





# GitLab CI/CD

---

David THIBAU – 2019

david.thibau@gmail.com



# Agenda

---

## Rappels Git

- Principes de base et principales commandes
- Les branches locales et distantes

## Gitlab

- Projets et Membres
- GitlabFlow
- Dépôts et branches
- Merge Request
- Gestion des issues

## Gitlab CI/CD

- Jobs et Runners
- Pipelines
- Directives disponibles
- Intégration docker
- Environnements et déploiements
- AutoDevOps



# Rappels Git

---

## **Concepts de base et principales commandes**

Branches locales et distantes



# SCM

---

Un **SCM** (*Source Control Management*) est un système qui enregistre les changements faits sur un fichier ou une structure de fichiers afin de pouvoir revenir à une version antérieure

Le système permet :

- De restaurer des fichiers
- Restaurer l'ensemble d'un projet
- Visualiser tous les changements effectués et leurs auteurs



# Types de fichiers

---

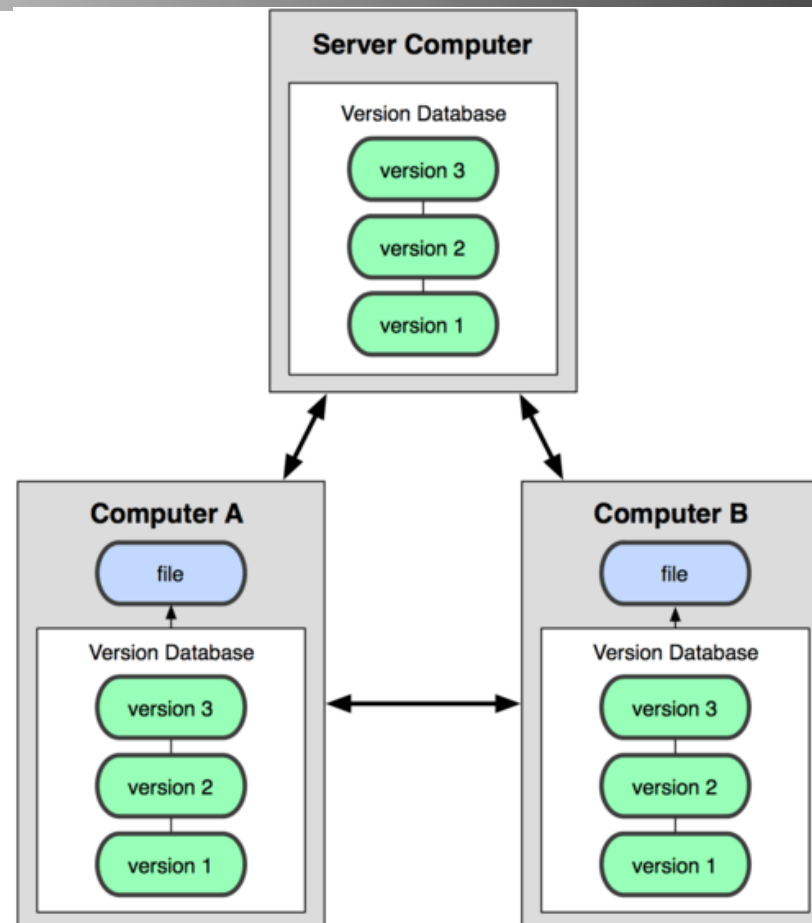
La plupart du temps les SCMs sont utilisés pour les fichiers **sources** des développeurs bien qu'ils soient capable de traiter **tout type** de fichiers

- Par exemple, un web designer peut vouloir garder toutes les versions d'une image ou d'une maquette de page

Cependant, les SCMs sont associés à des outils de comparaison de version (diff, patch).

Ces outils fonctionnent correctement avec les formats textes

# SCM distribué





# SCM distribué

---

Git est un SCMs **distribués**.

Chaque participant au projet détient l'intégralité du dépôt ou référentiel

- La plupart des opérations sont locales et donc rapides
- En cas de défaillance, il est facile de recréer le référentiel à partir d'une réplique
- Le fait de disposer de plusieurs référentiels distants permet de mettre en place différents workflows de collaboration





# Stockage des sources

---

*Git* adopte une approche radicalement différente pour le stockage des données par rapport aux systèmes traditionnels comme Subversion

Au lieu de stocker les fichiers initiaux et les changements entre révisions, *Git* stocke des **instantanés complets**

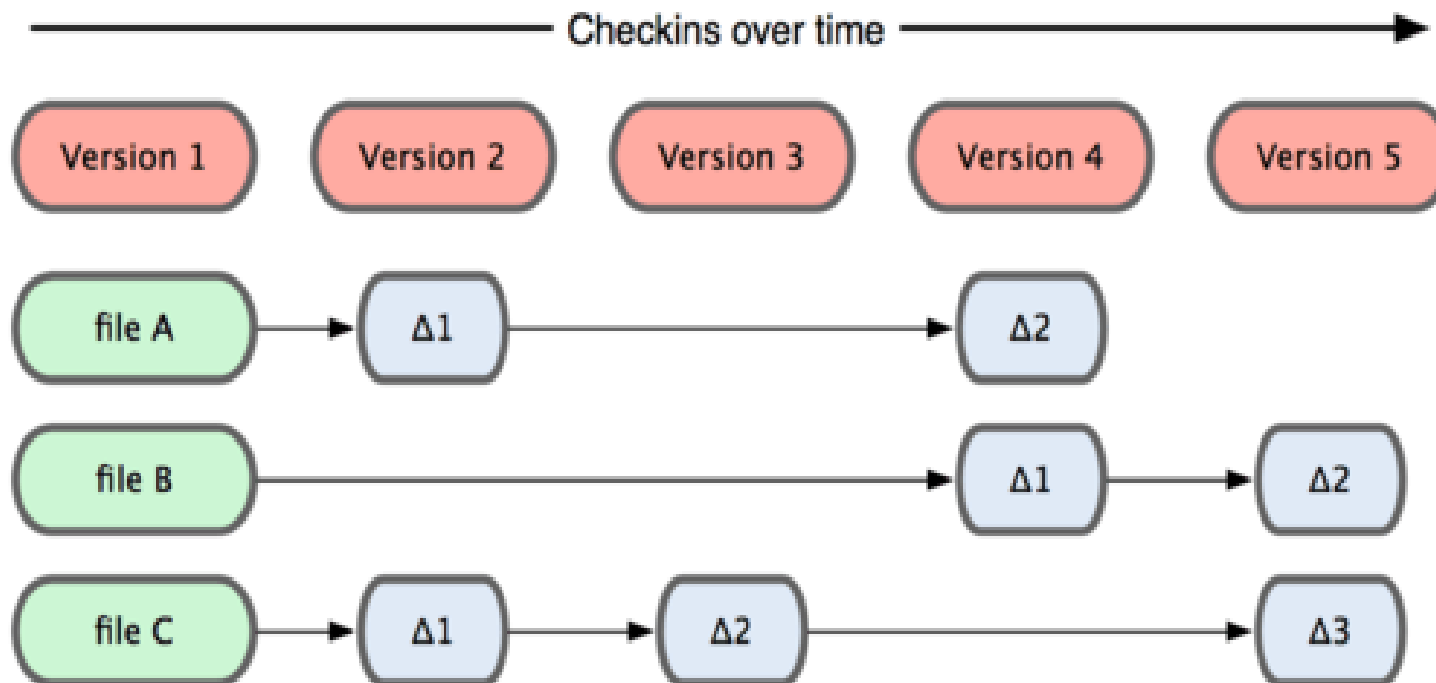
- A chaque commit, *Git* prend un instantané de l'état des fichiers et le stocke dans sa base.
- Pour être efficace, si un fichier est inchangé, son contenu n'est pas stocké une nouvelle fois mais plutôt une référence au contenu précédent

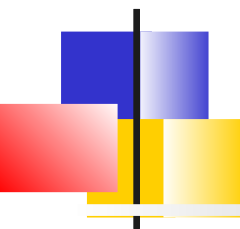
=> *Cette approche fait que Git se comporte plutôt comme un mini système de fichiers proposant des outils très efficaces*



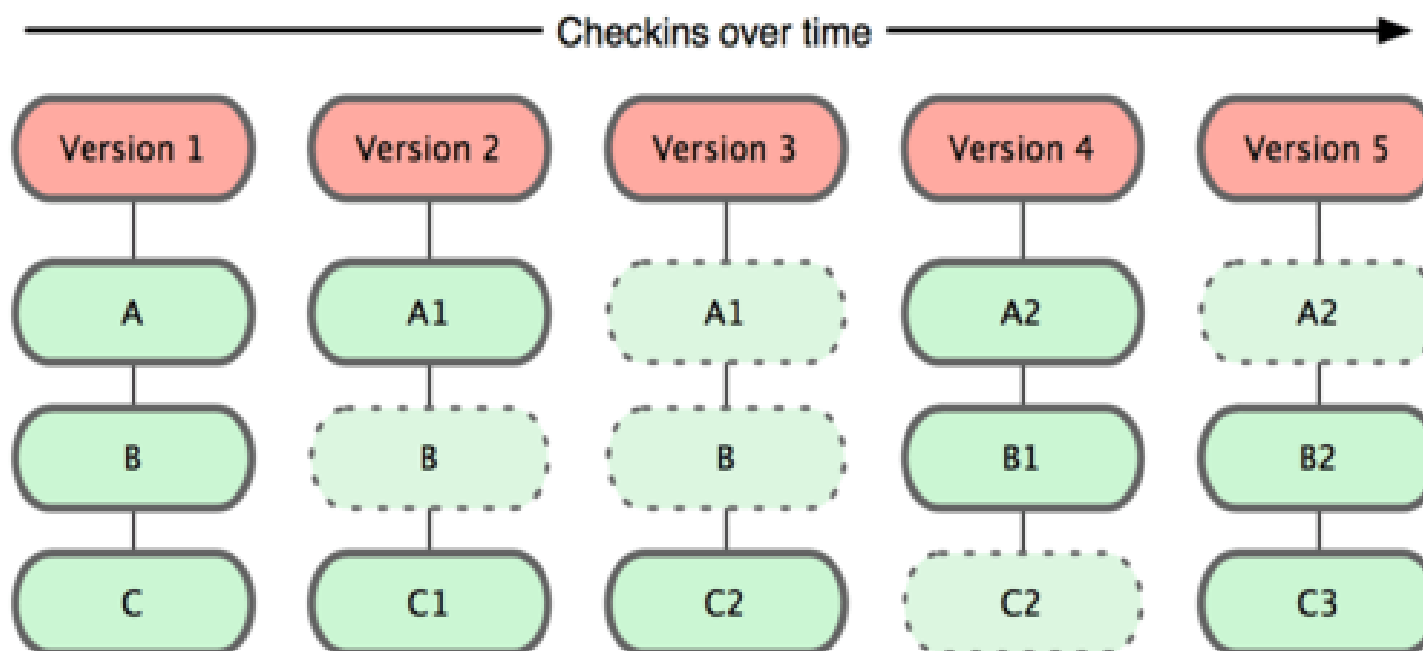
# Approche standard (CVS, SVN, ...)

---





# Approche Git





# Intégrité

Toutes les données du référentiel *Git* sont associées à un **checksum** avant qu'elles soient stockées. Le check-sum constitue l'identifiant de la donnée Git.

- Le *checksum* est un *hash* SHA-1 constitué de 40 caractères hexadécimaux fonction du contenu d'un fichier ou d'un répertoire.

Exemple :

*24b9da6552252987aa493b52f8696cd6d3b00373*

Les fichiers sont donc stockés dans le référentiel *Git* non pas par leur noms mais par leur clés de hachage

- Il est ainsi impossible de changer le contenu d'un fichier sans que *Git* s'en aperçoive



# Seulement des ajouts

---

La plupart des opérations dans Git consistent à ajouter des informations dans la base de données

- Ainsi, il est très difficile de faire des actions irréversibles

Comme avec tout SCM, il est possible de perdre des modifications si celles-ci n'ont pas été committées.

La synchronisation vers un référentiel distant fait office de sauvegarde



# État des fichiers

---

Les fichiers sources gérés par Git peuvent avoir 3 états :

- **Committed** : Les données sont stockées dans la base de données locale
- **Modified** : Le fichier a été changé mais pas encore committé dans la base
- **Staged** : Le fichier modifié a été marqué comme faisant partie du prochain commit

Les fichiers du projets que l'on désire pas suivre avec git sont listés dans des fichiers **.gitignore**



# Sections d'un projet

---

Ces 3 statuts fait qu'un projet Git est décomposé en 3 sections :

- Le **répertoire Git (.git/)** est l'endroit où Git stocke les métadonnées et les objets de sa base de données. Il contient l'intégralité des informations
- Le **répertoire de travail** est un « checkout » d'une version du projet. Les fichiers sont extraits de la base de données compressée et peuvent ensuite être modifiés
- La **zone de staging** est un simple fichier (quelquefois nommé **index**) qui stocke les informations sur ce qu'il faut inclure dans le prochain commit.



# Workflow standard

---

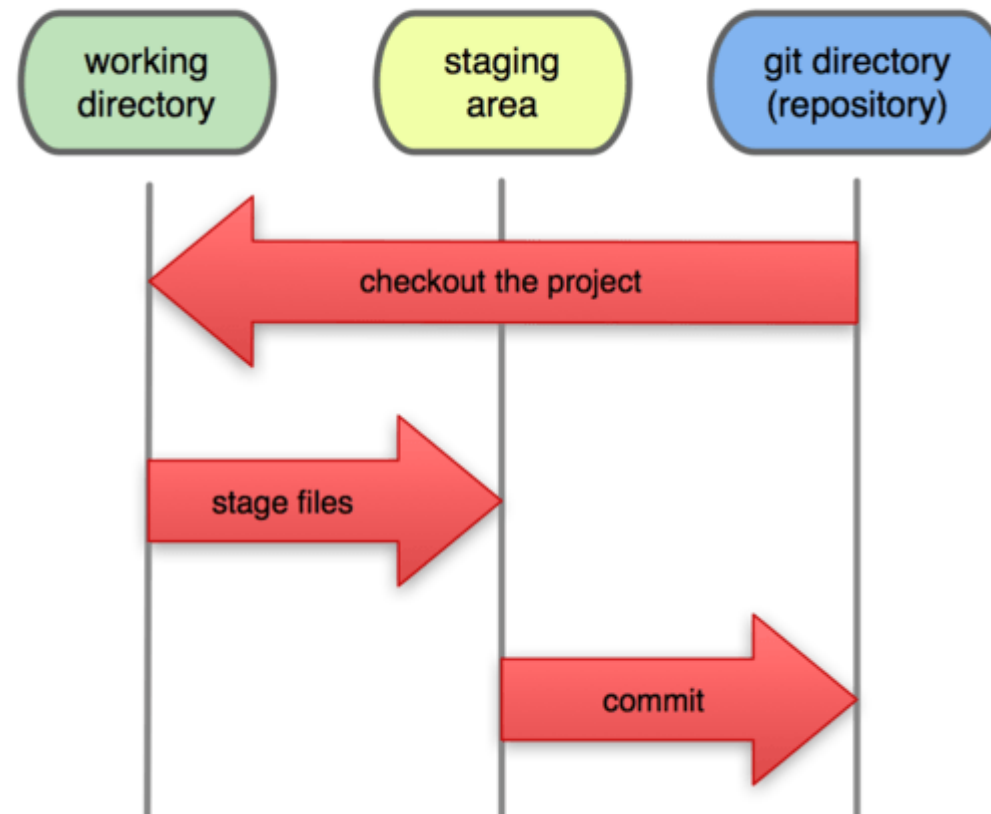
Le workflow standard de Git est :

1. Les fichiers du répertoire de travail sont **modifiés**
2. Ils sont ensuite placées dans la zone de **staging**.
3. Au **commit**, les fichiers de la zone de staging sont stockés dans le répertoire Git



# Sections d'un projet Git

## Local Operations





# Principales opérations

---

Configuration client : `git config`

Création de dépôt : `git init`, `git clone`

Statut du projet : `git status`, `git branch`, `git remote`

Enregistrement : `git add`, `git mv`, `git rm`, `git commit`

Consulation : `git diff`, `git log`

Synchronisation avec référentiel : `git push`, `git pull`, `git fetch`

Basculement du workspace : `git checkout`



# Rappels Git

---

Concepts de base et principales  
commandes

**Branches locales et distantes**



# Les branches

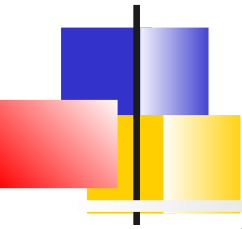
---

Les branchement signifie que le code diverge de la ligne principale de développement et que les deux branches évoluent indépendamment

Les branches Git sont très légères et les opérations de création et de basculement instantanées

=> Git encourage donc des workflows avec des branchements et des fusions de branches nombreuses.

=> En général, on crée une branche pour commencer un travail, quand le travail est terminé, on l'intègre dans la branche d'où l'on vient



# Branches locales/distantes

---

On distingue :

- les branches **locales** qui ne sont vues que par un développeur et qui lui facilitent son travail de tous les jours
- Les branches **distantes** qui sont des branches partagées par toute l'équipe ou par une partie de l'équipe

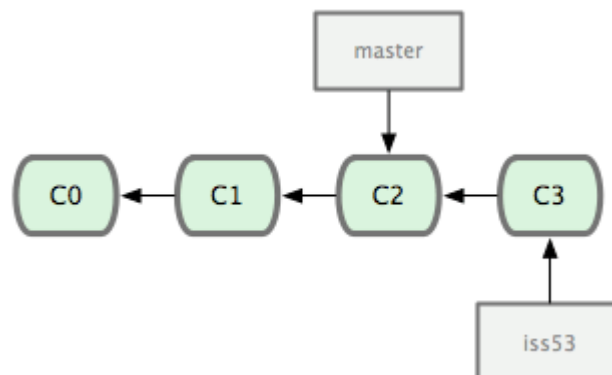
# Création branche locale

La création d'une branche le basculement du workspace se fait comme suite

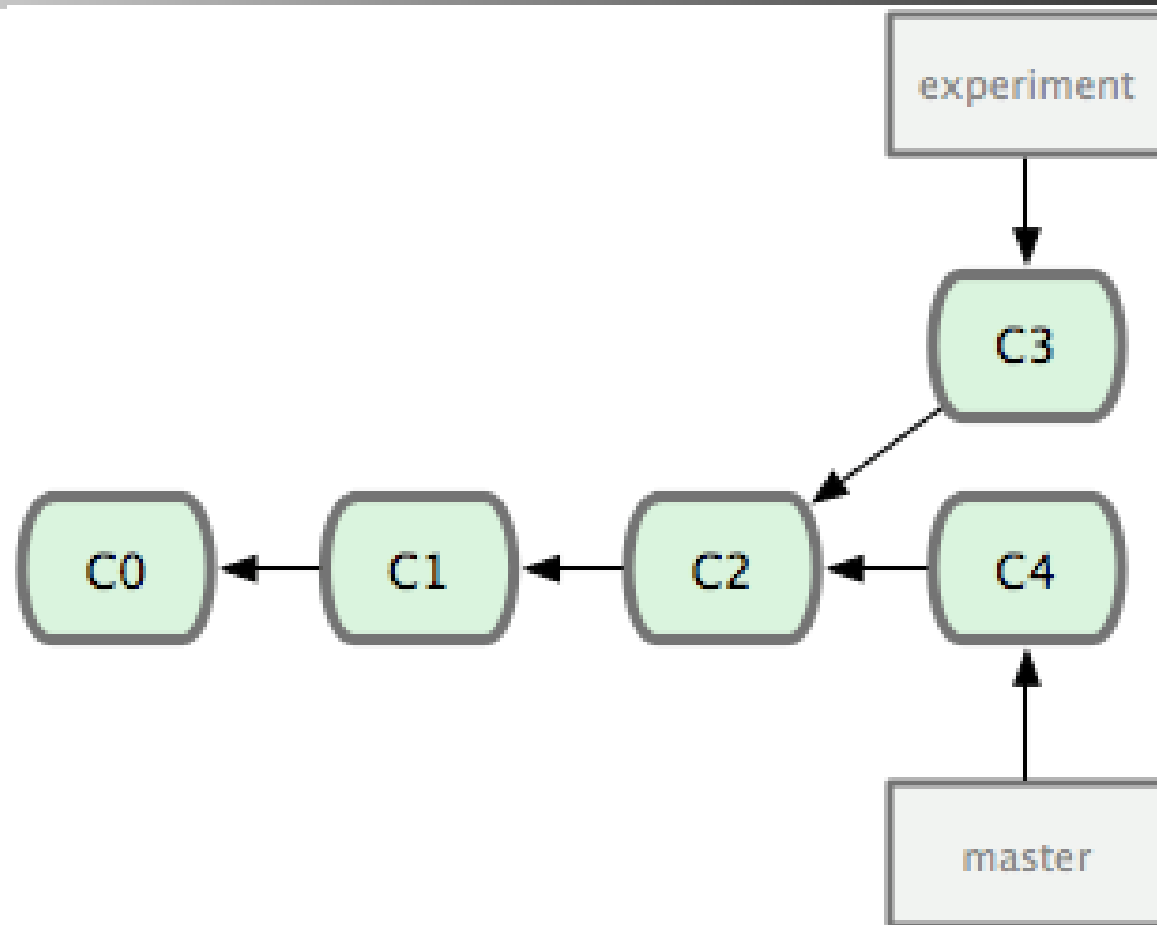
```
$ git checkout -b iss53
```

Switched to a new branch 'iss53'

Après quelques commits :

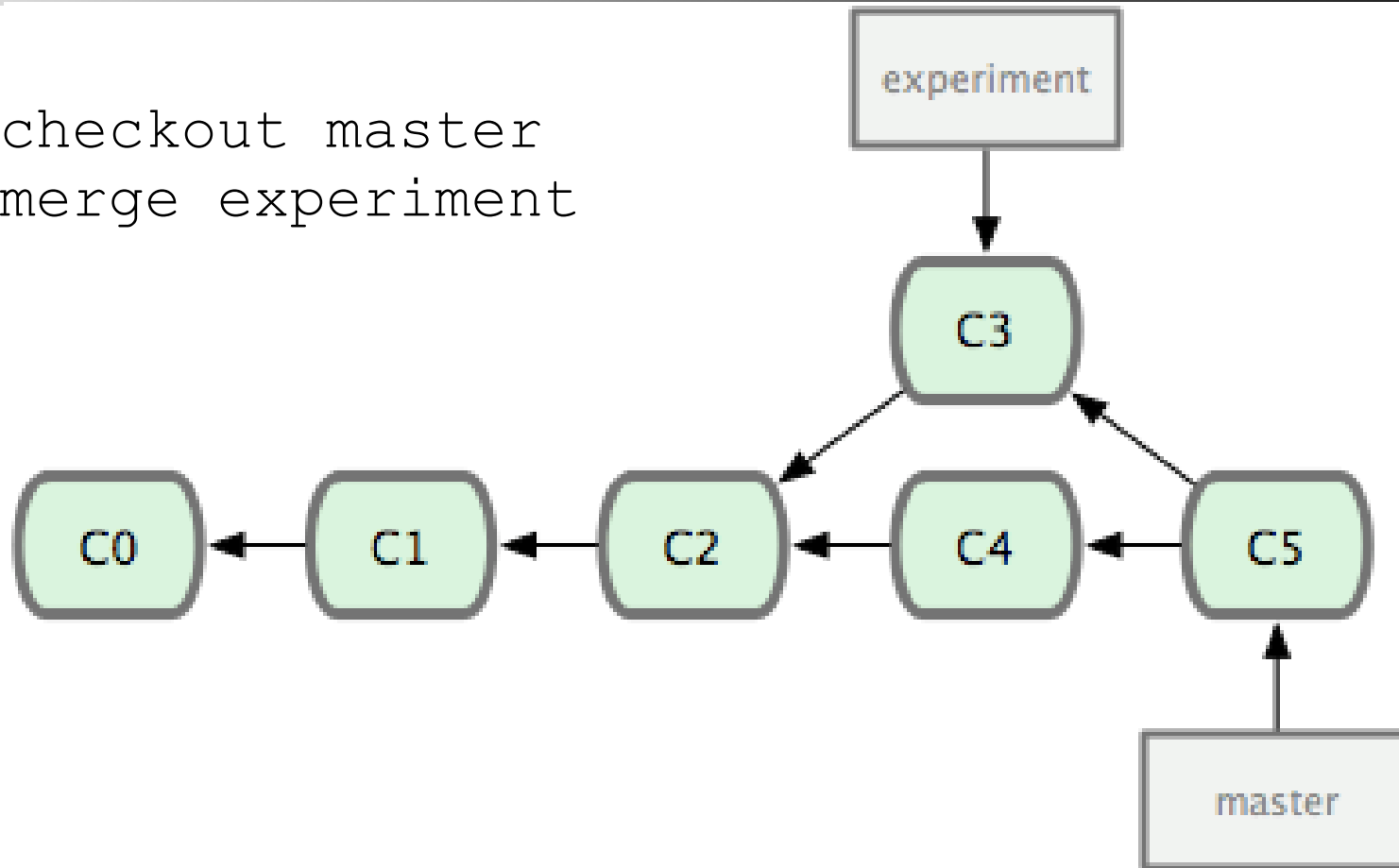


# 2 branches divergentes



# Résultat d'un merge

```
git checkout master  
git merge experiment
```







# Merge et conflit

---

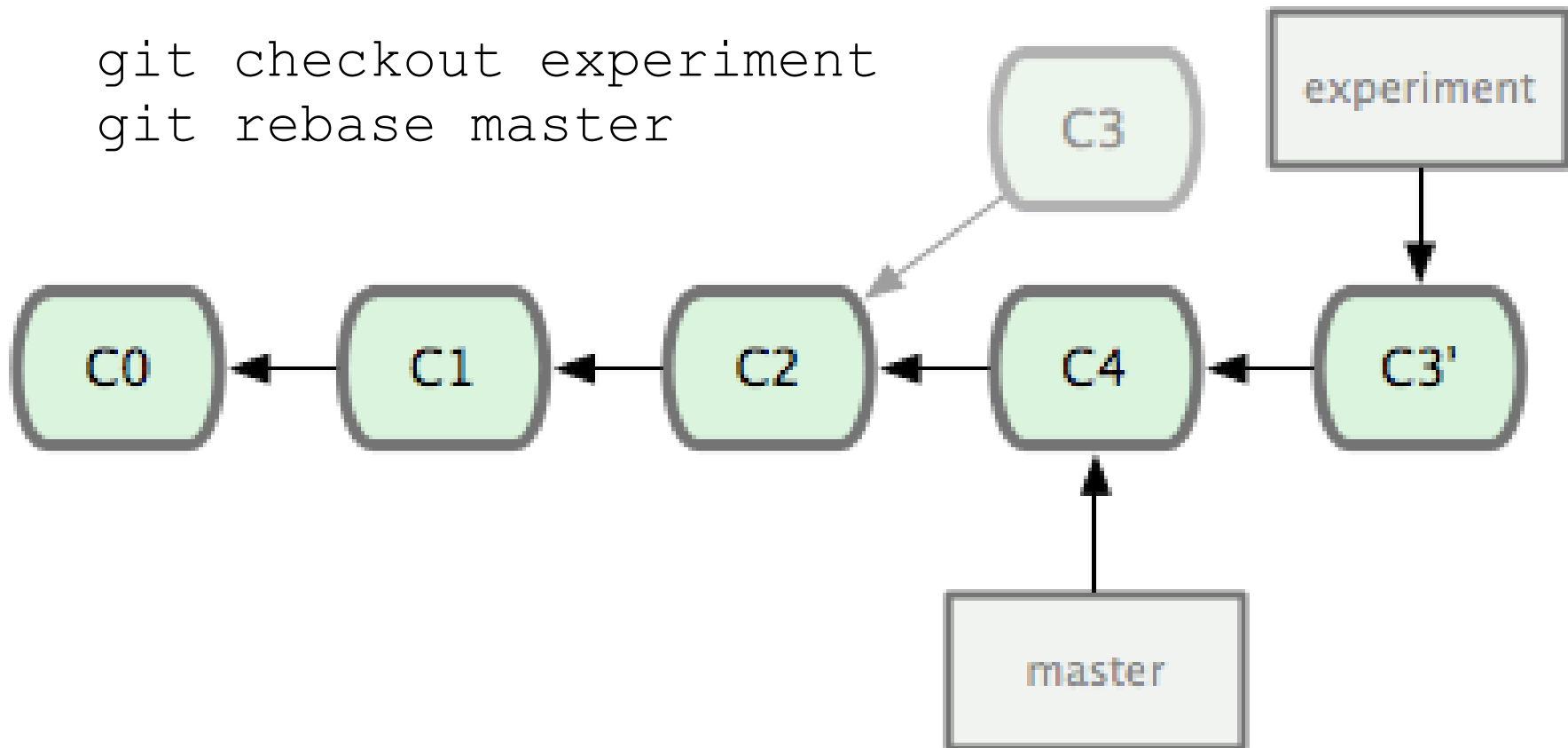
Si des conflits apparaissent lors de la fusion, l'opération s'interrompt

Il faut alors :

- Résoudre chaque conflit et l'indiquer à *git* avec  
`git add`
- Quand tous les conflits sont réglés  
`git commit`

# Résultat d'un rebase

```
git checkout experiment  
git rebase master
```





# Rebasing et conflit

---

Si un conflit apparaît lors de l'application d'un patch particulier, l'opération de rebasing s'interrompt

Il faut alors soit :

- Résoudre le conflit et continuer l'opération de rebasing  
`git add` après la résolution du conflit  
`git rebase --continue` pour continuer le rebasing
- Ignorer l'application de ce patch  
`git rebase --skip`
- Arrêter l'opération de rebasing  
`git rebase --abort`



# Branches distantes

---

Les **branches distantes** sont des références à l'état des branches sur un référentiel distant.

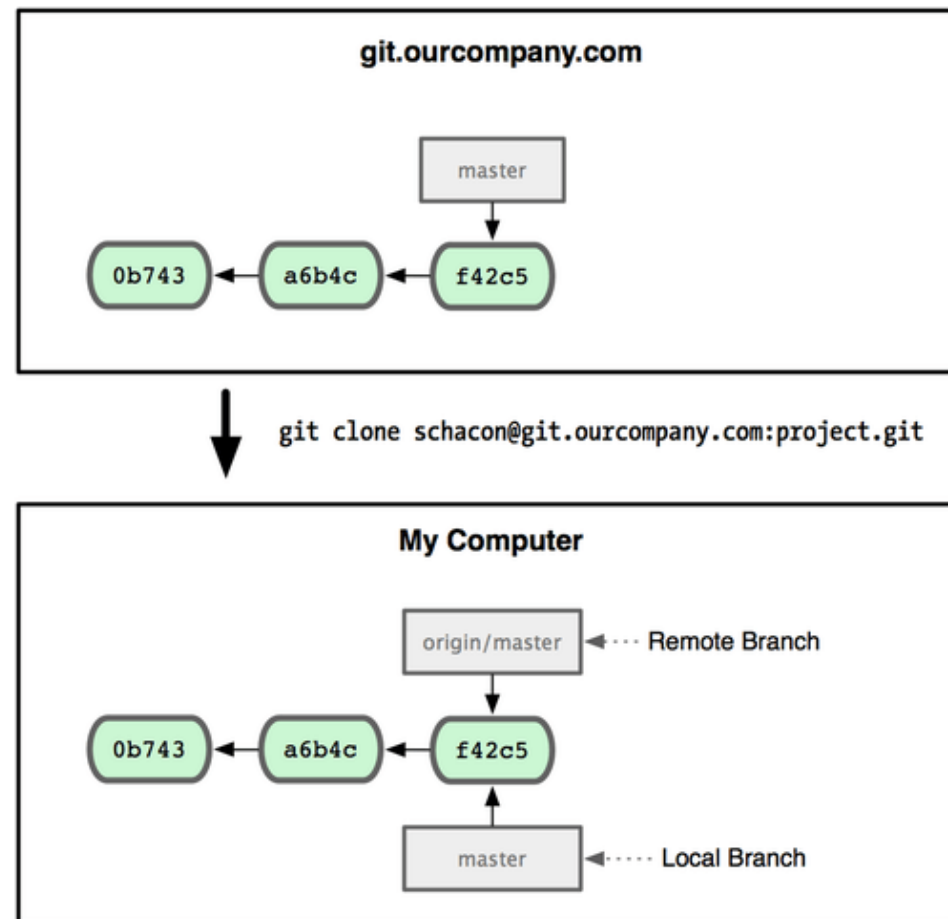
Ce sont des branches locales que l'on ne peut pas modifier.

La référence est mise à jour dès lors qu'il y a une communication réseau

- Les branches distantes sont donc comme des signets qui rappellent l'état de la branche, la dernière fois que l'on s'est connecté au référentiel distant

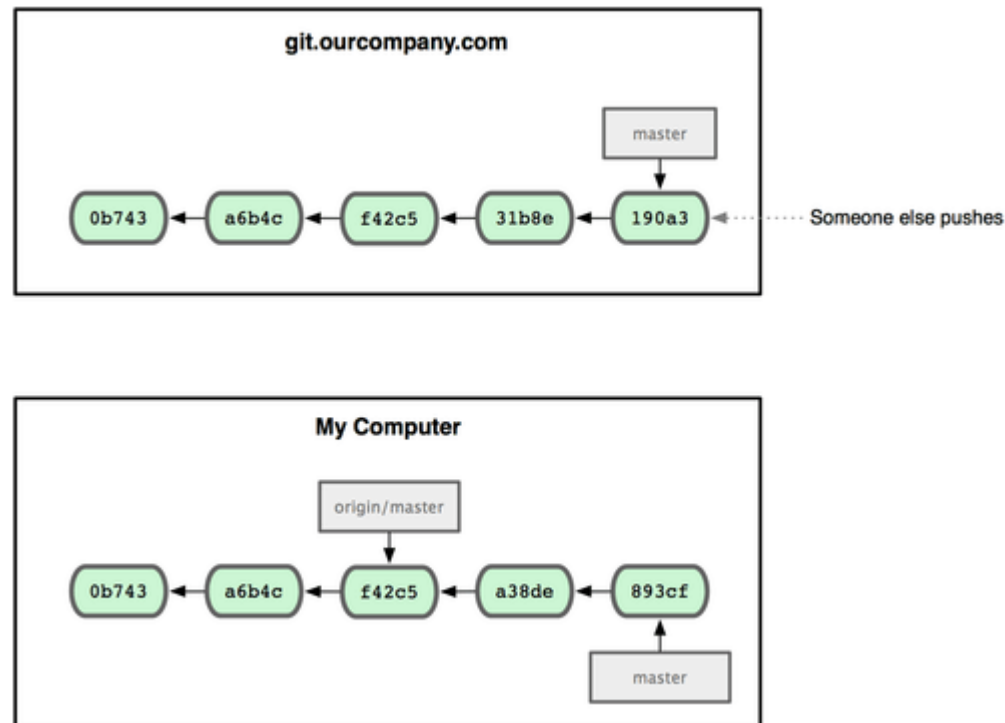
Elles sont référencées dans les commandes *Git* par **(remote)/(branch)**

# Exemple après clone



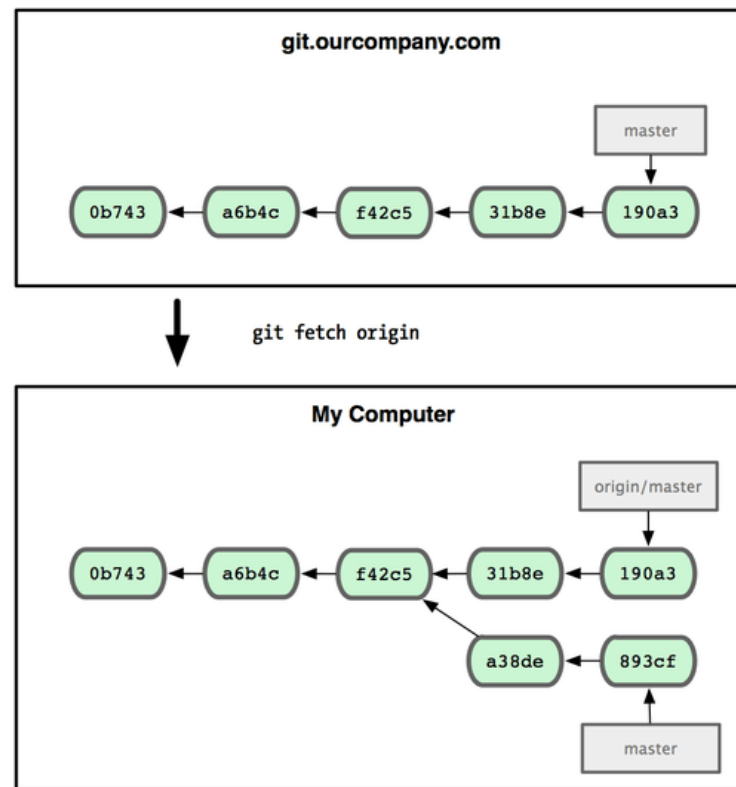
# Déplacement

Sans contact avec le serveur d'origine, le pointeur *origin/master* ne se déplace pas

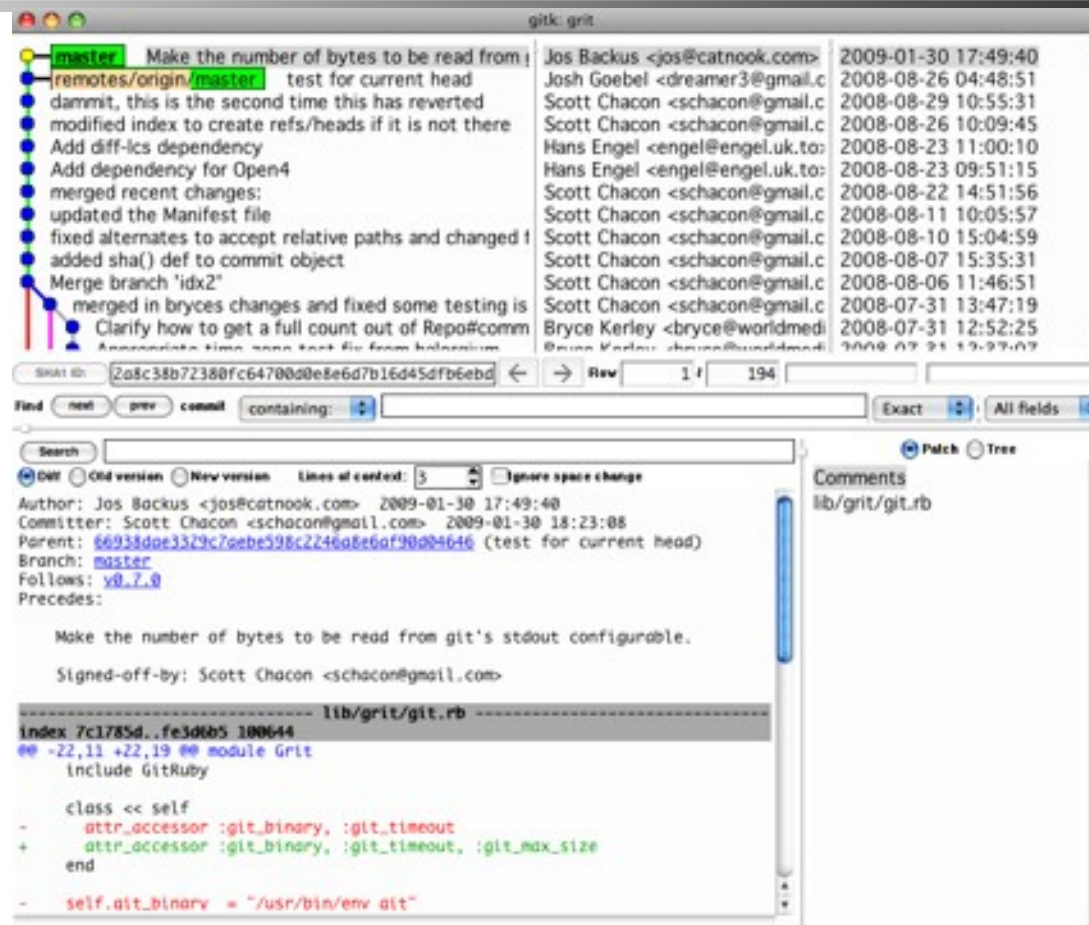


# Synchronisation

Pour synchroniser une branche distante, on exécute la commande ***git fetch origin*** qui rapatrie les nouvelles données et met à jour la base de données locale en déplaçant le pointeur *origin/master* à sa nouvelle position



# Exemple *gitk*







# Syntaxe complète push

---

La syntaxe complète de la commande push est :

```
$ git push origin serverfix:serverfix
```

Ce qui veut dire

« Recopier ma branche locale nommée **serverfix** dans la branche distante nommée **serverfix** »

Si l'on veut donner un autre nom à la branche distante, on peut utiliser :

```
$ git push origin serverfix:autrenom
```



# Branche de suivi

---

L'extraction d'une branche locale à partir d'une branche distante crée automatiquement une **branche de suivi**.

Les branches de suivi sont des branches locales qui sont en relation directe avec une branche distante

Dans une branche de suivi *git push*, et *git pull* sélectionne automatiquement le serveur impliqué

C'est le même mécanisme lorsque l'on clone un dépôt



# Branches de suivi

---

Il y a plusieurs façons de créer des branches de suivi :

L'option *--track* :

```
$ git checkout --track origin/serverfix
```

Si la branche n'existe pas localement et que son nom correspond exactement à une branche de suivi, on peut utiliser le raccourci :

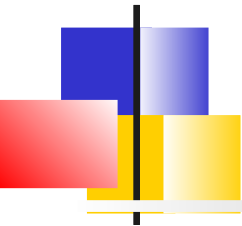
```
$ git checkout serverfix
```

Si l'on veut renommer la branche

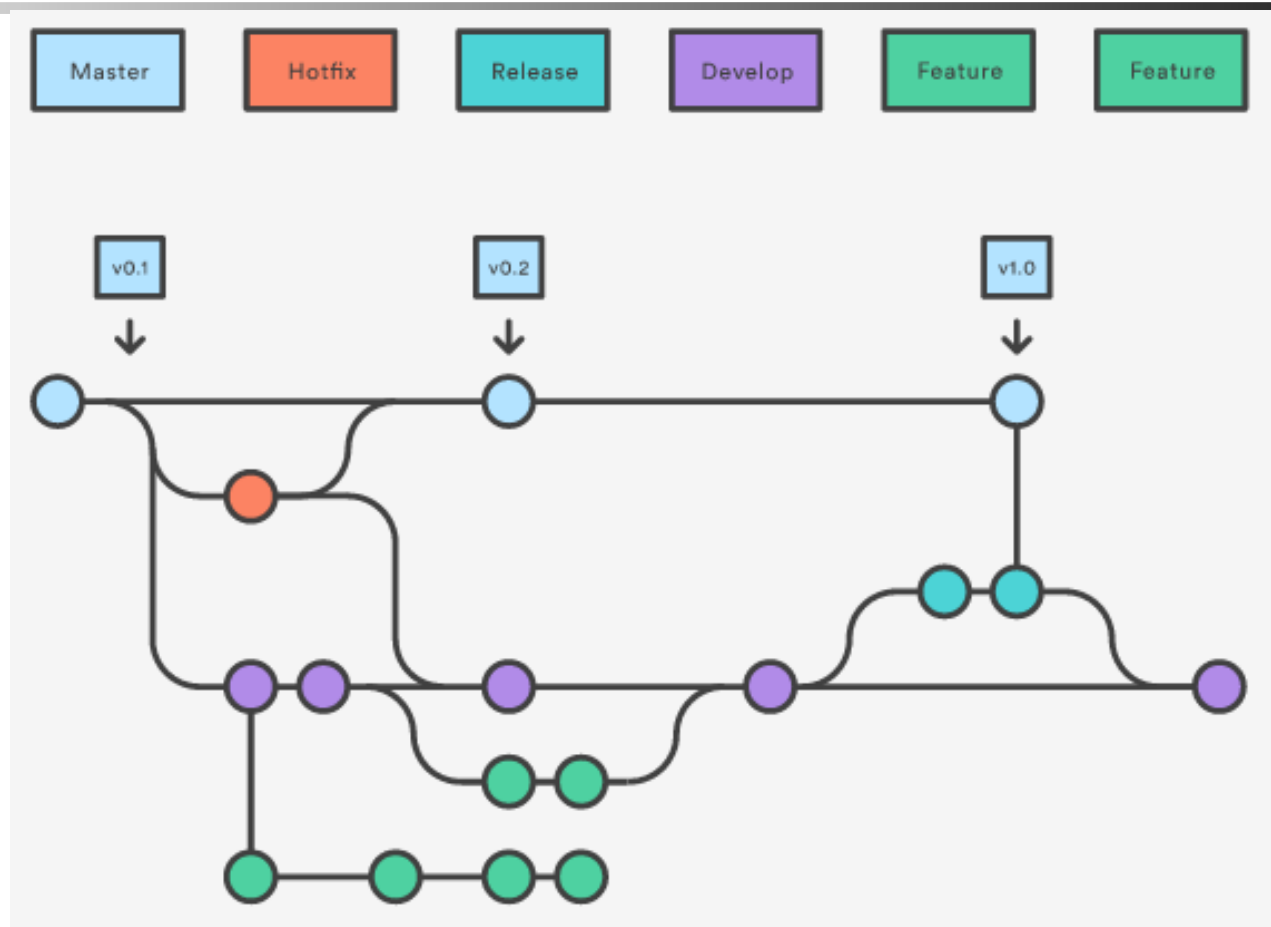
```
$ git checkout -b sf origin/serverfix
```

Enfin, si on veut utiliser une branche locale existante :

```
$ git branch --set-upstream-to origin/serverfix
```



# Gitflow





# Gitlab

---

Projets et membres

*GitlabFlow*

Dépôts et Branches

Merge Request

Gestion des issues



# Introduction GitLab

---

Gitlab est devenu un outil de gestion d'un cycle de vie **DevOps**

- Interface web gérant des référentiels Git et fournissant des fonctionnalités de collaboration, de suivi des problèmes et de pipeline CI / CD
- S'appuie sur les commandes de bases de Git
- S'intègre avec d'autres produits (annuaire LDAP, *Jira*, *Mattermost*, *Kubernetes*, *Slack* par exemple)
- Disponible sous une édition communautaire et entreprise



# Installations

---

Gitlab s'installe sous Linux. Différentes façons :

- **Omnibus Gitlab** : Packages pour différentes distributions de Linux
- **GitLab Helm chart** : Version Cloud, installation sous Kubernetes
- Images **Docker**
- A partir des **sources**

Également disponible en ligne : *gitlab.com*



# Community vs Entreprise

---

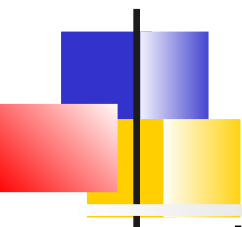
Le même cœur, l'entreprise edition ajoute du code propriétaire.

Il est possible d'utiliser Enterprise sans payer => Idem en fonctionnalités que l'édition communautaire

Les versions payantes apportent :

- Plus de fonctionnalités relatives aux Issues :
- Recherche de code avancé via Elastic Search, Revue de code visuel,
- Intégration LDAP, Kerberos, JIRA, Jenkins. Emailing
- Dépôts Maven, npm et docker
- Dépôts miroir distants
- PostgreSQL HA ...
- Support 24h/24





# Projets

---

Un projet *Gitlab* est associé à un dépôt 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.  
Tout utilisateur a la permission *Guest*
- **Interne** : Peut être cloné par tout utilisateur authentifié.  
Tout utilisateur a la permission *Guest*
- **Privé** : Ne peut être cloné et visible seulement par ses membres



# Fonctionnalités

---

Un projet apporte plusieurs fonctionnalités :

- **Suivi d'issues** : Collaboration sur le travail planifié, milestones,
- **Gestion de dépôts** : Organisation des branches, Merge request, accès au source, Web IDE
- **Pipelines de CI/CD**
- **Autres** : Wiki, Tableaux de bords, Gestion de release et environnements, dépôts Maven ou NPM, registres docker,

# Menus

**Projects** : Informations sur les commits, les branches, l'activité, les releases, tdb sur la productivité

**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

**Operations** : Gestion des environnements de déploiement

**Packages** : Accès au registre de conteneur

**Wiki** : Documentation annexe

**Snippets** : Bouts de code

**Settings** : Configuration projet, Visibilité, Merge Request, Membres, pipeline, intégration avec d'autres outils



# 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, Tags, Ajouts de clés SSH
- **Owner** : Créateur du projet, a le droit de le supprimer



# Groupes

---

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

- Pour ces groupes, il est possible de définir des membres  
=> Les membres d'un groupe ont alors accès à tous les projets du groupe

Les groupes peuvent être hiérarchiques

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

*Settings -> General -> Advanced -> Transfer project -> Select a new namespace*



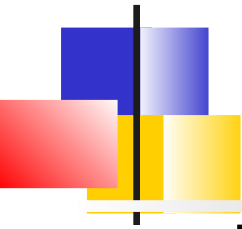
# *Share with group*

---

Si les membres du groupe  
« *Engineering* » doivent avoir accès à  
un autre projet appartenant déjà à un  
autre groupe, il faut utiliser la fonction  
« ***Share with group*** »

Sur l'autre projet :

*Settings → Members → Share with group*



# Configuration Utilisateur

---

Dans la partie *Settings* d'un utilisateur, en dehors des informations personnelles, on retrouve :

- La configuration des notifications par projet
- La gestion des clés SSH facilitant l'authentification
- La gestion des clés GPG permettant de signer des tags
- Les préférences (en particulier la langue)



# Mise en place clés ssh

---

La mise en place des clés *ssh* permet de pouvoir interagir avec Gitlab 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
```



# Gitlab

---

Projets et membres

**GitlabFlow**

Dépôts et Branches

Merge Request

Gestion des issues

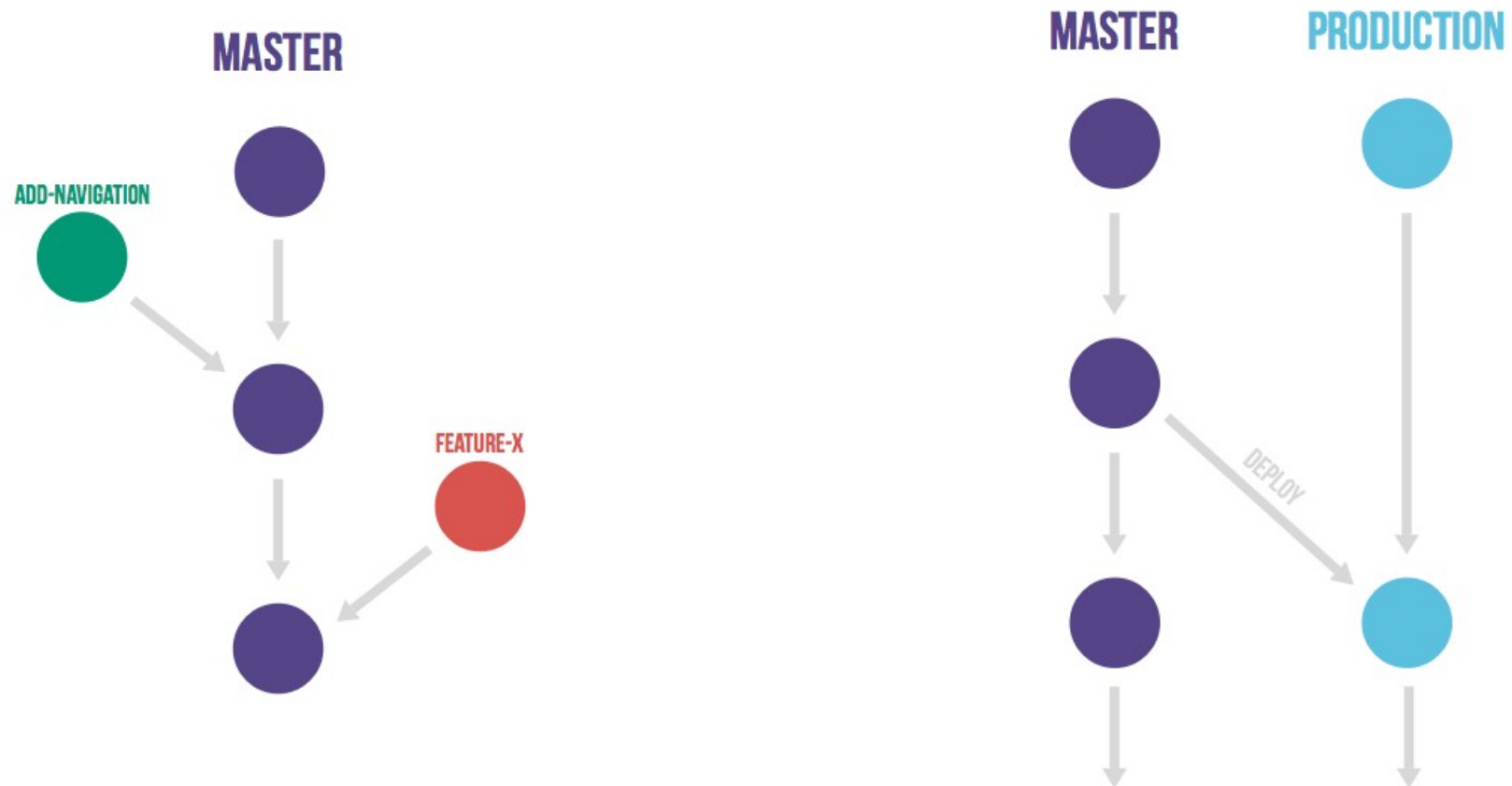


# Gitlab Flow

**Gitlab Flow** est une stratégie simplifiée d'utilisation des branches pour un développement piloté par les features ou le suivi d'issues

- 1) Les fix ou fonctionnalités sont développés dans une feature branch
- 2) Via un merge request, elles sont intégrées dans la branche master
- 3) Il est possible d'utiliser d'autres branches :
  - *production* : Chaque merge est taggée et correspond à une livraison dans l'environnement de production
  - *release* : Branche de release. Chaque merge est taggée et correspond à une distribution de release. Les Bug fixes sont repris de master via des cherry-picks dans les branches de release impactées

# Features, Master and Production





# Workflow typique

---

- 1) Les travaux sont effectués localement dans une branche
- 2) Ils sont ensuite poussés sur Gitlab
- 3) Un merge request est créé
- 4) *Gitlab* permet alors d'effectuer une revue de code ainsi que de collaborer sur les modifications en cours
- 5) Éventuellement, ces modifications peuvent être déployées sur une « Review Apps »
- 6) Des approbations peuvent être demandées aux *maintainers* avant le merge dans la branche master



# Déploiements

---

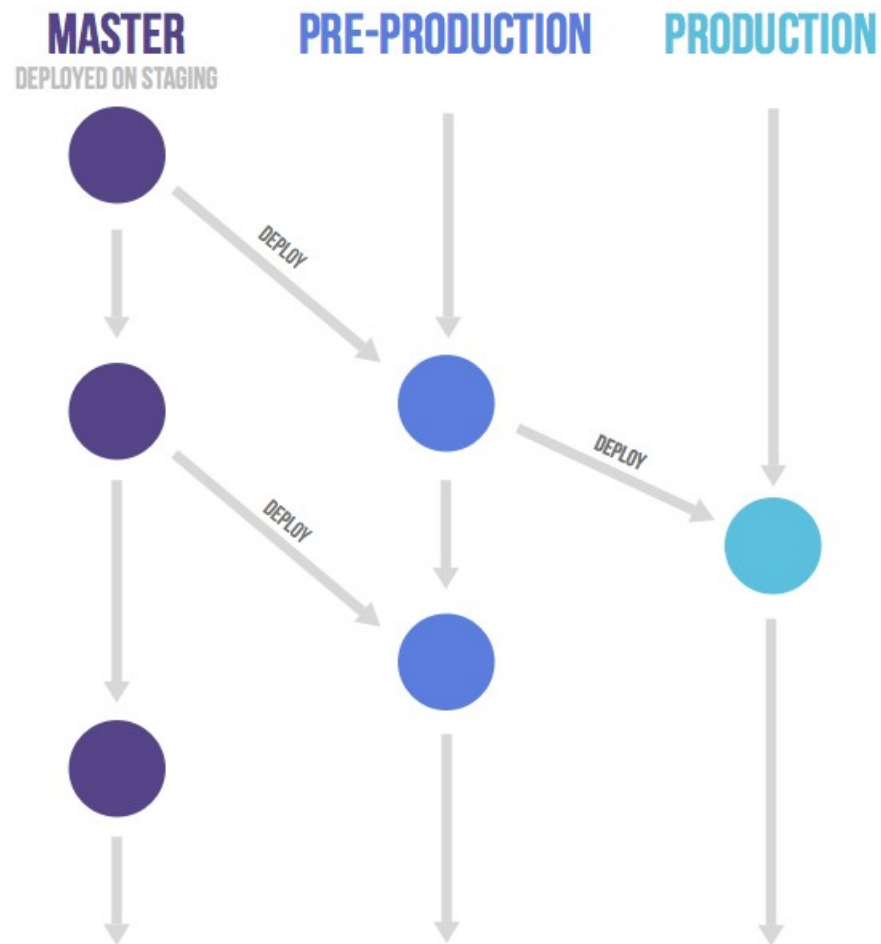
Les déploiements s'effectuent à partir de la branche *master*

Gitlab est capable de gérer plusieurs **environnements** de déploiement

- Pour chaque environnement, une branche est créé  
=> L'historique des déploiements sur un environnement particulier est aisément consultable



# Branches d'environnement





# Releases

---

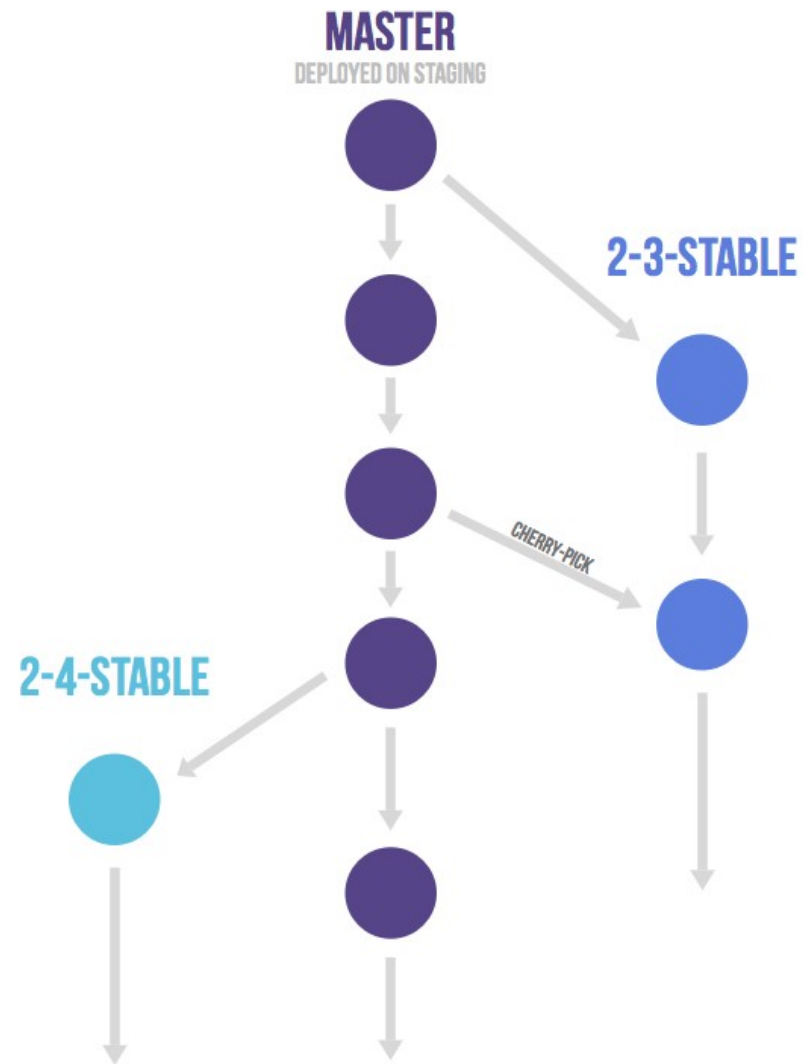
Pour la distribution de software, il est possible de mettre en place des **branches de release**

- Lors de la préparation d'une release, une branche stable est créé à partir du *master*
- Un tag est créé pour chaque version
- Les bugs critiques trouvés à posteriori sont appliqués via des Cherry-pick  
(Ne pas committer directement dans la branche stable)

Gitlab permet de visualiser les *Releases* d'un projet via l'UI et de fournir les artefacts construits via téléchargement



# Branches de releases





# Gitlab

---

Projets et membres  
GitlabFlow

**Dépôts et Branches**

Merge Request  
Gestion des issues



# Particularités *Gitlab*

On peut interagir avec les dépôts GitLab via **l'UI** ou en **ligne de commande**.

GitLab supporte des langages de **markup** pour les fichiers du dépôt. Utilisé principalement pour la documentation

Lorsqu'un fichier **README** ou index est présent, son contenu est immédiatement rendu (sans ouverture du fichier)

L'UI donne la possibilité de **télécharger** le code source et les archives générées par les pipelines

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

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

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



# Particularités du commit

---

- **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, ils seront affichés sur leur thread respectif.
- Il est possible via l'UI d'effectuer aisément un *cherry-pick* ou un *revert* d'un commit particulier
- Possibilité de signer les commits via GPG



# 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



# Branche par défaut

---

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

Peut-être changé *Settings* → *Repository*.

C'est dans la branche par défaut que sont fusionnées les modifications relatives à une issue lors d'un *merge request*.

La branche par défaut est également une branche protégée



# Création de branche

---

Plusieurs façons de **créer des branches** avec Gitlab :

- A partir d'une issue, la branche est donc documentée avec la collaboration sur l'issue
- A partir du tableau de bord projet, de la même façon la branche sera fusionnée dans la branche par défaut



# Branche protégée

---

## Un branche protégée

- Seul un membre avec au moins la permission *Maintainer* peut la créer
- Seul un *Maintainer* peut y faire des push
- Il empêche quiconque de forcer un push vers la branche
- Il empêche quiconque de supprimer la branche

On peut utiliser des *wildcards* pour protéger plusieurs branches en même temps. *Ex :*

*\*-stable, production/\**





# Permissions pour « push » et « merge »

---

Les permissions par défaut d'une branche protégée peuvent être surchargées avec les champs de configuration “**Allowed to push**” et “**Allowed to merge**”

Par exemple, on peut positionner

- “*Allowed to push*” à “*No one*”
- “*Allowed to merge*” à “*Developers + Maintainers*”

=> Tout le monde doit soumettre un merge request pour mettre à jour la branche protégée



# Gitlab

---

Projets et membres  
GitlabFlow  
Dépôts et Branches  
**Merge Request**  
Gestion des issues



# Introduction (1)

---

Le ***Merge Request*** est la base de la collaboration sur Gitlab

Un MR permet :

- Comparer les changements entre 2 branches
- Revoir et discuter des modifications de code
- Voir l'appli. en fonctionnement (*Review Apps*)
- Exécuter une pipeline
- Empêcher une fusion trop précoce avec le *WIP*
- Visualiser le processus de déploiement
- ...



# Introduction (2)

---

- Supprimer automatiquement la branche associée à la modification
- Assigner la MR à un responsable
- Affecter un milestone
- Utiliser des workflows de collaboration via les labels
- Faire un suivi du temps
- Résoudre les conflits de merge via l'UI
- Autoriser les fast-forward
- ...

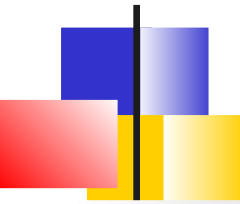



# Cycle de vie d'un MR

---

- 1) Au démarrage d'un nouveau travail, le développeur crée un *merge request*.  
Le travail n'est pas prêt à être fusionné mais la collaboration et la revue de code peuvent commencer dans une feature branch.  
Le *Merge Request* est préfixé par **WIP**
- 2) Lorsque la fonctionnalité est prête, le développeur assigne le *merge request* à un des membres du projet (un mainteneur en général)
- 3) Le mainteneur a le choix entre effectuer la fusion dans master, demander au développeur des améliorations, abandonner la MR
- 4) Lorsque la feature branch est fusionnée, elle est détruite .

# Vue projet



 GitLab Community Edition

Overview

Repository

Issues8,730



**Merge Requests472**

CI / CD

Wiki

Snippets


Settings

GitLab /  GitLab.org /  GitLab Community Edition

Merge Requests

Edit Merge RequestsNew merge request

Open472Merged11,188Closed1,969All13,629

 Search or filter results...

Last created

test MR

!13679 · opened 17 minutes ago by Mike Greiling

updated 14 minutes ago

Add docs for group issues page and group merge requests page

!13678 · opened 20 minutes ago by Victor Wu

updated 2 minutes ago

Docs update links guideline to inline links

!13677 · opened 36 minutes ago by Marcia Ramos 10.0 Documentation docs-update

updated 34 minutes ago

WIP: Clean up new dropdown styles 0 of 1 task completed

!13676 · opened 50 minutes ago by Winnie Hellmann 10.0 Deliverable UI polish frontend

updated 15 minutes ago

Greatly reduce test duration for git\_access\_spec

!13675 · opened 58 minutes ago by Robert Speicher 10.0 Edge backstage performance technical debt test

updated 20 minutes ago

Implement new system note icons 0 of 11 tasks completed

!13673 · opened about 3 hours ago by Bryce Johnson 10.0 Deliverable frontend

updated less than a minute ago

WIP: Prepare 9.5 RC6

!13672 · opened about 3 hours ago by Jose Ivan Vargas Lopez 9-5-stable Release

updated about an hour ago

Use Gitaly 0.33.0 0 of 11 tasks completed

!13671 · opened about 4 hours ago by Jacob Vosmaer (GitLab) 9.5 Gitaly Pick into Stable

updated about 4 hours ago

[WIP] Make the import take subgroups into account 0 of 9 tasks completed

!13670 · opened about 5 hours ago by Bob Van Landuyt 10.0 Platform

updated about 5 hours ago

70



# Commentaires et discussions

---

Des **commentaires** peuvent être associés aux différents objets de Gitlab dont les MR

Un commentaire peut être transformé en **discussion**. (via l'UI ou via un email)

Le statut de résolution d'une discussion peut affecter la merge request

- La discussion démarre avec un statut *unresolved*
- Elle s'applique en général à un *diff* d'un des commits du merge request



# Commentaires et discussions

---

Pour créer une discussion sur un commit

- Afficher les commits liés au MR
- Sur un commit, accéder à l'onglet *Changes* et laisser un commentaire
- La discussion apparaît dans l'onglet discussions du MR et peut être résolue via le bouton « *Resolve Discussion* »

Il est possible de

- voir toutes les discussions non résolues
- De déplacer les discussions non résolues vers une issue





# Conflits

---

Lorsqu'une MR a des conflits, il est possible de les résoudre via l'UI

*GitLab* résout les conflits en créant un commit de merge dans la branche source.

Le commit peut alors être testé avant d'affecter la branche cible.



# Squash

---

Lors d'un merge, il est possible de convertir tous les commits du merge en un seul et donc d'avoir un historique plus clair : ***squash***

Le message de commit est alors :

- Repris du premier message de commit multi-lignes
- Le titre du merge request si il n'y a pas de messages multi-lignes

Il peut être personnalisé au moment du merge

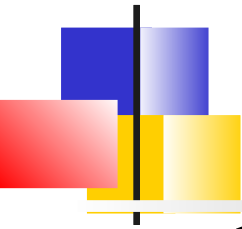


# Méthodes de merge

---

Les méthodes de merge ont une influence sur l'historique du projet :

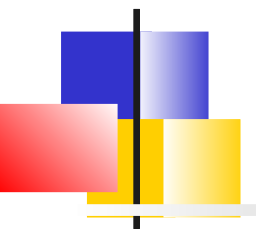
- **Merge commit (défaut)** : Chaque fusion crée un commit de merge
- **Merge commit avec historique semi-linéaire** :  
Chaque fusion crée un commit de merge mais la fusion n'est possible que si c'est une fast-forward  
Si un conflit arrive, l'utilisateur a la possibilité de rebaser
- **Fast-forward merge** : Pas de commit de merge, seules les fast-forward sont possibles  
Si un conflit arrive, l'utilisateur a la possibilité de rebaser



# Vérifications avant merge

---

- 2 vérifications effectuées avant un merge peuvent être configurées :
- Vérifier que la pipeline réussisse
  - Vérifier que les discussions sont closes



# Configuration Merge Request

## Merge requests

Collapse

Choose your merge method, options, checks, and set up a default merge request description template.

### Merge method

This will dictate the commit history when you merge a merge request

- ☒ Merge commit  
Every merge creates a merge commit
- ☐ Merge commit with semi-linear history  
Every merge creates a merge commit  
Fast-forward merges only  
When conflicts arise the user is given the option to rebase
- ☐ Fast-forward merge  
No merge commits are created  
Fast-forward merges only  
When conflicts arise the user is given the option to rebase

### Merge options

Additional merge request capabilities that influence how and when merges will be performed

- ☐ Automatically resolve merge request diff discussions when they become outdated
- ☒ Show link to create/view merge request when pushing from the command line

### Merge checks

These checks must pass before merge requests can be merged

- ☐ Pipelines must succeed  
Pipelines need to be configured to enable this feature. [?](#)
- ☐ All discussions must be resolved



# Approbateurs

---

Si l'installation Gitlab a été configuré il est possible de configurer la politique d'approbation d'une MR. Les approbateurs des *MRs* sont configurés au niveau du projet.

- => On peut alors indiquer des membres du projet (ou du groupe ou du groupe partagé) ainsi qu'un nombre



# Écran de configuration

## Merge request approvals

Set a number of approvals required, the approvers and other approval settings. [Learn more about approvals.](#)

### Add approvers

Members

No. approvals required

All members with Developer role or higher and code owners (if any)

0

Edit

Add approvers

- ☐ Require approval from code owners ?
- ☒ Can override approvers and approvals required per merge request ?
- ☒ Remove all approvals in a merge request when new commits are pushed to its source branch
- ☒ Prevent approval of merge requests by merge request author ?
- ☐ Prevent approval of merge requests by merge request committers ?
- ☐ Require user password to approve ?

Save changes



# Gitlab

---

Projets et membres  
GitlabFlow  
Dépôts et Branches  
Merge Request  
**Gestion des issues**





# Introduction

---

Les **Issues** permettent la collaboration avant et pendant leur implémentation

Elles peuvent être utilisées pour différents cas d'utilisation :

- Discuter de l'implémentation d'une nouvelle idée
- Suivi de tâches
- Backlog agile, Reporting de bug, Demande de support

Elles sont toujours associées à un projet.

Elles peuvent être visualisées par groupe de projets.



# Données associées à une *issue*

---

## Contenu :

- Titre
- Description et tâches
- Commentaires et activité

## Membres

- Auteur
- Responsables

## Etat

- Status (ouvert/fermé)
- Confidentialité
- Tâches (terminé ou en suspens)

## Planning et suivi

- Milestone
- Date de livraison
- Poids
- Suivi du temps
- Tags (Labels)
- Votes
- Reaction emoji
- Issues liées
- Epic (collection d'issues) affectée
- Identifiant et URL



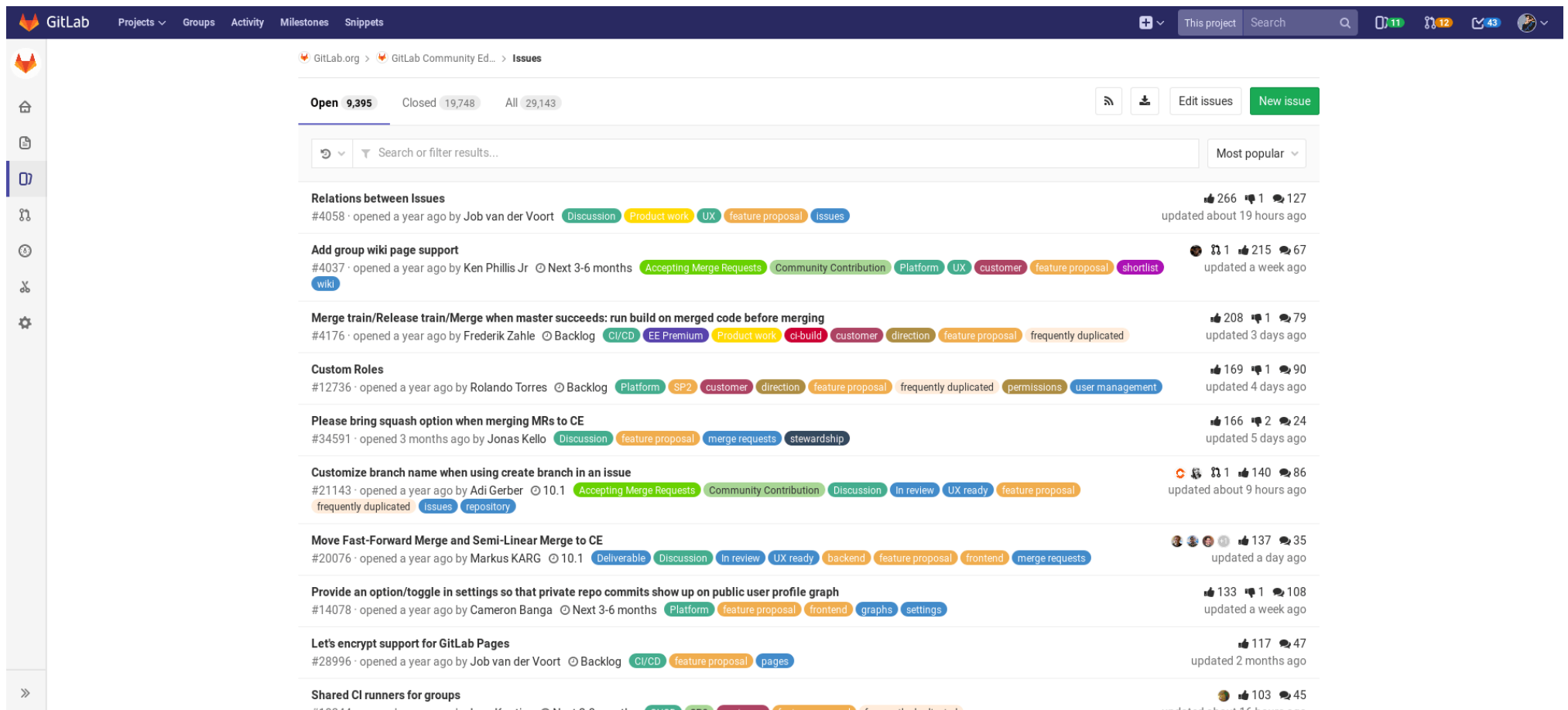
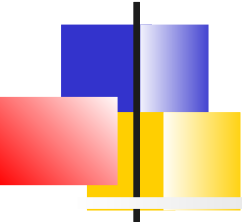
# Visualisation des issues

---

Les issues peuvent être visualisées via :

- Une **liste**. Elle affiche toutes les issues du projet ou de plusieurs projets. On peut les filtrer ou faire des actions par lots (bulk)
- Le **tableau de bord Kanban** qui affiche des colonnes en fonction des labels (par défaut statut de l'issue) ou des responsables. Les workflows sont customisable via les labels
- **Epic** : Vision transversale aux projets des issues partageant un thème, un milestone,

# Liste



The screenshot shows the GitLab web interface. The top navigation bar includes the GitLab logo, links for Projects, Groups, Activity, Milestones, and Snippets, and a search bar. The left sidebar contains icons for home, search, issues, wiki, and settings. The main content area displays the 'Issues' page for 'GitLab Community Edition'. It features a filter bar with 'Open' (9,395), 'Closed' (19,748), and 'All' (29,143) options. Below the filter bar is a search input field and a 'Most popular' dropdown. The issues are listed with their titles, IDs, authors, and various labels. Each issue entry includes a title, a description, a list of labels, and engagement metrics (likes, replies, and comments).

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 reply · 127 comments · 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 like · 215 likes · 67 comments · 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 reply · 79 comments · 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 reply · 90 comments · 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 replies · 24 comments · 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 like · 140 likes · 86 comments · 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 comments · 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 reply · 108 comments · 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 comments · updated 2 months ago

**Shared CI runners for groups**  
#10244 · opened a year ago by Juan Kertier · Next 3-6 months · CI/CD · SP2 · customer · feature proposal · frequently duplicated · 103 likes · 45 comments · updated about 16 hours ago

# Kanban

Development ▾ Search or filter results... View scope

**In dev** 80 174

- Read git data via gitaly  
In dev Plan backend  
Work for the Plan team. Covers Issues, Labels, Milestones, Boards, and more. See <https://about.gitlab.com/handbook/product/categories/>. Ping @victorwu for questions and comments.  
line in merge  
P1  
code review devops:create direction  
feature proposal frontend merge requests  
gitlab-org/gitlab-ce#18008 5
- Multiple blocking merge request approval rules  
Create Deliverable GitLab Premium In dev  
P1 UX approvals backend customer  
customer+ devops:create direction  
feature proposal frontend merge requests

**In review** 34 95

- `ExpireBuildArtifactsWorker` is broken  
In review P3 S3 Verify database  
devops:verify missed-deliverable performance  
gitlab-org/gitlab-ce#41057
- Ensure that all CI/CD queries take less than 15 seconds to complete  
Accepting merge requests In review Stretch  
Verify database devops:verify meta  
missed-deliverable performance  
gitlab-org/gitlab-ce#40524
- Further improvements to Project overview UI  
Deliverable In review Manage P2  
UX ready devops:manage direction frontend  
missed-deliverable missed:11.5 project  
release post item

**▼ Closed** 47352 19838

- include brand ai styles  
To Do  
gitlab-org/design.gitlab.com#5
- static page example  
To Do  
gitlab-org/design.gitlab.com#4
- welcome page  
To Do  
gitlab-org/design.gitlab.com#3
- add some real components  
To Do  
gitlab-org/design.gitlab.com#2

# Vue détaillée issue

The screenshot shows a GitLab issue page for "Issue #1395" opened by Marcia Ramos. The page includes a header with status buttons (Open, New issue, Close issue, Edit), a sidebar with navigation links (Todo, Assignees, Milestone, Time tracking, Due date, Labels, Weight, Participants, Notifications), and a main content area with the issue description, task list, merge requests, and comments. Numbered annotations (1-18) highlight specific UI elements: 1. "New issue" button; 2. "Close issue" button; 3. "Edit" button; 4. "Todo" button; 5. "Mark done" button; 6. "Assignees" section; 7. "Milestone" section; 8. "Time tracking" section; 9. "Due date" section; 10. "Labels" section; 11. "Weight" section; 12. "Participants" section; 13. "Notifications" section; 14. "Reference" section; 15. "Create a merge request" button; 16. "Create a branch" button; 17. "Write" button; 18. "Preview" button.

**Open** Issue #1395 opened about 14 hours ago by **Marcia Ramos** 0 of 2 tasks completed **1** **New issue** **Close issue** **Edit** **2** **Todo** **Mark done** **»**

## GitLab Issue 12

Hello World! 🍌

This is my issue's description, written in markdown (GitLab Flavored Markdown).

This is an **<h3>**

*Let's quote someone here*

Add a task list:

- ☐ Task 1
- ☐ Task 2

Mention merge requests ([gitlab-org/gitlab-ee!1784 \(merged\)](#)) and issues ([gitlab-org/gitlab-ee#2101 \(closed\)](#)) and hover over them to see their titles.

Invite users to collaborate with **@mentions**: **@marcia** **13**

Edited just now by Marcia Ramos

### 1 Related Merge Request

✓ !1784 **How to Configure LDAP with GitLab EE** in GitLab.org / GitLab Enterprise Edition **Merged** **14**

👍 0 🗑️ 0 🏠 **15**

**18** **Create a merge request**

- ✓ **Create a merge request**  
Creates a branch named after this issue and a merge request. The source branch is 'master' by default.
- Create a branch**  
Creates a branch named after this issue. The source branch is 'master' by default.

**10** **Notifications** **Unsubscribe**

**11** **Reference:** [gitlab-com/www-g...](#) 📄

**3** **Assignees** **Edit**

**4** **Milestone** **Edit**

**5** **Time tracking** **?**

**6** **Due date** **Edit**

**7** **Labels** **Edit**

**8** **Weight** **Edit**

**9** **3 participants**

**16** **Write** **Preview**

**17** **Comment** **Close issue**



# Actions sur une issue (1)

---

1. Création, Fermeture, Edition des champs de base
2. Ajouter à sa Todo List, la marquer comme terminé
3. Responsable(s) de l'issue, peut être changé à tout moment
4. Affecter une issue à un milestone
5. Temps estimé, temps passé
6. Date de livraison, peut être changée à tout moment
7. Labels. Catégorise les issues et permet de mettre en place des workflows personnalisés reflétés dans le Kanban
8. Poids. Indicateur sur l'effort nécessaire associé à l'issue



# Actions sur une issue

---

- 9. Participants. Indiqués dans la description ou qui ont participé à la discussion
- 10. Notifications. Permet de s'abonner/désabonner
- 11. Référence. Permet de copier l'URL d'accès
- 12. Titre et description (markup)
- 13. Mentions. Met en surbrillance pour la repérer facilement
- 14. Merge requests associés
- 15. emoji
- 16. Thread. Commentaires organisés en threads





# 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

Edition bulk

Import/Export d'issues

API Issues



# Labels

---

Ils permettent de catégoriser les issues ou MR.  
Par exemple : *bug*, *feature request*, ou *docs*.

- Chaque label a une couleur personnalisable.

Les **scoped labels** ont un format *clé :: valeur*.

- A un instant *t*, une issue ne peut pas avoir plusieurs labels de la même clé.
- Cela peut permettre de définir des workflows.

Exemple de scoped labels

*workflow::development*,

*workflow::review*

*workflow::deployed*



# Milestones

---

Ils permettent d'organiser les issues et MR dans un groupe cohérent, avec une date de début et une date d'échéance (facultatives).

Ils peuvent définir :

- des sprints Agile
- des releases

Ils peuvent être associés à des groupes



# Gitlab CI/CD

---

## **Jobs et Runners**

Pipelines

Directives Disponibles

Intégration Docker

Environnements et déploiements

AutoDevOps



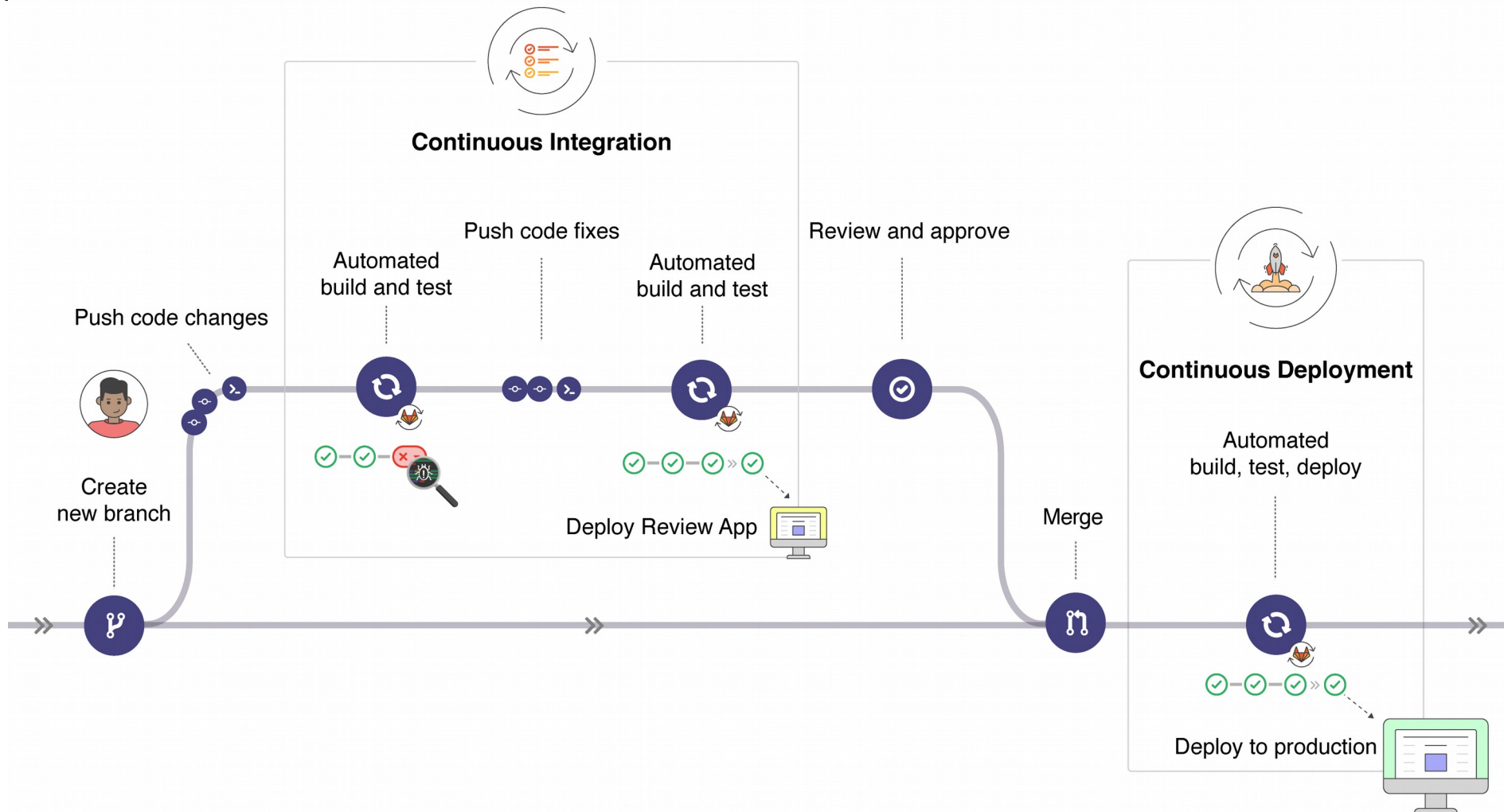
# Introduction

---

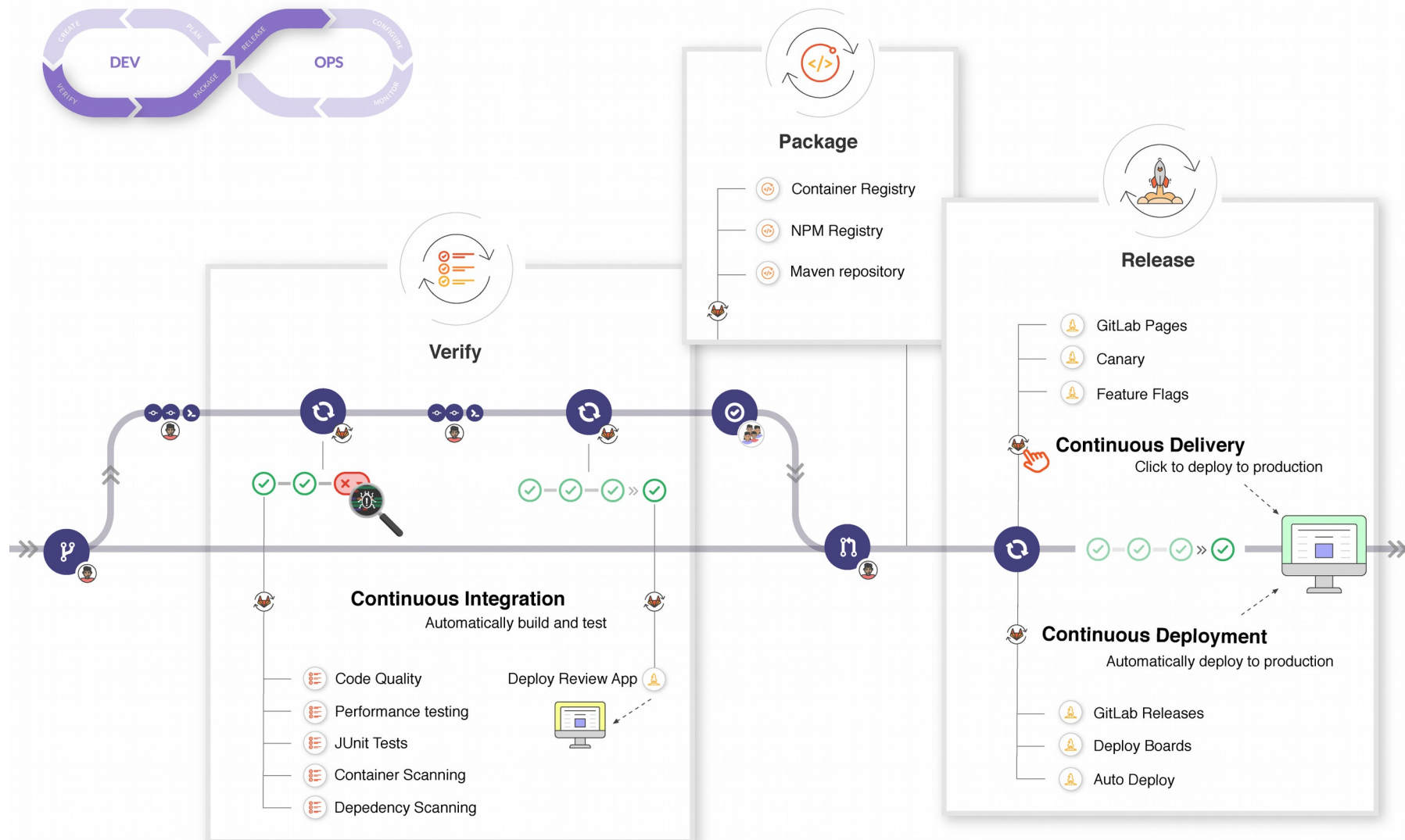
GitLab CI/CD est un outil permettant l'intégration, la livraison ou le déploiement continu

- **CI** : A chaque push, une pipeline de scripts pour construire, tester, analyser est exécutée avant de la fusionner dans la branche principale
- **CD** : A chaque push sur la branche principale, on déploie en pré-prod ou en prod

# CI/CD



# En plus détaillé





# Configuration

---

Les pipelines sont configurées via le fichier ***.gitlab-ci.yml*** placé à la racine du projet

Les scripts définis sont exécutés par des ***Runners*** qui doivent être préalablement configurés

Des gabarits de pipeline sont fournis par Gitlab  
***New File → Template***





# Jobs

---

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

Les **jobs** sont des éléments de haut-niveau avec un nom et contenant toujours le mot clé ***script***.

Les jobs sont exécutés par des **runners** différents dans des environnements indépendants des autres



# Directives de base

---

***image*** : Spécifie une image docker à utiliser pour exécuter le build.  
Possible seulement si le runner le supporte, permet d'avoir tous les outils nécessaires pour le build (*Maven, npm, gcc*)

***before-script, after-script*** : Les commandes exécutés avant/après chaque script. Permet d'initialiser le build, installer des outils

***script*** : Seul mot-clé nécessaire, contient une ou plusieurs commandes (shell Linux)



# Exemple

---

image: "ruby:2.5"

before\_script:

- apt-get update -qq && apt-get install -y -qq sqlite3 libsqlite3-dev nodejs
- ruby -v
- which ruby
- gem install bundler --no-document
- bundle install --jobs \$(nproc) "\${FLAGS[@]}"

rspec:

script:

- bundle exec rspec

rubocop:

script:

- bundle exec rubocop



# Runners

---

Un **Runner** peut être une machine virtuelle, une machine physique, un conteneur docker ou un cluster de conteneurs (*Kubernetes*).

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

Un Runner peut être spécifique à un projet ou servir à plusieurs projets.

*Settings → CI/CD*

Pour disposer d'un runner :

- Il faut l'installer
- L'enregistrer pour le 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



# Configuration runner

---

Lors d'une installation en service, la configuration est présente dans ***/etc/gitlab-runner/config.toml***

```
concurrent = 5  
check_interval = 0
```

```
[session_server]  
  session_timeout = 1800  
[[runners]]  
  name = "Another shell executeur"  
  url = "http://localhost"  
  token = "1NkCzKU1x_S6hz6VQ2Uu"  
  executor = "shell"  
[runners.custom_build_dir]  
[runners.cache]  
  [runners.cache.s3]  
  [runners.cache.gcs]
```



# Enregistrement

---

Avant la procédure d'enregistrement du runner, il faut obtenir un token via l'UI (soit partagé, soit spécifique à un projet)

Ensuite 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 exécutent les builds. Différents choix sont possibles :

- **Shell** : Le plus facile à installer mais toutes les dépendances du projet doivent être pré-installés sur le runner (git, npm, jdk, ...)
- **Virtual Machine** : Nécessite Virtual Box ou Parallels. Les outils sont pré-installés sur la VM
- **Docker** : Une image docker pour exécuter le build. Possibilité d'exécuter d'autres services docker pendant le build (une base de données par ex.)
- **Docker-machine** : Idem docker + auto-scaling. Les exécuteurs de build sont créés à la demande
- **Kubernetes** : Utilisation d'un cluster Kubernetes. Via l'API, le runner créé des pods (machine de build + services)
- **ssh** : Peu recommandé, exécute le build via ssh sur une machine distante





# Runners spécifiques ou partagés

---

Un runner peut être associé :

- Globalement à une instance de Gitlab
  - Nécessite les droits administrateurs pour obtenir le token
  - Et au niveau projet :  
*Settings → CI/CD → Allow shared Runners*
- A un groupe de projet.  
Pour obtenir le jeton, au niveau du groupe :  
*Settings → CI/CD*
- A un seul projet.  
Pour obtenir le jeton, au niveau du projet :  
*Settings → CI/CD*



# Gitlab CI/CD

---

Jobs et Runners

**Pipelines**

Directives Disponibles

Intégration Docker

Environnements et déploiements

AutoDevOps



# Pipeline

---

Les pipelines correspondent à un *.gitlab-ci.yml*.  
Elles contiennent

- Des **jobs**
- Des **stages** qui définissent quand les jobs doivent être exécutés (séquence, condition, ...)

Les jobs d'un même stage sont exécutés par des runners en parallèle, si possible

- Si tous les jobs d'un stage réussissent. La pipeline exécute la stage suivant.
- Si un job échoue, la phase suivante n'est généralement pas exécutée.



# Syntaxe

---

## **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:

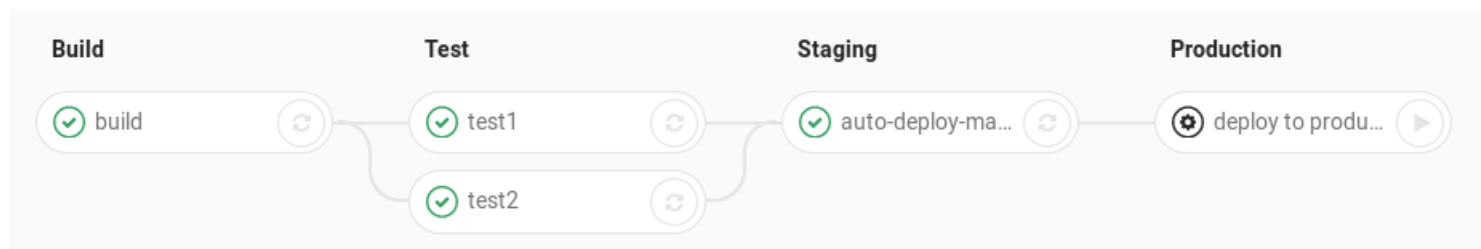
**stage: Production**

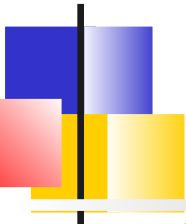
script: make deploy



# Visualisation

Dans l'UI, les pipelines sont visualisées graphiquement





# Variables d'environnement

---

GitLab CI/CD fournit un ensemble prédéfini de variables d'environnement (Id d'issue, commit ID, branch ...)

```
test_variable:  
  stage: test  
  script:  
    - echo $CI_JOB_STAGE
```

Le build peut utiliser des variables d'environnement spécifiques qui sont fixées via l'UI ou directement dans *.gitlab-ci.yml*

- *Settings → CI/CD → Variables*

- variables:

```
TEST: "HELLO WORLD"
```



# Configuration variables

---

2 types de variables sont supportés :

- “*Variable*”: Le Runner crée une variable d’environnement du nom de la variable
- “*File*”: Le Runner écrit la valeur de la variable dans un fichier temporaire et positionne le chemin du fichier comme la valeur d’une variable d’environnement du nom de la variable.

La variable peut également être configurée comme étant masquée, sa valeur n’apparaît pas dans les logs



# Artifacts

---

Les ***artifacts*** sont une liste de fichiers et répertoires attachés à un job terminé.  
Ils sont téléchargeable (tar.gz) via l'UI  
Ils sont conservés 1 semaine (par défaut)

pdf:

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

```
artifacts:
```

```
  paths:
```

```
    - mycv.pdf
```

```
  expire_in: 1 week
```





# Réutilisation des artefacts

---

La directive ***dependencies*** permet d'indiquer une dépendance entre 2 jobs.

Elle a pour effet de récupérer les artefacts générés par la dépendance.

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
```



# Cache des dépendances

---

Le cache des dépendances permet d'accélérer l'exécution des jobs.

Particulièrement utile lorsque le projet utilise de nombreuses librairies sur Internet

Les caches sont désactivés si ils ne sont pas définis globalement ou par projet

- Globalement, le cache est réutilisé entre jobs de différentes pipeline
- Par projet, le cache est utilisé
  - par la prochaine pipeline, par le job l'ayant défini
  - Par un job aval qui définit un cache du même nom.

Les caches sont gérés par les Runner (éventuellement téléchargé sur S3 si cache distribué).



# Example

---

```
#
# https://gitlab.com/gitlab-org/gitlab-ce/tree/master/lib/gitlab/ci/templates/
#   Nodejs.gitlab-ci.yml
#
image: node:latest

# Cache modules in between jobs
cache:
  key: ${CI_COMMIT_REF_SLUG}
  paths:
    - node_modules/

before_script:
  - npm install

test_async:
  script:
    - node ./specs/start.js ./specs/async.spec.js
```



# Exécution des pipelines

---

Les pipelines s'exécutent  
automatiquement à chaque push

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

*Settings → CI/CD → Schedules*



# Gitlab CI/CD

---

Jobs et Runners

Pipelines

**Directives Disponibles**

Intégration Docker

Environnements et déploiements

AutoDevOps



# Variables

---

Le mot-clé ***variables*** permet de définir des variables qui sont ensuite transmises à l'environnement du job.

Elles peuvent être définies globalement ou par job (surcharge le global).

`variables:`

`DATABASE_URL: "postgres://postgres@postgres/my_database"`

Le runner définit également des variables, par exemple : *CI\_COMMIT\_REF\_NAME*

Plus les variables positionnés dans l'UI



# ***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 (par défaut true)

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. 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

**only** et **except** limitent l'exécution d'un job à une branche ou une tag. Il est possible d'utiliser des expressions régulières



# Exemples

---

**#Toutes les refs démarrant avec issue-, mais pas les branches**

```
job:
  only:
    - /^issue-.*$/
  except:
    - branches
```

**#Seulement les tags, une API ou une planification**

```
job:
  only:
    - tags
    - triggers
    - schedules
```

**#Seulement les branches en fonction d'une variable**

```
deploy:
  script: cap staging deploy
  only:
    refs:
      - branches
  variables:
    - $RELEASE == "staging"
    - $STAGING
```



# 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
```





# Gitlab CI/CD

---

Jobs et Runners

Pipelines

Directives Disponibles

**Intégration Docker**

Environnements et déploiements

AutoDevOps



# Docker

---

Le mot-clé ***image*** spécifie l'image docker à utiliser pour exécuter la pipeline.

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"]
```

La clé *entrypoint* définit la commande à exécuter au démarrage du container, équivalent à l'argument *--entrypoint* de la commande *docker*



# Docker services

---

Le mot-clé ***services*** définit des autres images exécutées durant le build et liée à l'image principale. 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

---

Il y a 3 possibilités afin de permettre l'exécution de commande docker durant le build :

- Avec l'exécuteur shell et une pré-installation de docker sur le runner
- Avec l'exécuteur docker et l'image docker, ainsi que le service docker-in-docker
- Avec l'exécuteur docker, une pré-installation de docker sur le runner et une redirection de socket docker

Attention : Pour ces 3 techniques le serveur gitlab doit être accessible des containers  
=> Il doit avoir une adresse publique



# 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 privilégié

```
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)

---

## Pipeline :

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

---

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:lates
```



# Gitlab CI/CD

---

Jobs et Runners

Pipelines

Directives Disponibles

Intégration Docker

**Environnements et déploiements**

AutoDevOps



# 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 job 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.

*Operations → Environments*

Le nom de l'environnement est accessible via le job par la variable `$CI_ENVIRONMENT_NAME`



# Example

---

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



# Déploiement manuel

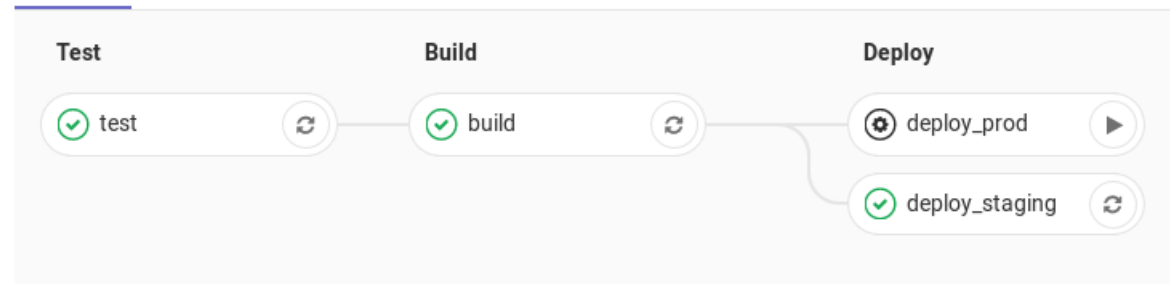
L'ajout de *when: manual* convertit le job en un job manuel et expose un bouton Play dans l'UI

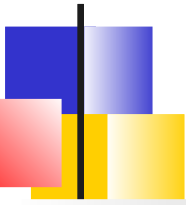
## Use busybox

🕒 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

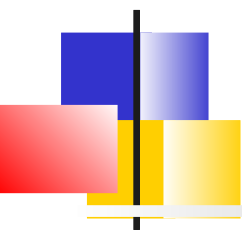


# Example

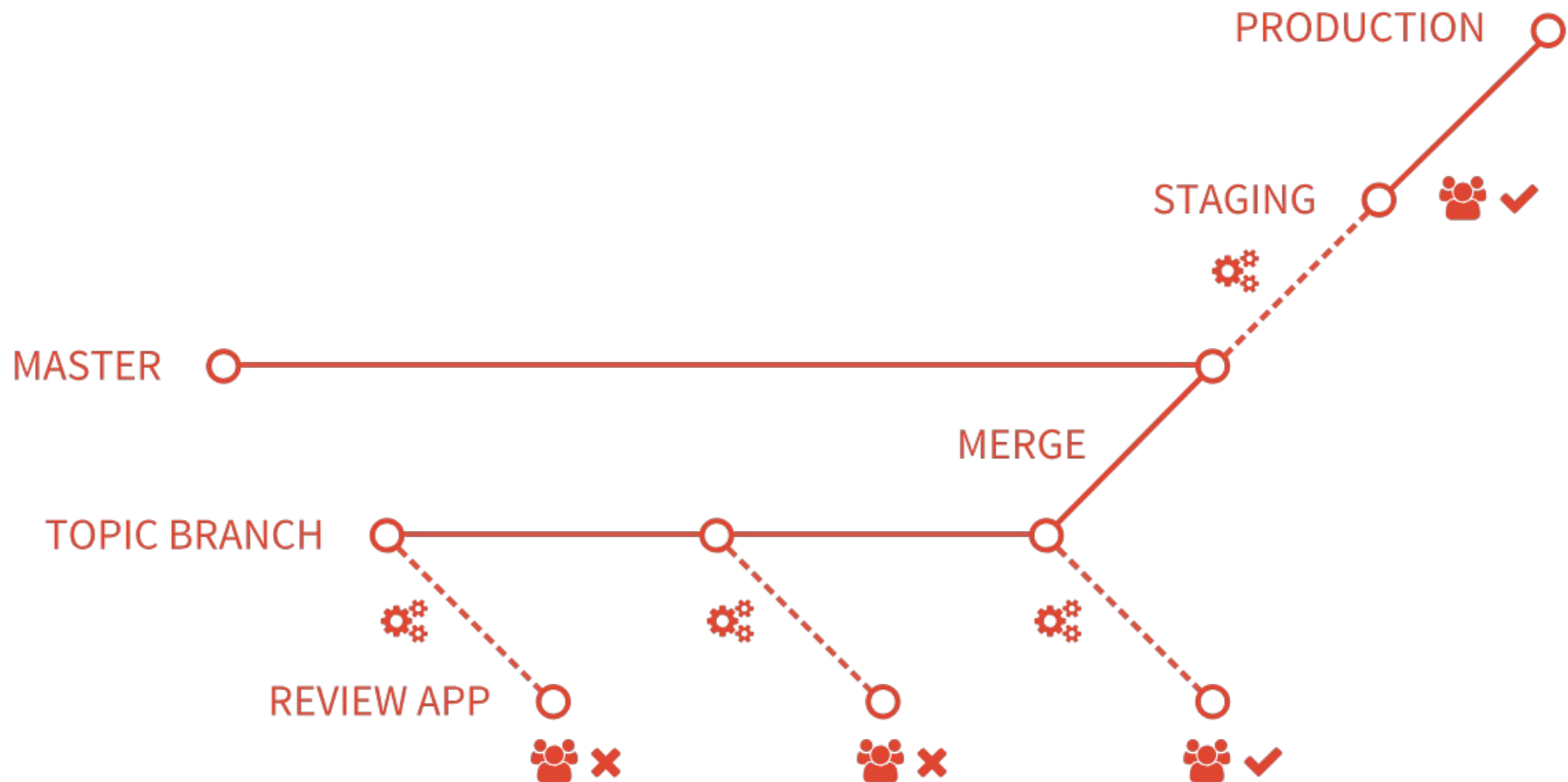
---

```
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

---

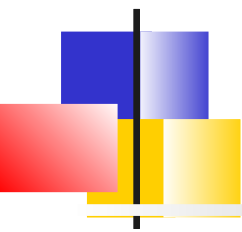
```
stages:
  ..
  - deploy

...

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
```



# Gitlab CI/CD

---

Jobs et Runners  
Pipelines  
Directives Disponibles  
Intégration Docker  
Environnements et déploiements  
**AutoDevOps**



# Auto Devops



# Introduction

---

***Auto DevOps*** fournit une configuration CI / CD prédéfinie qui permet de détecter la nature du projet et d'appliquer un cycle full DevOps automatiquement.

*Auto DevOps* est activé par défaut sur les projets, il se désactive automatiquement au premier échec de pipeline

*Auto DevOps* peut également être explicitement activé



# Pré-requis

---

## Pré-requis nécessaire :

- GitLab Runner (Pour toutes les phases) : Doit être configuré pour utiliser Docker ou l'exécuteur Kubernetes and mode privilégié
- Base Domain (Pour les review apps) : Un domaine configuré avec un DNS \* utilisé par tous les projets
- Kubernetes (GKE ou Existant) : Pour les déploiements
- Prometheus : Pour obtenir les métriques
- Helm : Gitlab utilise Helm pour accéder au cluster Kubernetes





# Stratégies

---

La configuration d'AutoDevOps permet de choisir parmi 3 stratégies de déploiement :

- CD vers la production
- CD incrémentale vers la production (les containers sont progressivement déployés)
- Déploiement automatique vers la pré-prod et déploiement manuel en production



# Phases (1)

---

**Auto Build** : Crée un build en utilisant un Dockerfile ou les buildpacks Heroku. L'image est poussée et taggée vers le registre de conteneur du projet

**Auto Test** : Exécute les tests appropriés si il détecte les langages de votre projet

**Auto Code Quality** : Exécuter une analyse statique et autres vérification du code

**Auto SAST** : Exécute une analyse statique pour détecter des vulnérabilités

**Auto dependency** : Vérifie les dépendances du projet et les éventuelles failles de sécurité

**Auto License Management** : Génère un rapport sur les dépendances utilisées et leurs licences

**Auto Container Scanning** : Analyse les failles de sécurité dans les images Docker



# Phases (2)

---

***Auto Review Apps*** : Déploie vers un cluster Kubernetes

***Auto DAST*** : Test dynamique de la sécurité avec OWASP ZAPProxy

***Auto Browser Performance Testing*** : Test de la performance d'une page web avec l'image Sitespeed.io

***Auto Deploy*** : Déploiement en production par défaut.  
Possibilité de faire du canary testing

***Migrations*** : Possibilité de configuration de scripts de migration Postgres

***Auto Monitoring*** : Monitoring de l'application déployée via Prometheus (pré-déployé sur le cluster)



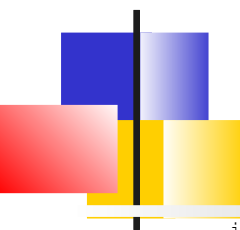
# Personnalisation

---

Il est possible de personnaliser la pipeline via

- Des buildpacks Heroku personnalisés
- Un Dockerfile personnalisé à la racine du projet
- Des graphiques Helm
- Ou en copiant la configuration dans le fichier de pipeline

*New File → Template AutoDevOps*



# Template AutoDevOps (1)

---

```
image: alpine:latest
```

```
variables:
```

```
  # KUBE_INGRESS_BASE_DOMAIN is the application deployment domain and should be set as a variable at the group or project level.
```

```
  POSTGRES_USER: user
```

```
  POSTGRES_PASSWORD: testing-password
```

```
  POSTGRES_ENABLED: "true"
```

```
  POSTGRES_DB: $CI_ENVIRONMENT_SLUG
```

```
  POSTGRES_VERSION: 9.6.2
```

```
  KUBERNETES_VERSION: 1.11.10
```

```
  HELM_VERSION: 2.14.0
```

```
  DOCKER_DRIVER: overlay2
```

```
  ROLLOUT_RESOURCE_TYPE: deployment
```

```
stages:
```

- build
- test
- deploy # dummy stage to follow the template guidelines
- review
- dast
- staging
- canary
- production
- incremental rollout 10%
- incremental rollout 25%
- incremental rollout 50%
- incremental rollout 100%
- performance
- cleanup



# Template *AutoDevOps* (2)

---

```
include:
  - template: Jobs/Build.gitlab-ci.yml
  - template: Jobs/Test.gitlab-ci.yml
  - template: Jobs/Code-Quality.gitlab-ci.yml
  - template: Jobs/Deploy.gitlab-ci.yml
  - template: Jobs/Browser-Performance-Testing.gitlab-ci.yml
  - template: Security/DAST.gitlab-ci.yml
  - template: Security/Container-Scanning.gitlab-ci.yml
  - template: Security/Dependency-Scanning.gitlab-ci.yml
  - template: Security/License-Management.gitlab-ci.yml
  - template: Security/SAST.gitlab-ci.yml

# Override DAST job to exclude master branch
dast:
  except:
    refs:
      - master
```