise propose un registre de conteneur.

Pour l'utiliser, il faut :

- Que l'administrateur est autorisé le registre Docker  
Nécessite un nom de domaine
- De s'authentifier auprès du registre.
- Utiliser `docker build --pull` pour récupérer les changements sur l'image de base
- Faire explicitement un *docker pull* avant chaque *docker run*. Sinon, on peut être gêné par des problèmes de cache si l'on a plusieurs runner.
- Ne pas construire directement vers le tag *latest* si plusieurs jobs peuvent être lancés simultanément



# Authentification auprès du registre Gitlab

---

Si le registre hébergé par Gitlab est autorisé, 3 façons sont disponibles pour l'authentification :

- Utiliser les variables `$CI_REGISTRY_USER` et `$CI_REGISTRY_PASSWORD` qui sont des créden-tiels éphémères disponibles pour le job
- Utiliser un jeton d'accès personnel  
*User Settings → Access token*
- Utiliser le jeton de déploiement :  
*gitlab-deploy-token*



# Exemple

---

```
build:
  image: docker:stable
  services:
    - docker:dind
  variables:
    DOCKER_HOST: tcp://docker:2375
    DOCKER_DRIVER: overlay2
  stage: build
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker build -t $CI_REGISTRY/group/project/image:latest .
    - docker push $CI_REGISTRY/group/project/image:latest
```



# Syntaxe gitlab-ci.yml

---

Basiques *.gitlab-ci.yml*

Principales Directives

Réutilisation

Intégration docker

**Environnements et déploiements**

Packaging et Releasing



# Introduction

---

GitLab CI/CD est également capable de deployer sur différents environnements

Les **environnements** sont comme des tags décrivant où le code est déployé

Les **déploiements** sont créés lorsque les jobs déploient des versions de code vers des environnement  
=> ainsi chaque environnement peut avoir plusieurs déploiements

GitLab:

- Fournit un historique complet des déploiements pour chaque environnement
- Garde une trace des déploiements => On sait ce qui est déployé sur les serveurs





# Définition des environnements

---

Les environnements sont définis dans *.gitlab-ci.yml*

Le mot-clé ***environment*** indique à GitLab que ce job est un job de déploiement. Il peut être associée à une URL

=> Chaque fois que le job réussit, un déploiement est enregistré, stockant le SHA Git et le nom de l'environnement.

*Operate → Environments*

Le nom de l'environnement est accessible via le job par la variable *\$CI\_ENVIRONMENT\_NAME*



# Exemple

---

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



# Déploiement manuel

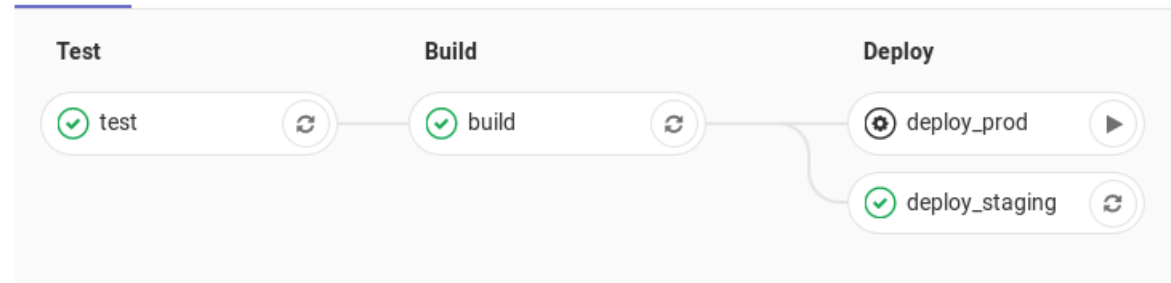
L'ajout de *when: manual* convertit le job en un job manuel et expose un bouton Play dans l'UI

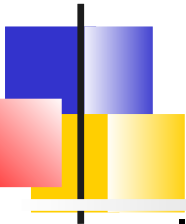
## Use busybox

🕒 4 jobs from [master](#) in 5 minutes 25 seconds (queued for 1 minute 45 seconds)

🔑 [ec75f5bf](#) ... 📄

Pipeline Jobs 4





# Environnements dynamiques

---

Il est possible de déclarer des noms d'environnement à partir de variables : **environnements dynamiques**

Les paramètres *name* et *url* peuvent alors utiliser :

- Les variables d'environnement prédéfinies
- Les variables de projets ou de groupes
- Les variables de *.gitlab-ci.yml*

Ils ne peuvent pas utiliser :

- Les variables définies dans *script*
- Du côté du runner

=> Il est possible de créer un environnement/déploiement pour chaque issue ou MR : Les Review Apps



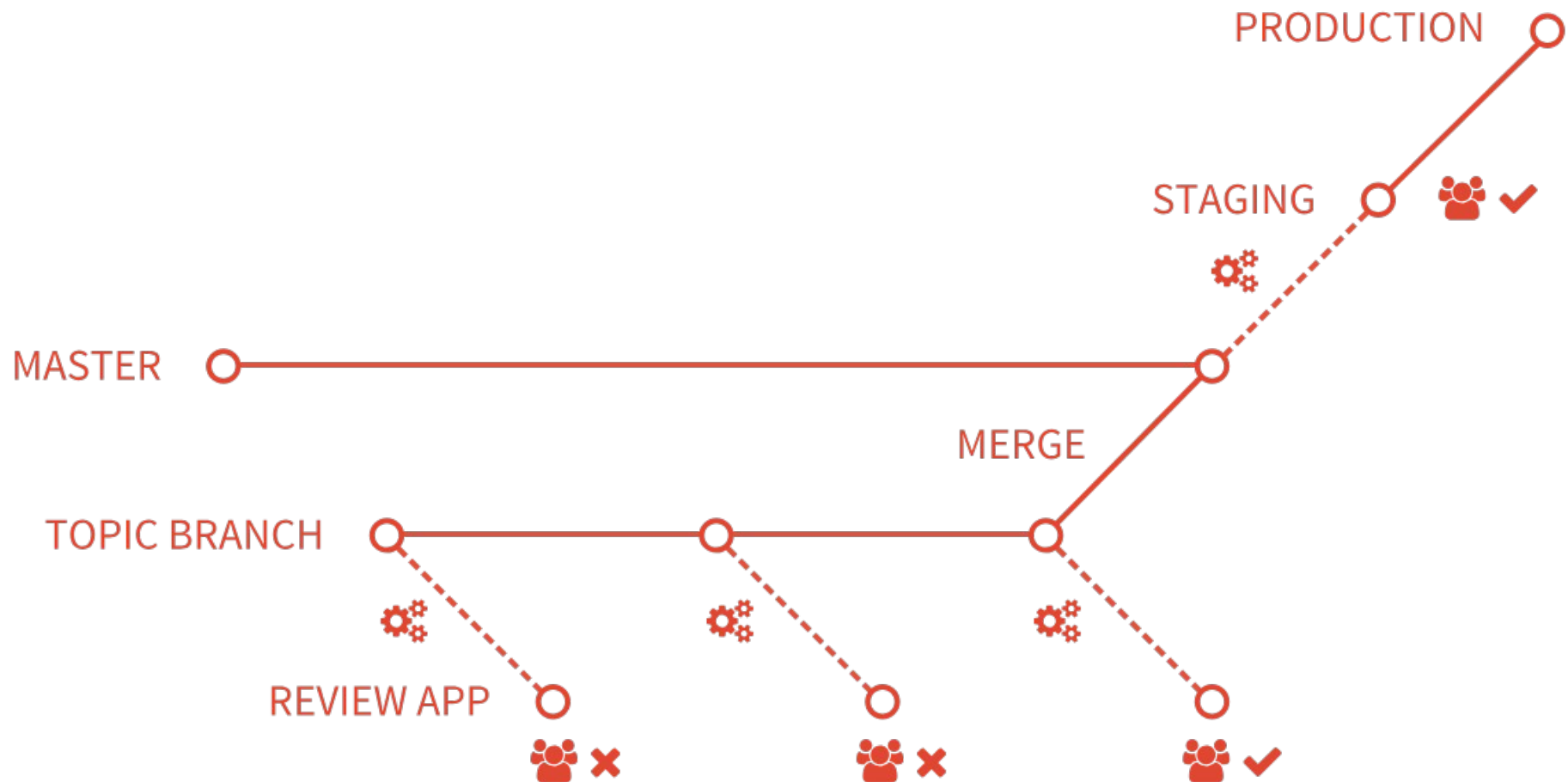
# Example

---

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



# Review App dans le workflow





# Exemple complet

---

```
stages:
  ..
  - deploy
  ...

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

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

deploy_prod:
  stage: deploy
  script: echo "Deploy to production server"
  environment:
    name: production
    url: https://example.com
  when: manual
  only:
    - master
```



# Arrêter un environnement

---

Arrêter un environnement consiste à appeler l'action ***on\_stop*** si elle est définie.

- Cela peut se faire par l'UI ou automatiquement dans la pipeline.
- Lors du workflow « Review App », l'action *on\_stop* est automatiquement appelée à la suppression de la branche de feature.





# Example

---

```
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
  only:
    - branches
  except:
    - master

stop_review:
  stage: deploy
  variables:
    GIT_STRATEGY: none
  script:
    - echo "Remove review app"
  when: manual
  environment:
    name: review/$CI_COMMIT_REF_NAME
    action: stop
```



# Syntaxe gitlab-ci.yml

---

Basiques *.gitlab-ci.yml*

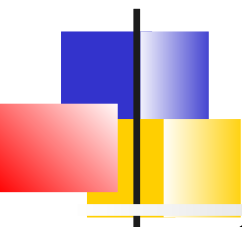
Principales Directives

Réutilisation

Intégration docker

Environnements et déploiements

**Package et Releasing**



# Gitlab

Gitlab offre plusieurs supports pour stocker et partager les artefacts construits :

- **GitLab Package Registry** est un registre privé ou public supportant les gestionnaires de packages courants : *Composer, Conan, Generic, Maven, npm, NuGet, PyPI, RubyGems*
- **GitLab Container Registry** est un registre privé pour les images Docker
- **GitLab Terraform Module Registry** supporte les modules Terraform

D'autre part, *Dependency Proxy* est un proxy local utilisé pour les images et paquets fréquemment utilisés.

Bien sûr, il est possible d'intégrer des solutions tierces (Nexus, Artifactory ...)



# *Gitlab Package Registry*

---

***GitLab Package Registry*** est  
disponible pour chaque projet.

Chaque projet a son propre registre de  
packages

Pour y publier, il faut utiliser les bonnes  
Urls d'accès et les bonnes permissions  
en utilisant un jeton.



# Création de jeton

---

Pour pouvoir utiliser le registre dans une pipeline, il faut créer un jeton nommé Job-Token qui sera exposé comme variable avec `${CI_JOB_TOKEN}`



# Exemple Maven

---

## settings.xml

```
<server>
  <id>gitlab-maven</id>
  <configuration>
    <httpHeaders>
      <property>
        <name>Job-Token</name>
        <value>${env.CI_JOB_TOKEN}</value>
      </property>
    </httpHeaders>
  </configuration>
</server>
```

## pom.xml

```
<distributionManagement>
  <repository>
    <id>gitlab-maven</id>
    <url>https://gitlab.example.com/api/v4/projects/<project_id>/packages/maven</url>
  </repository>
  <snapshotRepository>
    <id>gitlab-maven</id>
    <url>https://gitlab.example.com/api/v4/projects/<project_id>/packages/maven</url>
  </snapshotRepository>
</distributionManagement>
```



# *GitLab Container Registry*

---

Avec **GitLab Container Registry**, chaque projet peut avoir son propre espace pour stocker les images Docker.

La fonctionnalité doit être activée par l'administrateur.  
Elle n'est typiquement accessible qu'en **https**

Les images doivent suivre une convention de nommage :

**<registry URL>/<namespace>/<project>/<image>**

Des permissions fines peuvent être associés au registre



# Release

---

Dans GitLab, une **Release** permet de créer un instantané du projet incluant les packages et les notes de version.

- La release peut être créée sur n'importe quelle branche.
- La création d'une Release crée un tag. Si le tag est supprimé, la release également.





# Contenu et création d'une release

---

Une release peut contenir :

- Un instantané du code source
- Des packages créés à partir des artefacts des jobs
- Des méta-données de version
- Des releases notes



# Création : Edition

---

Une *release* peut être créée

- Via un Job CI/CD
- Manuellement via l'UI Releases page
- Via l'API

Après avoir créé une release, on peut :

- Ajouter des release notes
- Ajouter un message pour le tag associé
- Associer des milestones avec
- Joindre des ressources (packages ou autres)



# Création de release via un job CI/CD

---

Dans `.gitlab-ci.yml`, on crée des release en utilisant le mot-clé ***release***

3 méthodes typiques :

- Créer une release lorsqu'un tag est créé
- Créer une release quand un commit est fusionné dans la branche par défaut.
- Créer les méta-données de release dans un script personnalisé.



# Exemple

## *Création lors fusion dans la branche par défaut*

---

```
release_job:
  stage: release
  image: registry.gitlab.com/gitlab-org/release-cli:latest
  rules:
    - if: $CI_COMMIT_TAG
      when: never          # Ne pas exécuter si création manuelle de tag
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH # Branche par défaut
  script:
    - echo "running release_job for $TAG"
  release:
    tag_name: 'v0.$CI_PIPELINE_IID'          # Version incrémentée par pipeline
    description: 'v0.$CI_PIPELINE_IID'
    ref: '$CI_COMMIT_SHA'                    # tag créé à partir du SHA1.
```



# Phases des pipelines

---

Construction et tests développeur  
Analyses statiques  
**Dépôts d'artefacts**  
Gestion de l'infrastructure  
AutoDevOps