





# GitLab CI/CD

---

David THIBAU – 2019

david.thibau@gmail.com



# Agenda

---

## **Introduction**

- Agilité et DevOps
- CI et CD
- Architecture des systèmes
- La plateforme Gitlab

## **Pilotage de projet**

- Groupes, Projets et membres
- Issues et Milestones

## **Gestion des sources et collaboration**

- Particularités Gitlab
- MergeRequest
- Déclinaisons de GitlabFlow

## **Concepts pipelines CI/CD**

- Introduction
- Jobs et Runners
- UI pipelines
- Basiques Pipelines
- Directives disponibles
- Environnements et déploiements
- Intégration docker

## **Phases des pipelines et support Gitlab**

- Construction et tests développeur
- Analyses statiques
- Dépôts d'artefacts
- Déploiement et release
- Tests post-déploiements
- Gestion de l'infrastructure



# Introduction

---

## **Agilité et DevOps**

Pipelines CI/CD

Architecture et Infrastructure

La plateforme Gitlab



# L'agilité

---

- Le terme agile (regroupant de nombreuses méthodes) est consacré par le manifeste Agile : <http://agilemanifesto.org/> 2001
  - Personnes et interactions plutôt que processus et outils
  - Logiciel fonctionnel plutôt que documentation complète
  - Collaboration avec le client plutôt que négociation de contrat
  - Réagir au changement plutôt que suivre un plan



# Méthodes agiles

---

2 facteurs communs à toutes les méthodes agiles :

- la mise en place de pratiques **itératives**,
- des projets plus **petits**  
=> des équipes de dév. de plus en plus petites, des périmètres fonctionnels limités

→ C'est ce l'on retrouve dans *RUP, XP Programming, Scrum, Safe, Spotify, etc..*



# Contraintes sur la fréquence des déploiements

---

L'agilité suppose d'augmenter la fréquence des déploiements dans les différents environnements : intégration, recette, production afin :

- De fiabiliser les processus de déploiement
- De mieux piloter le projet
- De prendre en compte rapidement les retours utilisateurs

Problème : Avant DevOps, l'organisation des services informatiques ne facilitait pas les déploiements



# Le constat DevOps

---

Les différents objectifs donnés à des équipes qui se parlent peu créent des tensions et des dysfonctionnements dans le processus de mise en production d'un logiciel.

=> Pour l'équipe Ops, l'équipe de développement devient responsable des problèmes de qualité du code et des incidents survenus en production.

=> L'équipe Dev blâme son alter ego Ops pour sa lenteur, les retards et leur méconnaissance des livrables qu'elle manipule





# Approche *DevOps*

---

*DevOps* vise l'alignement des équipes par la réunion des "Dev engineers" et des "Ops engineers" .

Cela impose :

- la réunion des équipes
- la montée en compétence des différents profils.  
=> Arrivée des profils *full-stack*

Une équipe *DevOps* est une équipe qui regroupe toutes les compétences nécessaires à un projet<sup>1</sup> et qui est complètement indépendante



# Pratiques *DevOps* (1)

---

- Un déploiement régulier/continu des applications dans les différents environnements.  
La répétition fiabilise le processus ;
- Un décalage des tests "vers la gauche".  
Autrement dit de tester au plus tôt ;
- Une pratique des tests dans un environnement similaire à celui de production ;
- Une intégration continue incluant des "tests continus" (à chaque commit);



# Pratiques *DevOps* (2)

---

- Une boucle d'amélioration courte  
i.e. un feed-back rapide des utilisateurs ;
- Une surveillance étroite de l'exploitation et de la qualité de production actualisée par des métriques et indicateurs "clé".
- La production de métriques reflétant la qualité du projet
- Une unique source de vérité : Le dépôt de source inclut le code de build, de test, d'infrastructure et ... le code source



# « As Code »

---

Les outils *DevOps* permettent d'automatiser/exécuter toutes les tâches nécessaires à la construction d'un logiciel de qualité (build, test, provisionnement) à partir de l'unique point central de vérité

On parle alors de *Build As Code*,  
*Infrastructure As Code*, *Pipeline As Code*,  
*Load Test As Code*, ...



# Objectif ultime

---

- 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



# Introduction

---

Agilité et DevOps

**Pipelines CI/CD**

Architecture et Infrastructure

La plateforme Gitlab



# 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 (intégration, tests, déploiement) sont essayées.

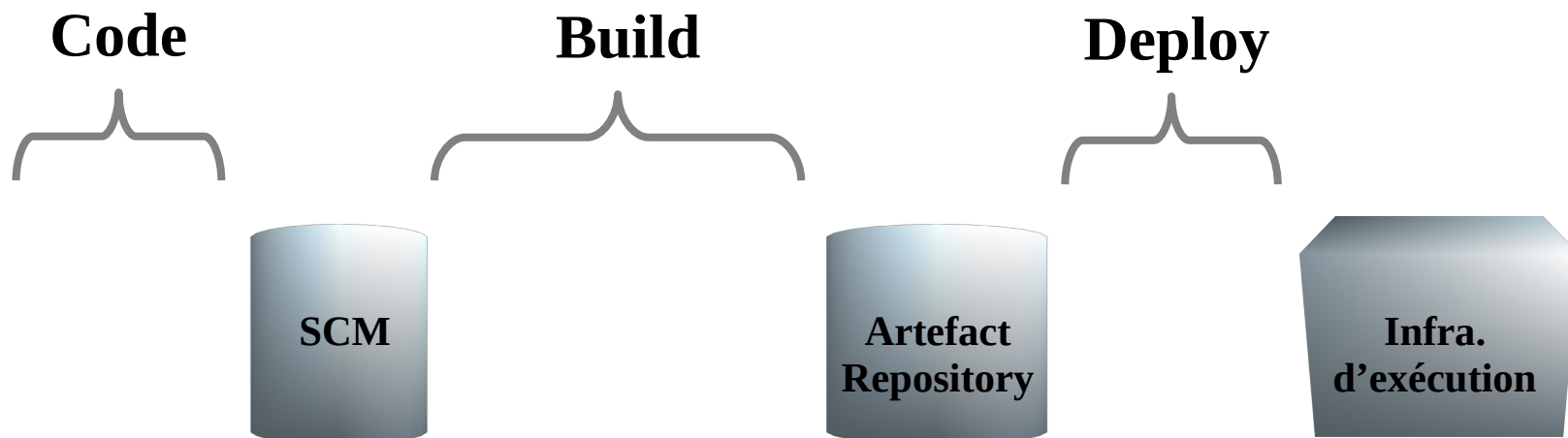
En fonction de leurs succès, l'application est déployée dans les différents environnements (intégration, staging, production)



# Cycle de vie du code Build / Release / Run

La pipeline suit le cycle de vie du code.

- 1) Le code est testé localement puis poussé dans le **dépôt de source**
- 2) Le build construit l'artefact et si le build est concluant stocke une release dans un **dépôt d'artefact**
- 3) L'outil de déploiement accède aux différentes release et les déploie sur l'**infra d'exécution**







# Build is tests !

---

La construction de l'application consiste principalement à :

- Packager le code source dans un format exécutable, déployable
- Effectuer toutes les vérifications automatiques permettant d'avoir confiance dans l'artefact généré et autoriser son déploiement  
(Tests Unitaires/Intégration, Fonctionnel, Performance, Sécurité, Licenses, ...)



# Pipelines

---

Les étapes de construction automatisées sont séquencées dans une **pipeline**.

- Une étape est exécutée seulement si les étapes précédentes ont réussi.
- Les plate-formes d'intégration/déploiement continu ont pour rôle de démarrer et observer l'exécution de ces pipelines



# Distinction CI/CD

---





# Outil de communication

---

La Plateforme a pour vocation de publier les résultats des builds :

- Nombre de tests exécutés, succès/échecs
- Couverture des tests
- Complexité, Vulnérabilités du code source
- Performance : Temps de réponse/débit
- Documentation, Release Notes
- ...

Les métriques sont visibles de tous en toute transparence

- => Confiance dans ce qui a été produit
- => Motivation pour s'améliorer



# Phases de la mise en place

---

La mise en place de pipeline de CI/CD passe généralement par plusieurs phases :

1. Mise à disposition d'une PIC
2. Automatisation tests unitaires et d'intégration
3. Déploiement dans un environnement d'intégration (review app)
4. Mise en place d'analyse de code, de tests fonctionnels, performance, d'acceptation automatisés, Collecte des métriques
5. Renforcement des tests, Release automatique, déploiement en QA
6. Renforcement des tests d'acceptation, Validation de déploiements
7. Totale confiance dans les tests permet le Déploiement continu



# Introduction

---

Agilité et DevOps  
Pipelines CI/CD

**Architecture et Infrastructure**

La plateforme Gitlab



# Introduction

---

Avec DevOps une nouvelle architecture de systèmes visant à améliorer la rapidité des déploiements des retours utilisateur est apparu : les « ***micro-services*** »

C'est le même objectif que l'approche *DevOps* : « *Déployer plus souvent* »



# Architecture

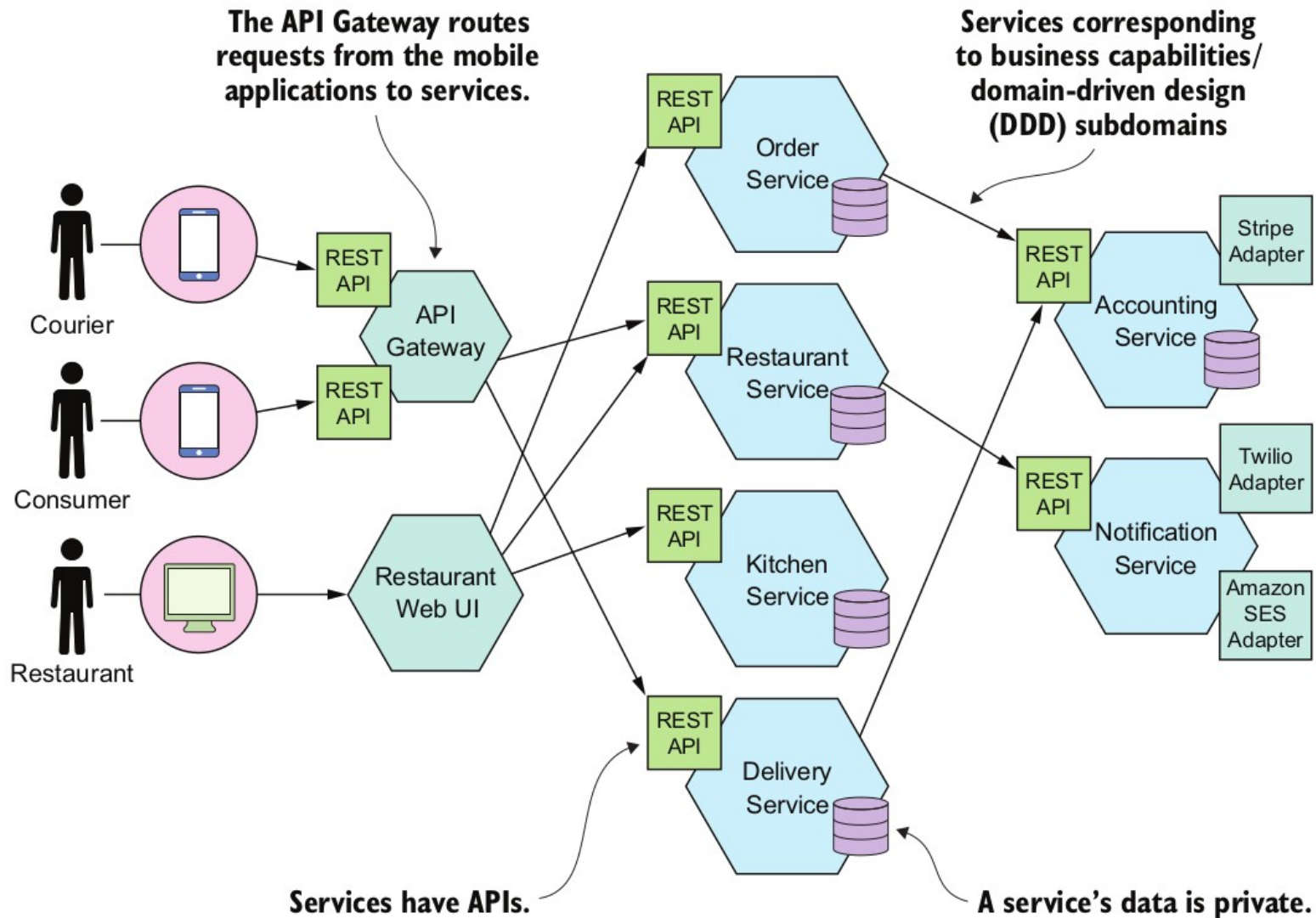
---

Une architecture micro-services implique la décomposition des applications en petits services

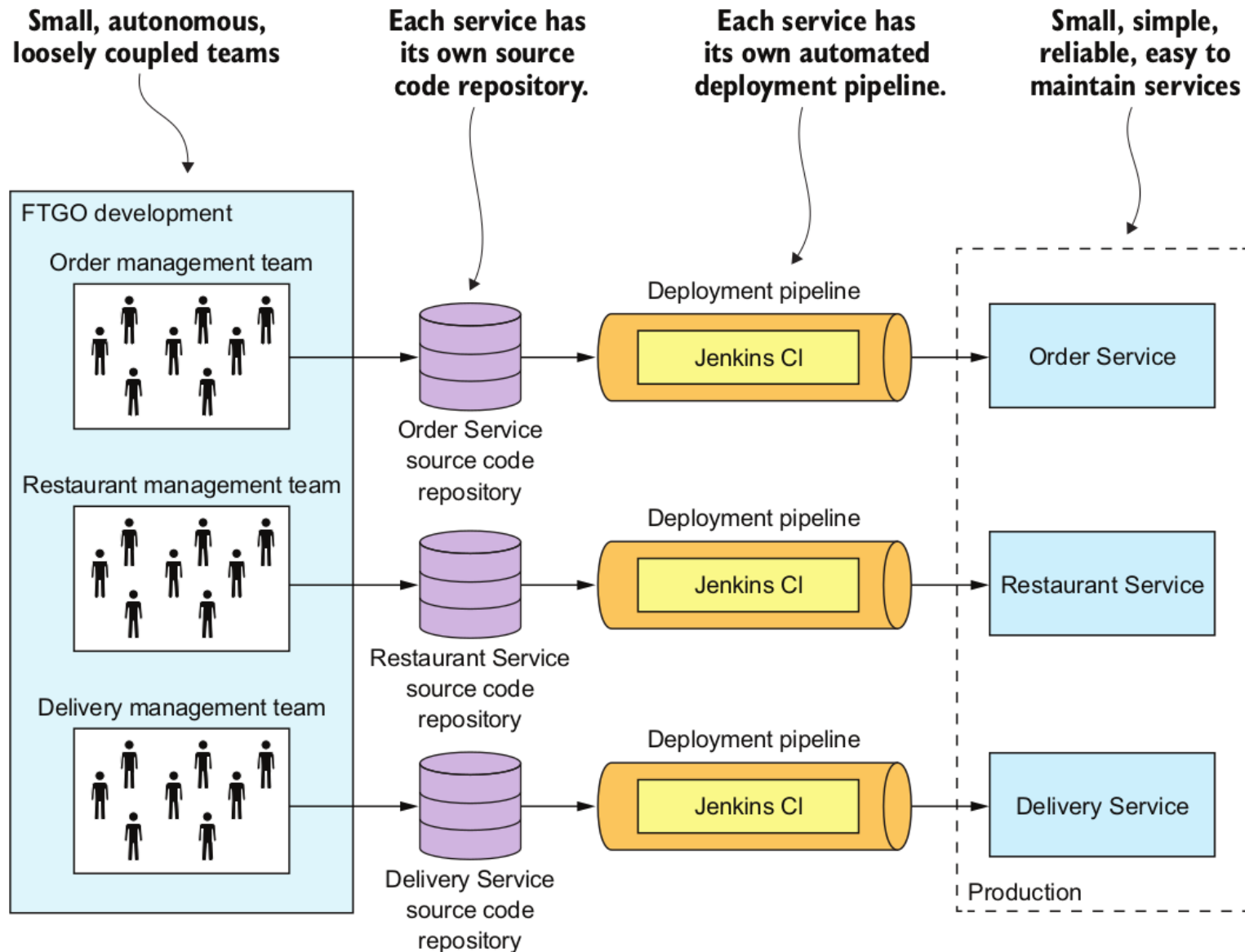
- faiblement couplés
- ayant une seule responsabilité métier
- Développés par des équipes full-stack indépendantes.

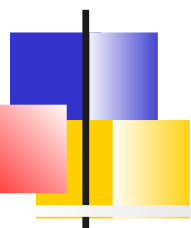


# Une architecture micro-service



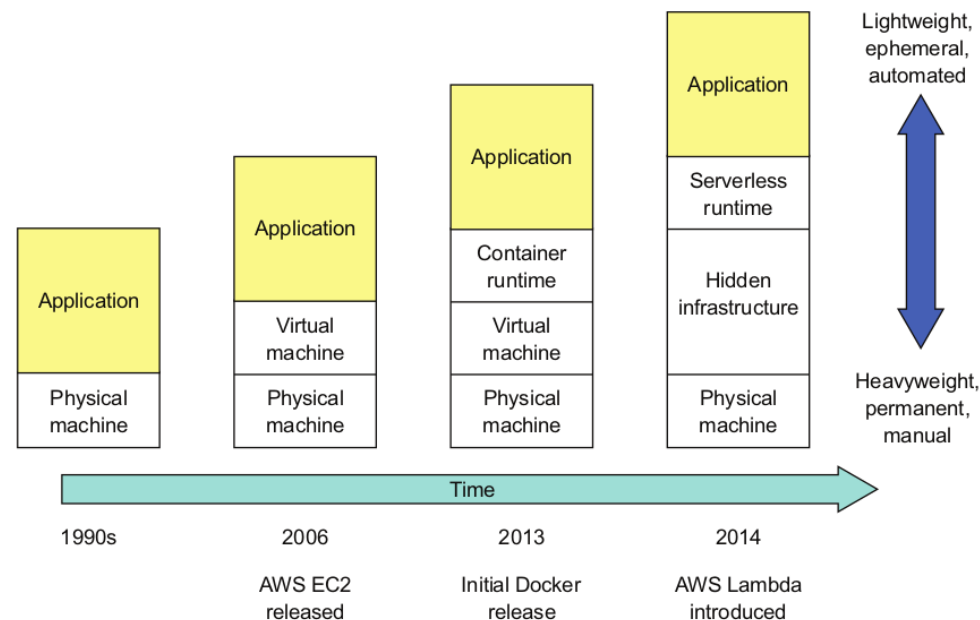
# Organisation DevOps





# Infrastructure de déploiement

Même si plusieurs alternatives peuvent être envisagées, l'utilisation des technologies de container et d'orchestrateur de container sont plus adaptées





# Service comme Container

---

## ***Deploy a service as a container***

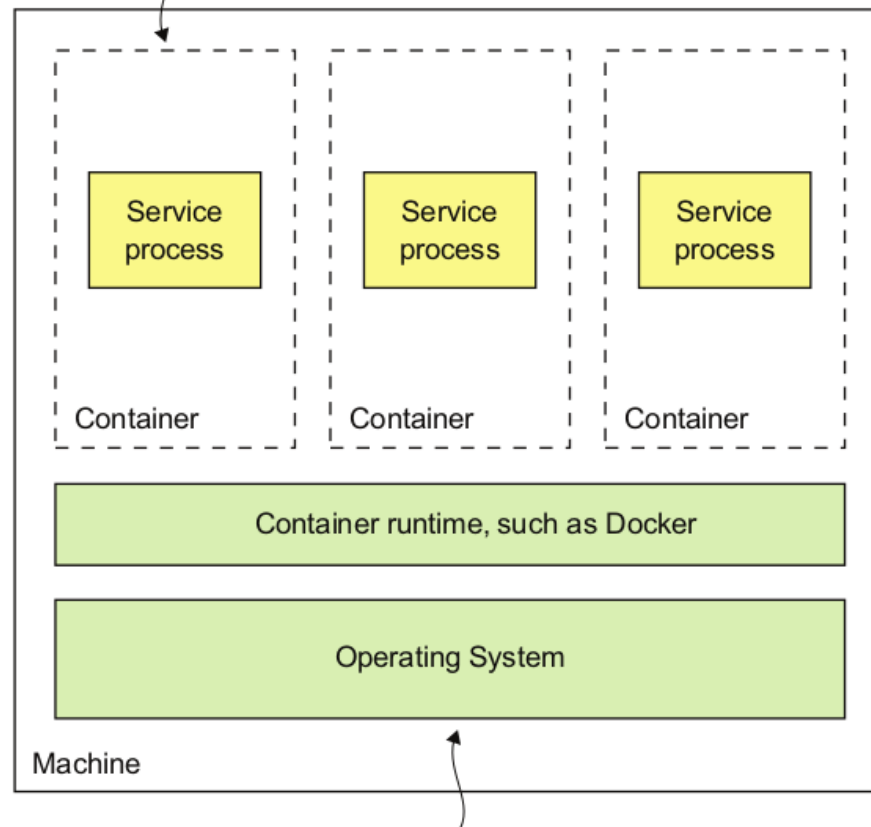
***Pattern***<sup>1</sup> : Déployé les services packagés comme des images de conteneur. Chaque service est un conteneur

Le packaging en image fait partie de la pipeline de déploiement

1. <http://microservices.io/patterns/deployment/service-per-container.html>

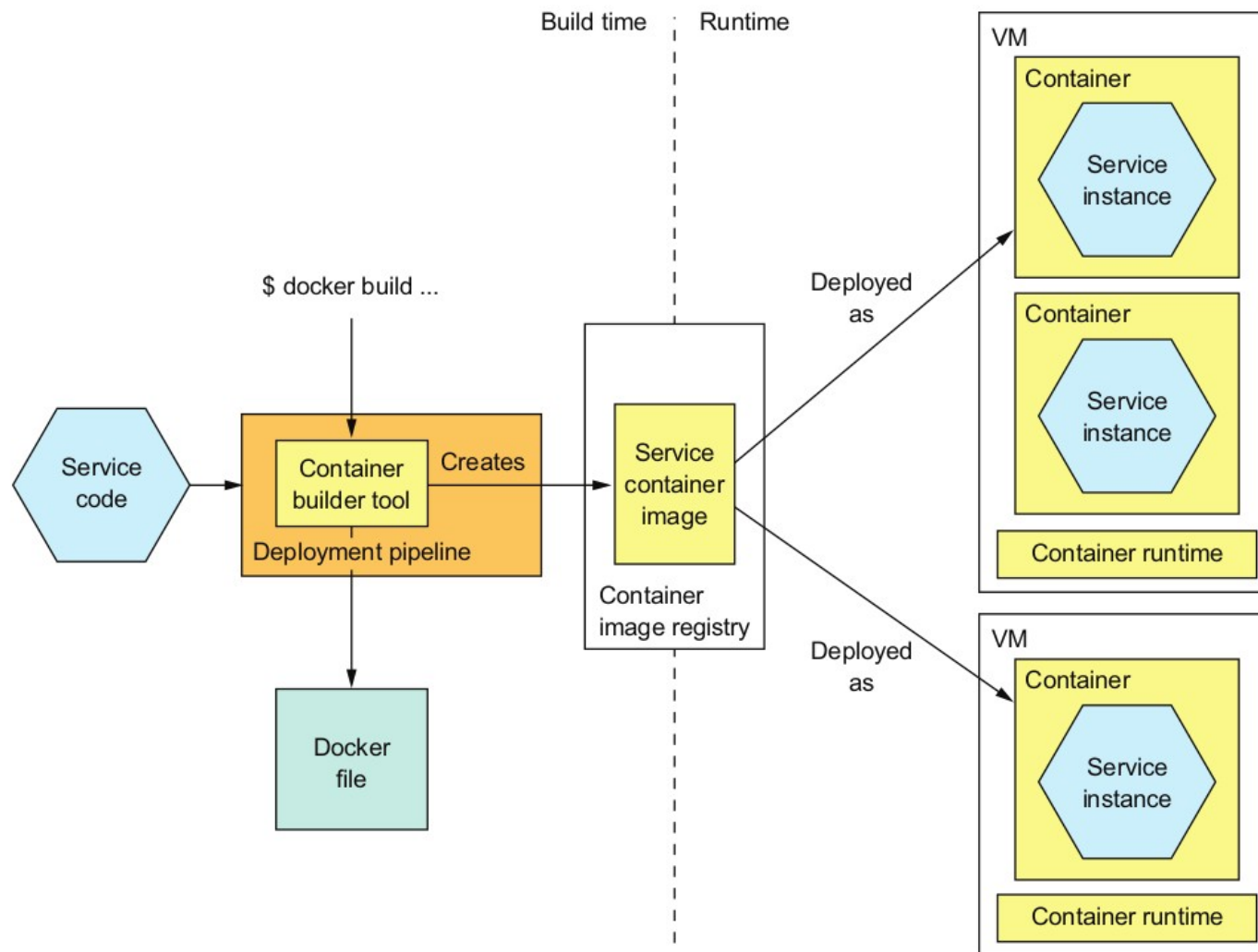
# Exécution

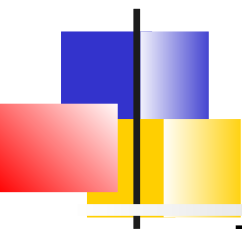
Each container is a sandbox that isolates the processes.



Shared by all of the containers

# Déploiement





# Bénéfices / Inconvénients

---

## Bénéfices

Encapsulation de la pile technologique. Déploiements immuables

Les instances de service sont isolées.

Les ressources des instances de service sont limitées.

## Inconvénients

Équipe DevOps responsable de l'administration des images du conteneur. (Patches de l'OS par exemple)

Administrer l'infrastructure du conteneur-runtime et éventuellement des VMs associés



# Apports des déploiement immuables

---

Facilite le provisionnement

Permet facilement des déploiements  
blue-green sans interruption de service

Facilite le roll-back

Permet le canary deployment (plusieurs  
versions déployées simultanément)

Multiplie les environnements (un  
environnement par branche)





# Introduction

---

Agilité et DevOps

Pipelines CI/CD

Architecture et Infrastructure

**La plateforme Gitlab**



# Introduction

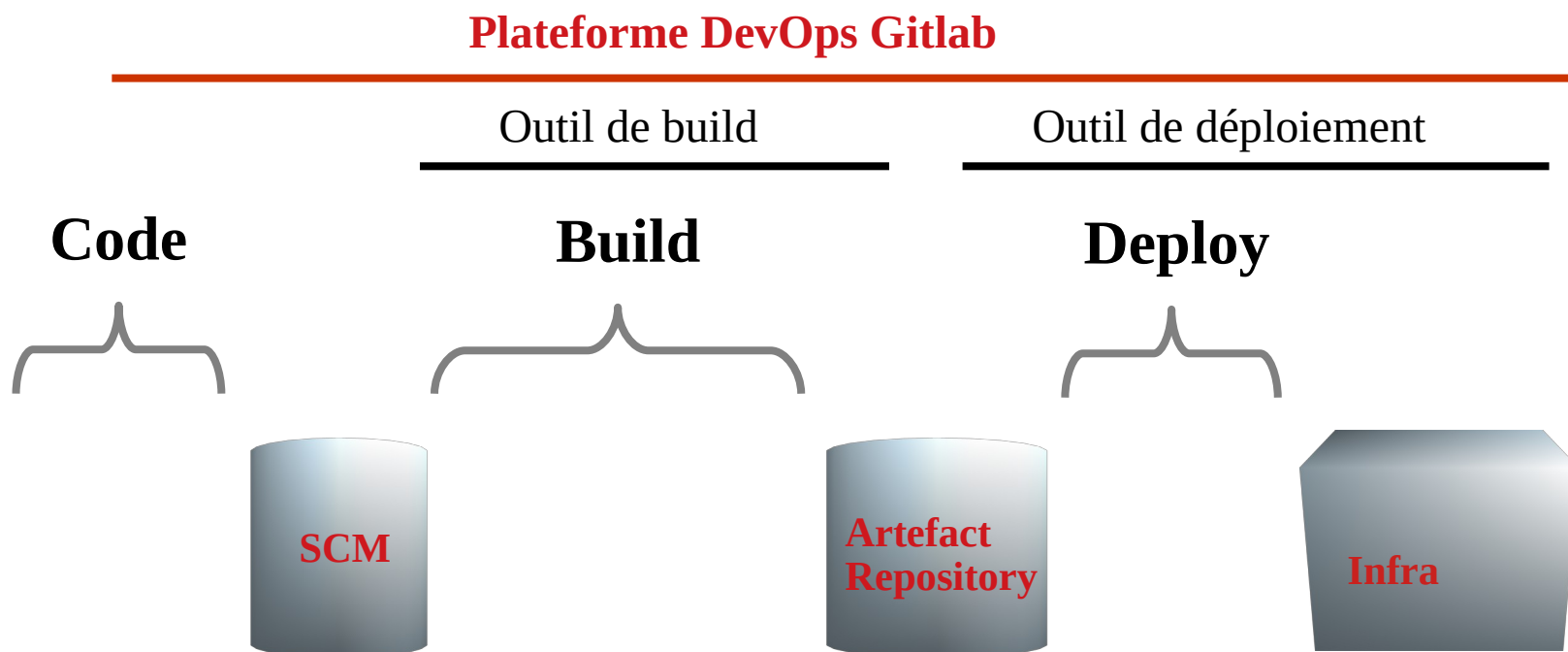
---

Gitlab se définit comme une plateforme de DevOps complète qui inclut :

- La gestion des codes sources
- Le pilotage de projet agile
- Les 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



# Gitlab et Build/Release/Run





# Intégration GitLab

---

*Gitlab* peut également s'intégrer avec d'autres outils tierces qui pourront prendre une part du Devops

- Annuaire utilisateurs et sécurité (LDAP, OpenID, ...)
- Pilotage de projet (Jira, ...)
- Communication projet (Slack, Gmail, ...)
- Dépôt d'artefact (Nexus, Artifactory, Docker registry, ... )
- Outils d'analyse (Sonarqube, ...)
- Exécution de pipeline (Jenkins, ...)
- Infrastructure (Terraform, Kubernetes, Cloud)
- ...



# 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 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

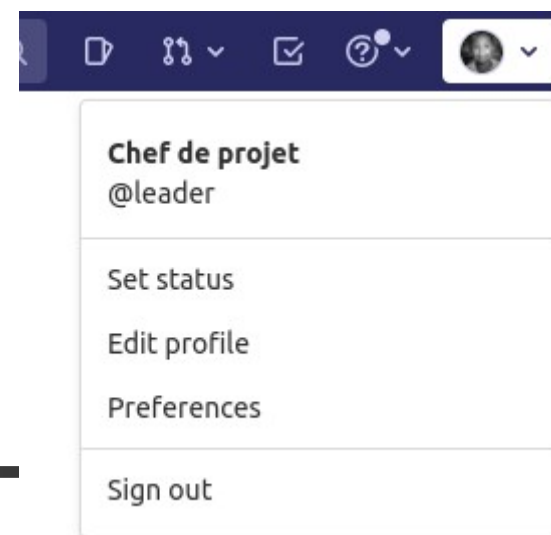
---

2 interfaces utilisateurs sont disponibles :

- 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 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 :

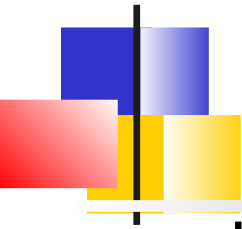
```
ssh -T git@gitlab.com
```



# Pilotage de projet DevOps

---

**Projets, Groupes et Membres Gitlab**  
Issues, milestones et tableaux de bord



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



# Groupes

---

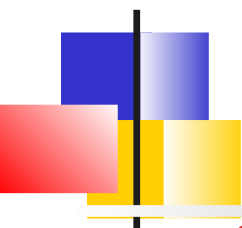
Afin de faciliter la gestion des projets et des membres associés, il est possible de définir des **groupes de projets**.

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

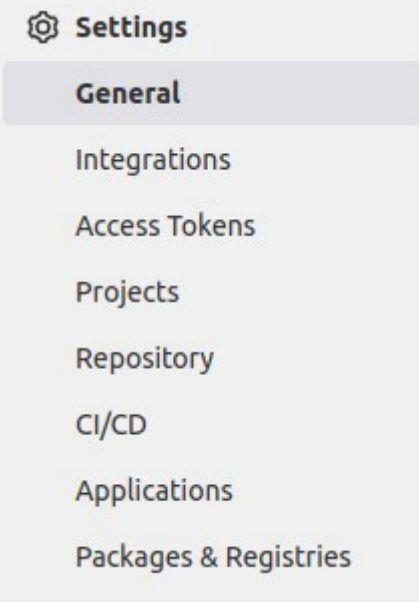
Il sera possible de visualiser toutes les issues et Merge Request des projets du groupe

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 :

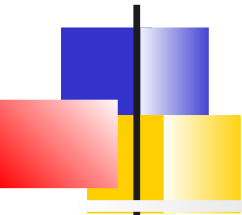
- D'un projet vierge
- D'un projet à partir d'un gabarit contenant déjà certains fichiers
- En important un projet d'un autre dépôt Git

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



# Outils de pilotage

---

Scopes des outils  
**Issues et milestones**



# Issues et Milestones

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

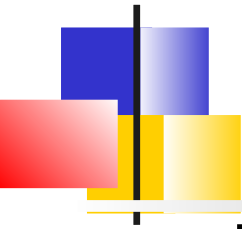
Une **issue** peut ainsi représenter :

- 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

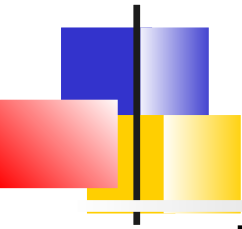


# Visualisation des issues

---

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.**

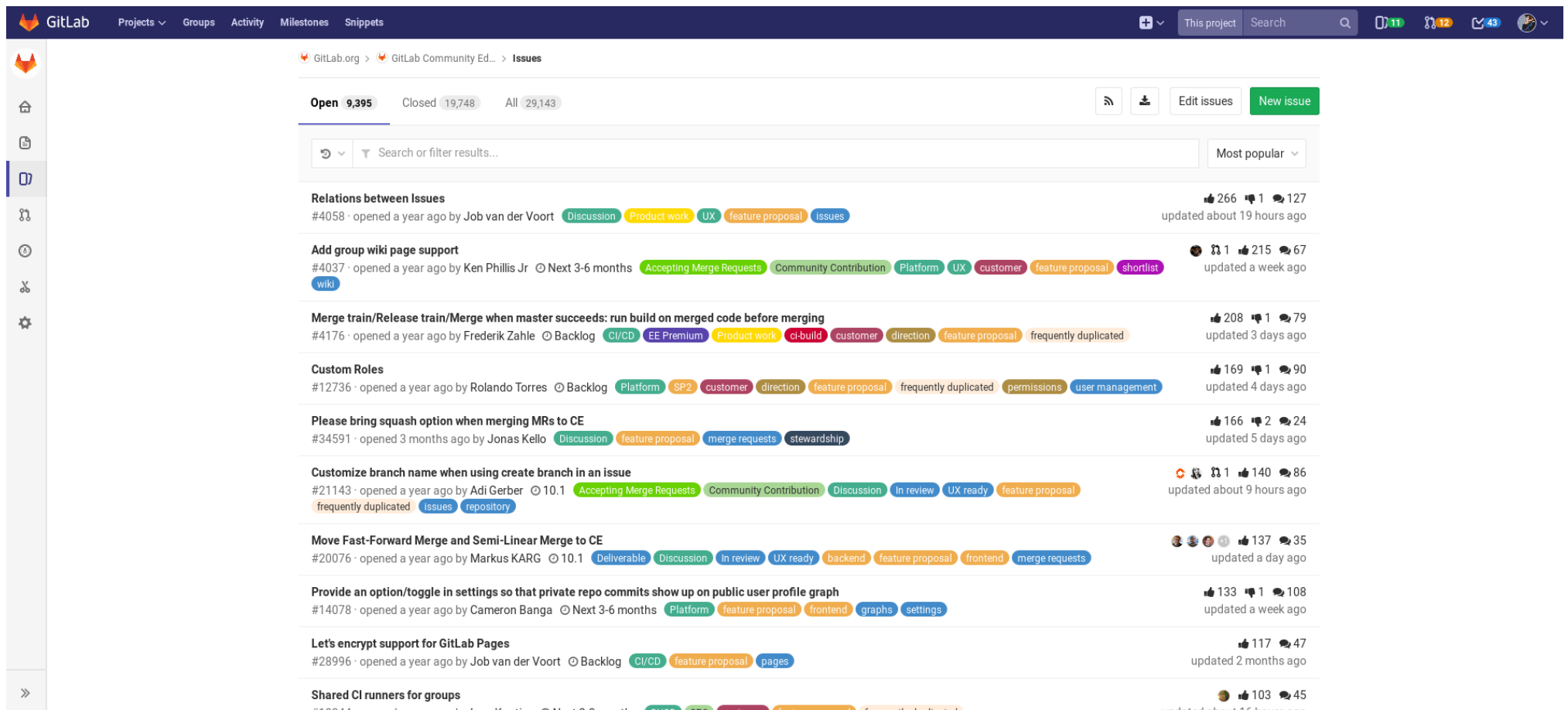
Plusieurs labels peuvent être associés à la même issue

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

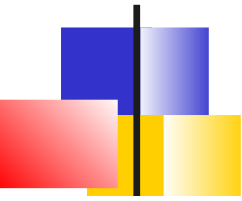
**Let's encrypt support for GitLab Pages**

#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

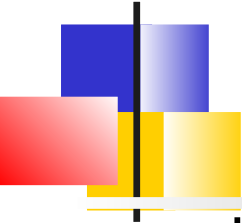
Créer une maquette pour la page Historique

#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



# Gestion des sources et collaboration

---

**Particularités Gitlab**  
MergeRequest  
Déclinaisons de GitlabFlow



# 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) lorsque l'on accède au projet

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.

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

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 (`#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





# 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 ***main*** comme branche par défaut.

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

C'est en général dans la branche par défaut que sont fusionnées les modifications effectuées dans les autres branches

Lors de l'accès aux sources, c'est la branche par défaut qui est affichée



# Création de branche

---

Plusieurs façons de créer des branches avec Gitlab :

- **A partir d'une issue**, en créant une Merge Request

La branche est créée 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

- A partir du **menu** (*Repository* → *Branches*), Il est possible d'indiquer la branche de départ
- En commande en ligne, en poussant une branche locale vers le dépôt



# Branche protégée

---

Un branche peut être protégée

- Seul un membre avec au moins la permission *Maintainer* peut la créer

***Project → Settings → Protected branches***

- Elle définit des permissions :

- *Allow to Merge* : Qui peut y fusionner une autre branche.  
Par défaut seulement le Maintainer
- *Allow to Push* : Qui peut y faire un push  
Par défaut seulement le Maintainer

- Seul le mainteneur peut supprimer une branche protégée

La branche par défaut est une branche protégée.

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

*\*-stable, production/\**



# Gestion des sources et collaboration

---

Particularités Gitlab  
**MergeRequest**  
Déclinaisons de GitlabFlow



# Introduction

---

Les **Merge Request** sont la base de la collaboration sur Gitlab

Un MR est associée à une branche, elle est généralement associée à une issue

Une MR permet de :

- Empêcher une fusion trop précoce via le statut *Draft*
- Comparer les changements entre 2 branches
- Revoir et discuter des modifications de code
- Visualiser les pipelines associées
- Accéder à l'appli. en fonctionnement (*Review Apps*)
- Faire un suivi du temps
- Effectuer la fusion avec une branche protégée




# Cycle de vie d'un MR

---

- 1) Au démarrage d'un nouveau travail, le développeur crée une *merge request*.  
La collaboration peut commencer et une feature branch est créée.  
La *Merge Request* est préfixée par **Draft**
- 2) Lorsque la fonctionnalité est prête, le développeur le signale en enlevant le statut *Draft*
- 3) Le mainteneur peut alors faire de la revue de code, éventuellement accéder à la *review app*.
- 4) Le mainteneur a ensuite le choix entre :
  - Effectuer la fusion dans la branche protégée, la MR obtient le statut **merged**.  
Si elle est associée à une issue, l'issue est fermée
  - Demander au développeur des améliorations,
  - Abandonner la MR, elle obtient alors le statut **closed**

# Vue projet/groupe

 **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

Open 472Merged 11,188Closed 1,969All 13,629




↺ ▾

Search or filter results...

Last created ▾

**test MR**



!13679 · opened 17 minutes ago by Mike Greiling

   1

updated 14 minutes ago


**Add docs for group issues page and group merge requests page**




!13678 · opened 20 minutes ago by Victor Wu

  0

updated 2 minutes ago


**Docs update links guideline to inline links**




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

   0

updated 34 minutes ago


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




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

   3

updated 15 minutes ago


**Greatly reduce test duration for git\_access\_spec**




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

   2

updated 20 minutes ago


**Implement new system note icons** 0 of 11 tasks completed



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

   1

updated less than a minute ago


**WIP: Prepare 9.5 RC6**




!13672 · opened about 3 hours ago by Jose Ivan Vargas Lopez  Release

  0

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)  Gitaly Pick into Stable

   4

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  Platform

   0

updated about 5 hours ago





# Commentaires et discussions

---

Des **commentaires** peuvent être associés aux MR

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

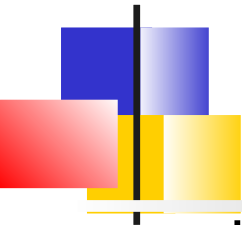
Un commentaire peut être transformé en **discussion/thread**.

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 Rebasing si conflit
  - Rebasing obligatoire
- 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



# Conflits

---

Lorsqu'une MR a des conflits, Gitlab peut proposer de les résoudre via l'UI

- *GitLab* résout les conflits en créant un commit de merge dans la branche source (branche de feature)
- Le commit peut alors être testé avant d'affecter la branche cible.

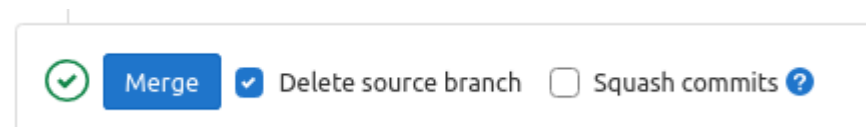


# Bouton Merge

---

Lorsque le mainteneur active le bouton *Merge*, il a généralement le choix :

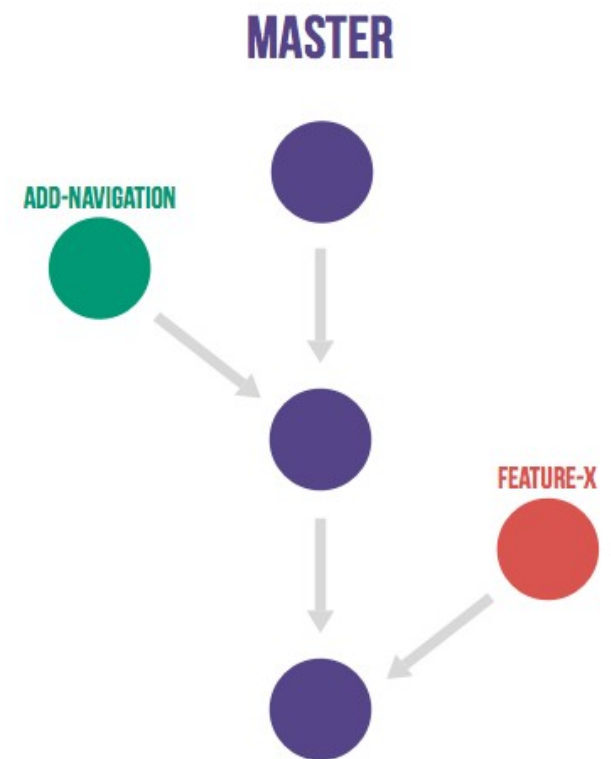
- Pour supprimer automatiquement la branche source
- Pour fusionner (squash) tous les commits en 1 seul



# Gitlab Flow

**Gitlab Flow** désigne le workflow par défaut de gitlab.

Cela consiste à fusionner les branches de features (liées aux issues) dans la branche protégée par défaut via les Merge Request





# Gestion des sources et collaboration

---

Particularités Gitlab  
Gitlabflow et MergeRequest  
**Déclinaisons de GitlabFlow**



# Introduction

---

Il est possible de complexifier le workflow de collaboration autour de Gitlab.

Typiquement en créant d'autres branches protégées qui représentent des branches de déploiement dans des environnements QA ou production ou des branches de releases de version





# Branche de production

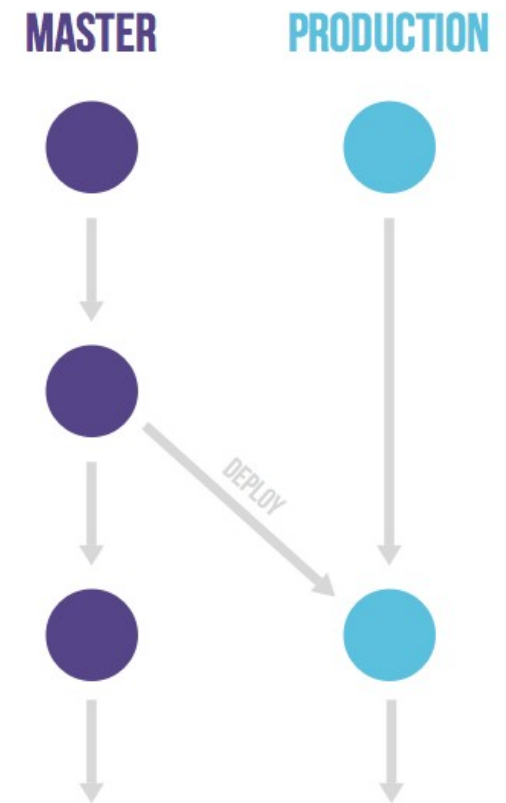
---

Si l'on veut maîtriser les déploiement vers la production, Il est possible d'utiliser une branche ***production*** qui reflète le code qui a réellement été déployé

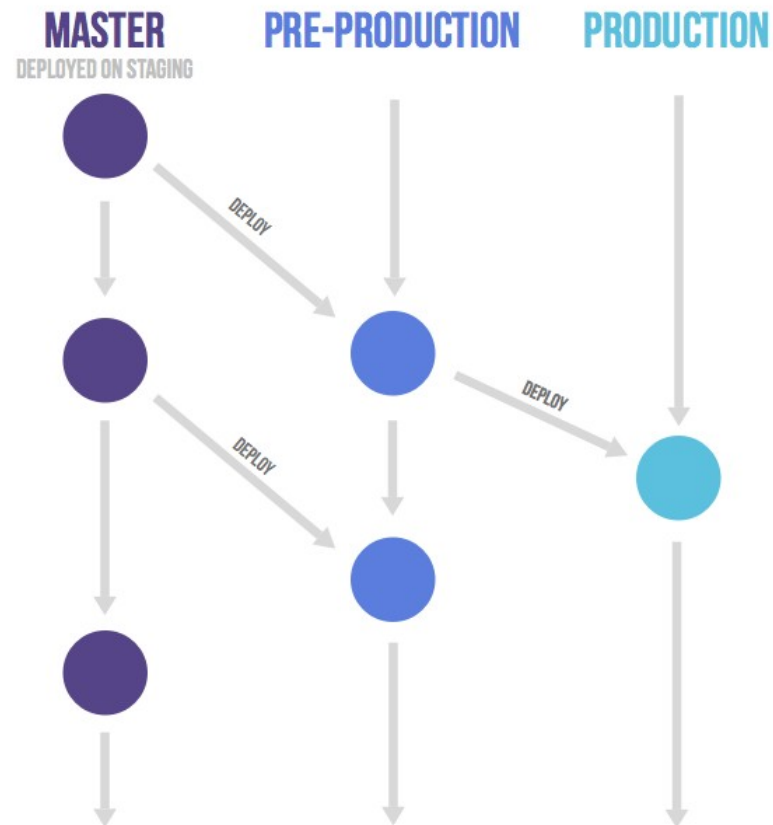
La mise en production consiste alors à fusionner *main* avec la branche de production puis déployer à partir de production.

# main et branche de production

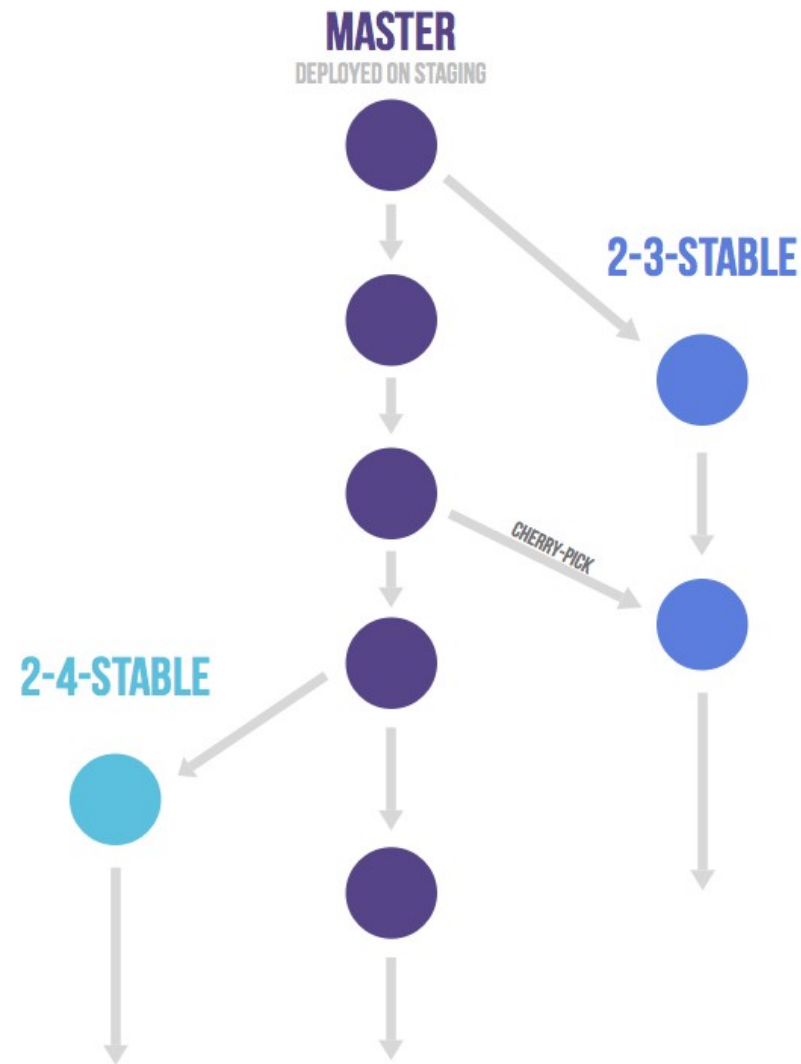
La fusion entre master et production est typiquement gérée par une pipeline



# Déclinaison avec staging



# Branches de releases : Gitflow





# Pipelines CI/CD

---

## **Introduction**

Jobs et Runners

UI Pipelines

Basiques *.gitlab-ci.yml*

Directives disponibles

Environnements et déploiements

Intégration docker



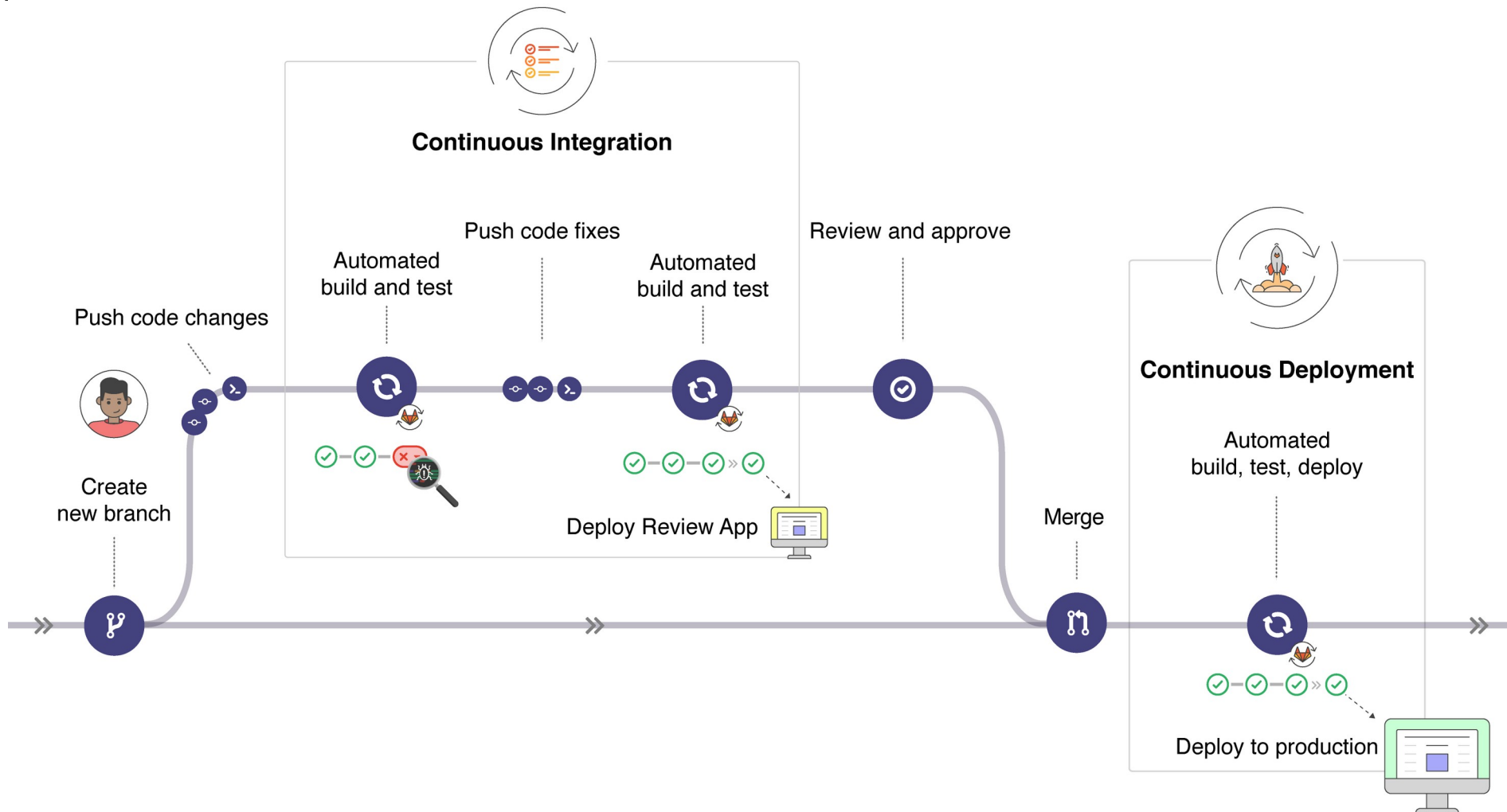
# Introduction

---

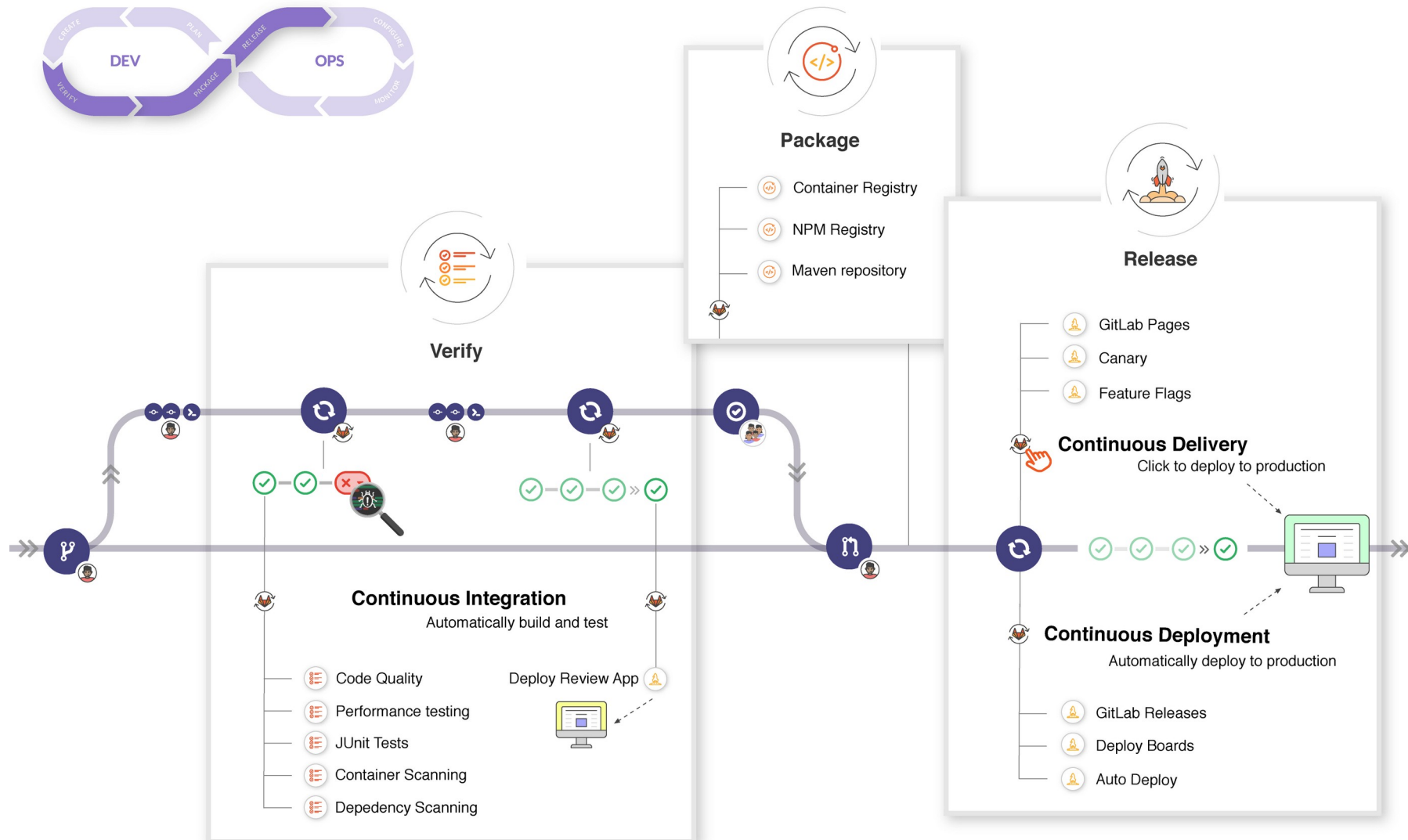
*GitLab CI/CD* est un outil permettant l'intégration, la livraison ou le déploiement continu

- **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 effectue une release ou on déploie en prod  
Les dernières étapes peuvent être manuelles

# CI/CD



# En plus détaillé







# Typologie des tests

---

## Test Unitaire :

*Est-ce qu'une simple classe/méthode fonctionne correctement ?*  
*JUnit, PHPUnit, UnitTest, Karma, Mockito*

## Test d'intégration :

*Est-ce que plusieurs classes/couches fonctionnent ensemble ?*  
*BD/Serveurs embarqués*

## Test fonctionnel :

*Pour l'utilisateur final, est-ce que mon application fonctionne ?*  
*Selenium, HttpUnit, Protractor*

## Test de performance :

*Est-ce que mon application fonctionne bien ?*  
*JMeter, Gatling*

## Test d'acceptation :

*Est-ce que mon client aime mon application ?*  
*Cucumber*



# Classification pipeline DevOps

---

Dans le cadre d'une pipeline DevOps, classification en fonction de :

- Le test nécessite t il un provisionnement d'infrastructure ?
- Durée d'exécution

=> Conditionne la position des tests dans la pipeline

Typiquement :

- Tests unitaires/intégration en premier,
- Test fonctionnel, d'acceptation de performance sur les infrastructures d'intégration, recette, pré-production et de production



# Analyse statique

---

L'analyse statique est l'analyse du code sans l'exécuter.

Les objectifs sont :

- La mise en évidence d'éventuelles erreurs de codage
- La vérification du respect du formatage convenu.
- La vérification des licences utilisées
- Les détections de vulnérabilités



# Pipelines CI/CD

---

Introduction

**Jobs et Runners**

UI Pipelines

Basiques *.gitlab-ci.yml*

Directives disponibles

Environnements et déploiements

Intégration docker



# 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** : Le plus facile à installer mais toutes les dépendances du projet doivent être pré-installées sur le runner (git, npm, jdk, ...)
- **Virtual Machine** : Nécessite Virtual Box ou Parallels. Les outils sont pré-installés sur la VM
- **Docker** : Permet d'exécuter des builds avec une image docker. D'autres services docker peuvent être démarrés 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 crée des pods (machine de build + services)
- **ssh** : Peu recommandé, exécute le build via ssh sur une machine distante



# Configuration runner

---

La configuration d'un runner présente dans  
***/etc/gitlab-runner/config.toml*** déterminent d'autres  
paramètres techniques du runner

*concurrent = 5* // **Nombre de jobs concurents**  
*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]
```



# 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)



# Pipelines CI/CD

---

Introduction

Jobs et Runners

**UI Pipelines**

Basiques *.gitlab-ci.yml*

Directives disponibles

Environnements et déploiements

Intégration docker



# 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

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>

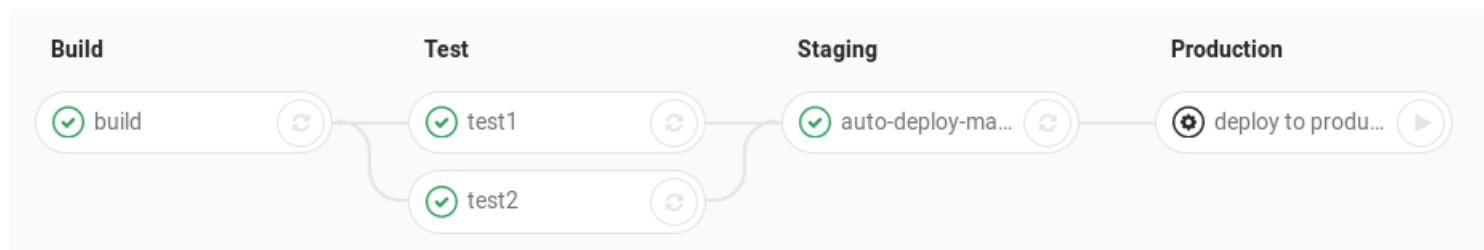
Un tableau de bord permet de voir les dernières exécutions de pipeline, leurs status, le commit, le déclenchement et l'exécution des tâches

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





# Pipelines CI/CD

---

Introduction

Jobs et Runners

UI Pipelines

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

Directives disponibles

Environnements et déploiements

Intégration docker



# 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.

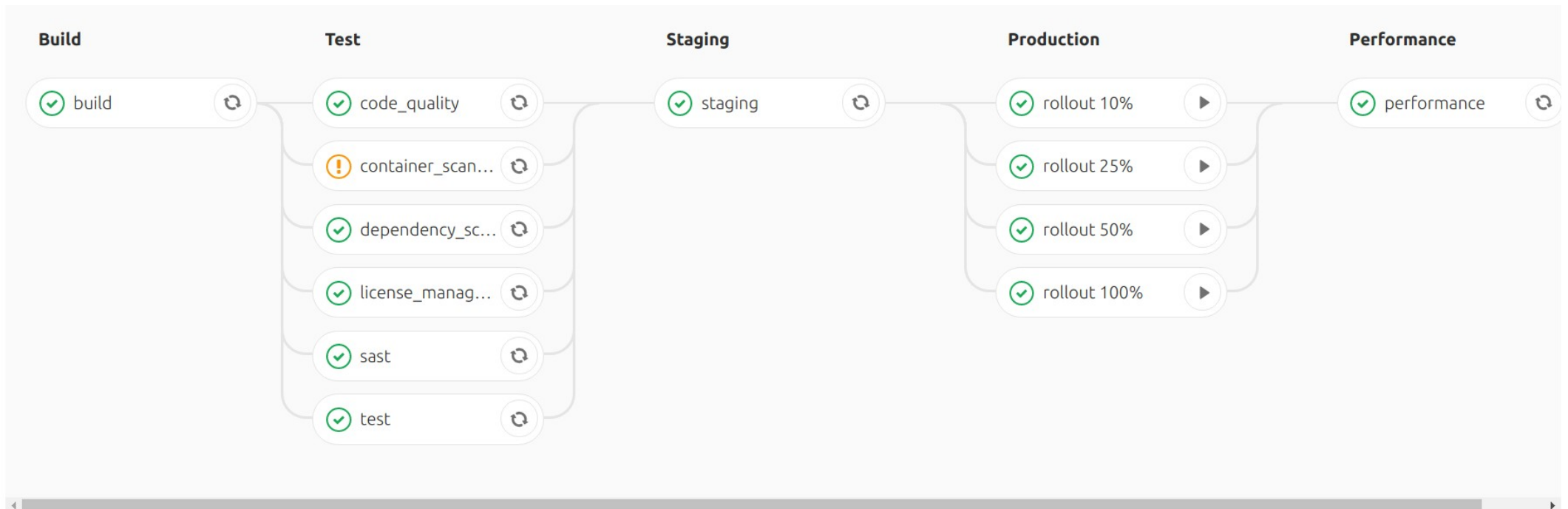
PréRequis :

- 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 sur Review App
  - 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



# Directives de base

---

***script*** : Seul mot-clé nécessaire, décrit les commandes du job

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



# 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



# Contexte des directives

---

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

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, ...**
- **variables** : Variables d'environnement disponibles pour le job
- **stages** : Liste des phases



# 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 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 "$SCI_REGISTRY_USER" -p  
"$SCI_REGISTRY_PASSWORD" $SCI_REGISTRY
```

1. Accessible seulement des branches protégées



# Pipelines CI/CD

---

Introduction

Jobs et Runners

UI Pipelines

Basiques *.gitlab-ci.yml*

**Directives disponibles**

Environnements et déploiements

Intégration docker



# 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:

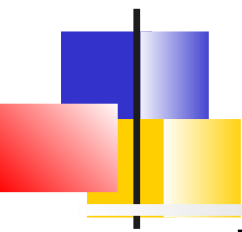
```
script: xelatex mycv.tex
```

```
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*





# ***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
```



# Pipelines CI/CD

---

Introduction

Jobs et Runners

UI Pipelines

Basiques *.gitlab-ci.yml*

Directives disponibles

**Environnements et déploiements**

Intégration docker



# 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

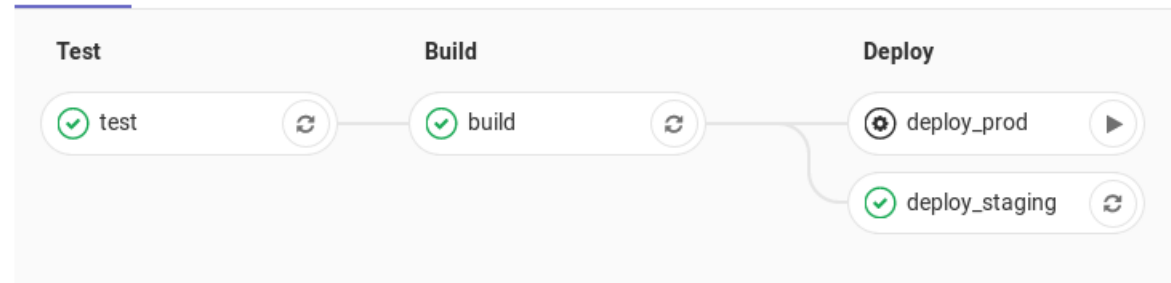
L'ajout de *when: manual* convertit le job en un job manuel et expose un bouton Play dans l'UI

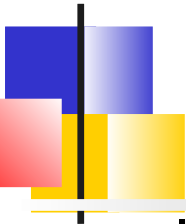
## Use busybox

🕒 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

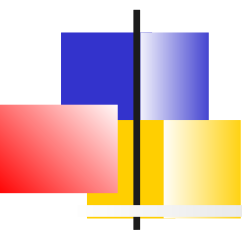


# Example

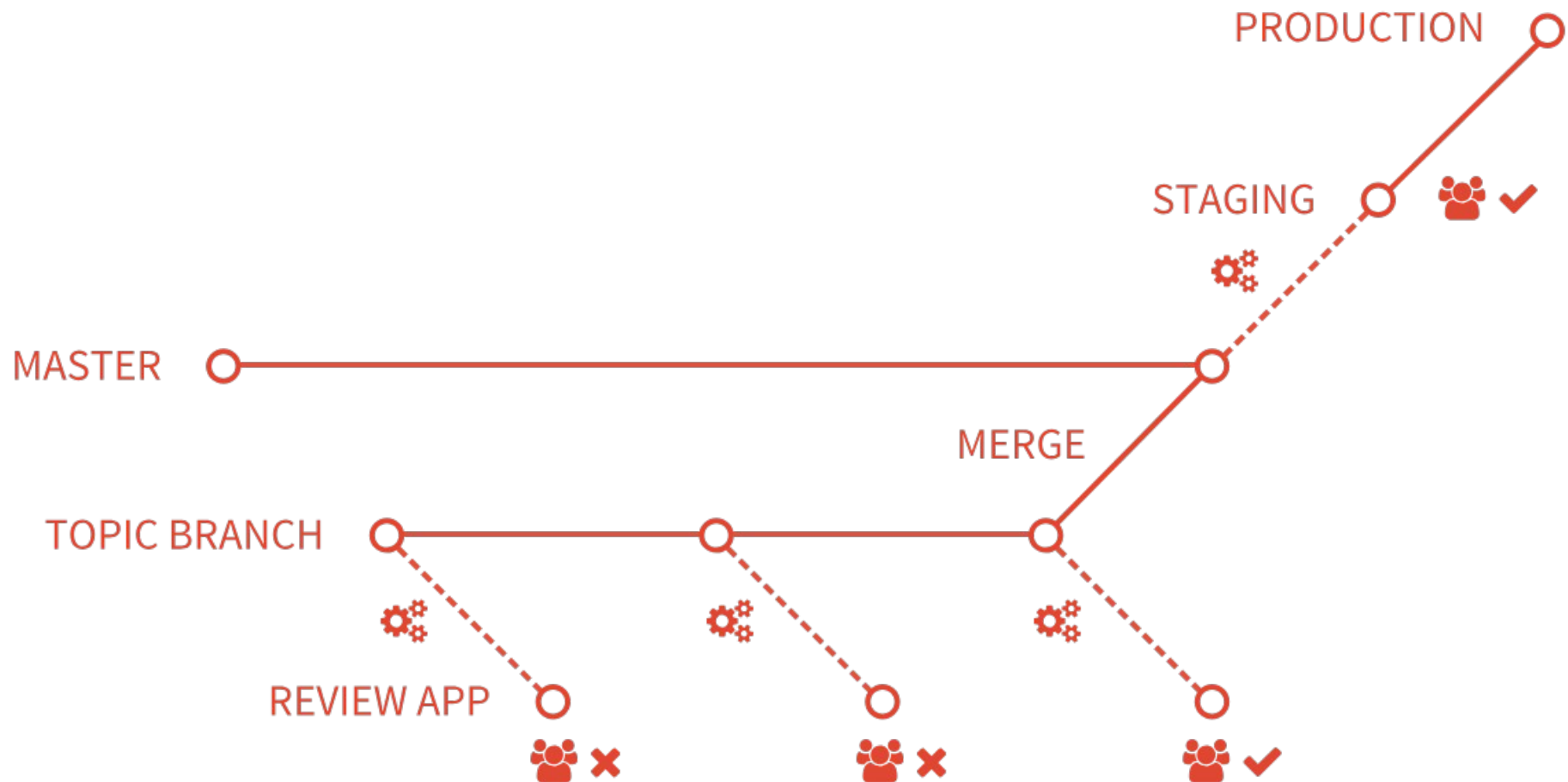
---

```
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
```



# Pipelines CI/CD

---

Introduction  
Jobs et Runners  
UI Pipelines  
Basiques *.gitlab-ci.yml*  
Directives disponibles  
Environnements et déploiements  
**Intégration docker**



# 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 (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 de 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

=> 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  
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)

---

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 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:lates
```



# Phases des pipelines

---

## **Construction et tests développeur**

Analyses statiques

Dépôts d'artefacts

Déploiement et release

Tests post-déploiements

Gestion de l'infrastructure



# Construction

---

Les premières tâches d'une pipeline consiste généralement

- A exécuter les tests unitaires développeurs
- Si ceux-ci réussissent, packager le code source

En fonction des piles technologiques, le packaging du code source diffère

- Java : Compilation + création d'un jar
- Javascript : Minification et obfuscation du code + création d'un zip
- ...

Gitlab s'appuie alors sur des outils de la pile technologique qui sont soit préinstallés sur des runners ou dans des images docker



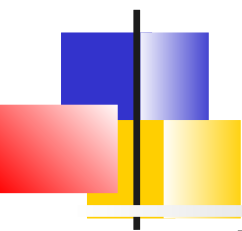
# Outils de build

---

La phase de build s'appuie typiquement sur un outil de build propre à la techno :

- **Maven** (Java): Le plus répandu et le plus supporté.
- **Gradle** : « *Build As Code* ».  
S'applique à d'autres langages que Java (C, C+, Python, Php, ...)
- **npm, yarn, webpack, ...** : Monde JavaScript
- **tslint** : Linter typescript
- **Composer, PHPUnit** : PHP
- **pip, unittest** : Python





# Support Gitlab pour les tests

---

Pour ces tests de début de pipeline,  
Gitlab permet :

- De publier les résultats des tests et de les attacher à une MR
- Si *Ruby*, de définir des *FailFast Test* permettant d'économiser des ressources runners



# Publier les résultats de test

---

Gitlab ne supporte que le format JUnit

Il suffit de placer la directive

***artifacts:reports:junit*** 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
```

# Publication résultat de test

*Gitlab* propose alors des rapports de tests qui permettent de facilement isoler les tests ayant échoués

Pipeline Needs Jobs 1 Failed Jobs 1 Tests 3

**Summary**

3 tests	2 failures	0 errors	33.33% success rate	16.00ms
---------	------------	----------	---------------------	---------

**Jobs**

Job	Duration	Failed	Errors	Skipped	Passed	Total
jest	16.00ms	2	0	0	1	3

! Test summary contained 3 failed out of 3 total tests

! rspec found 1 failed out of 1 total test

✖ Failed 1 time in master in the last 14 days User#full\_name returns first\_name + last\_name

! jest found 2 failed out of 2 total tests

✖ New Calculator #add returns the sum of the 2 given numbers

✖ Failed 1 time in master in the last 14 days Calculator #subtract returns the difference between the 2 given numbers

[View full report](#)[Collapse](#)



# Couverture des tests

---

Si l'on s'est équipé d'un outil calculant la couverture des tests, il est possible de publier ces métriques dans Gitlab.

Les résultats sont visibles :

- Dans les Merge request
- Dans les analytiques projets
- Dans les analytiques groupe
- Sous forme de badge au niveau d'un dépôt

Il est possible de conditionner une MR à un certain seuil de couverture *Premium*



# Mise en place

---

La mise en place consiste à utiliser le mot-clé **coverage** dans *.gitlab-ci.yml* et d'indiquer une expression régulière permettant d'extraire l'information de la sortie standard.

Exemple *JacoCo (Java/Kotlin)*

```
/Total.*?([0-9]{1,3})%/.
```

# Publication

⚠ Pipeline #4637061 passed with warnings for 99f87a83. [View details](#)

Coverage 91.98%

Accept Merge Request

The source branch will be removed.

[✎ Modify commit message](#)

✅ Post Test

✅ passed

#5199993

coverage

🕒 01:20

📅 about 2 hours ago

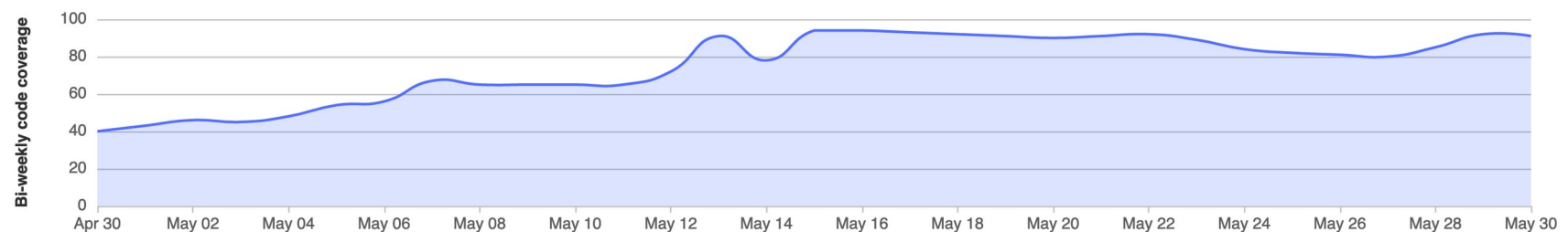
91.98%



## Code coverage statistics for master Mar 12 - Jun 10

[Download raw data \(.csv\)](#)

rspec ▾



— rspec Avg: 74.9 · Max: 94



# Phases des pipelines

---

Construction et tests développeur

**Analyses statiques**

Dépôts d'artefacts

Déploiement et release

Tests post-déploiements

Gestion de l'infrastructure



# Introduction

---

*Gitlab* fournit également du support pour les analyses statiques de code source

- Analyse qualité
- Utilisation des licences
- Détection de vulnérabilités

La mise en place s'effectue dans *.gitlab-ci.yml* et les résultats sont publiés dans le projet





# Analyse qualité

---

*Gitlab* s'appuie sur l'outil **Code Climate**<sup>1</sup> et ses plugins pour analyser le code source.

Les résultats sont disponibles<sup>2</sup> :

- Dans une Merge request
- Dans le détail d'une pipeline
- Dans la vue qualité d'un projet

Ils peuvent également être téléchargés au format brut

1. Supporte : Ruby, Python, PHP, JavaScript, Java, TypeScript, GoLang, Swift, Scala, Kotlin, C#

2. Dépend fortement de la licence



# Mise en place

---


La mise en place nécessite des pré-requis :



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

Pour autoriser l'analyse qualité :

- Utiliser *AutoDevOps*
- Inclure le gabarit de qualité dans *.gitlab-ci.yml*
  - include:
    - template: Code-Quality.gitlab-ci.yml

# Affichage Merge Request



 Code quality degraded on 7746 points 

Collapse

- ◆ Critical - Consider simplifying this complex logical expression.  
in [scripts/frontend/stylelint/stylelint-utils.js:15](#)
- ◆ Critical - Consider simplifying this complex logical expression.  
in [app/assets/javascripts/projects/settings/access\\_dropdown.js:248](#)
- ◆ Critical - CSRF vulnerability in OmniAuth's request phase  
in [Gemfile.lock:810](#)
- ▼ Major - Function `buildConfig` has 7 return statements (exceeds 4 allowed).  
in [workhorse/main.go:67](#)
- ▼ Major - Function `run` has 8 return statements (exceeds 4 allowed).  
in [workhorse/main.go:152](#)
- ▼ Major - Method `Resizer.Inject` has 5 return statements (exceeds 4 allowed).  
in [workhorse/internal/imageresizer/image\\_resizer.go:164](#)
- ▼ Major - Method `Resizer.tryResizeImage` has 7 return statements (exceeds 4 allowed).  
in [workhorse/internal/imageresizer/image\\_resizer.go:268](#)
- ▼ Major - Function `unpackFileFromZip` has 7 return statements (exceeds 4 allowed).  
in [workhorse/internal/artifacts/entry.go:64](#)



# Sécurité

GitLab analyse la sécurité d'une application, soit dans le cadre d'une pipeline, soit de façon planifiée.

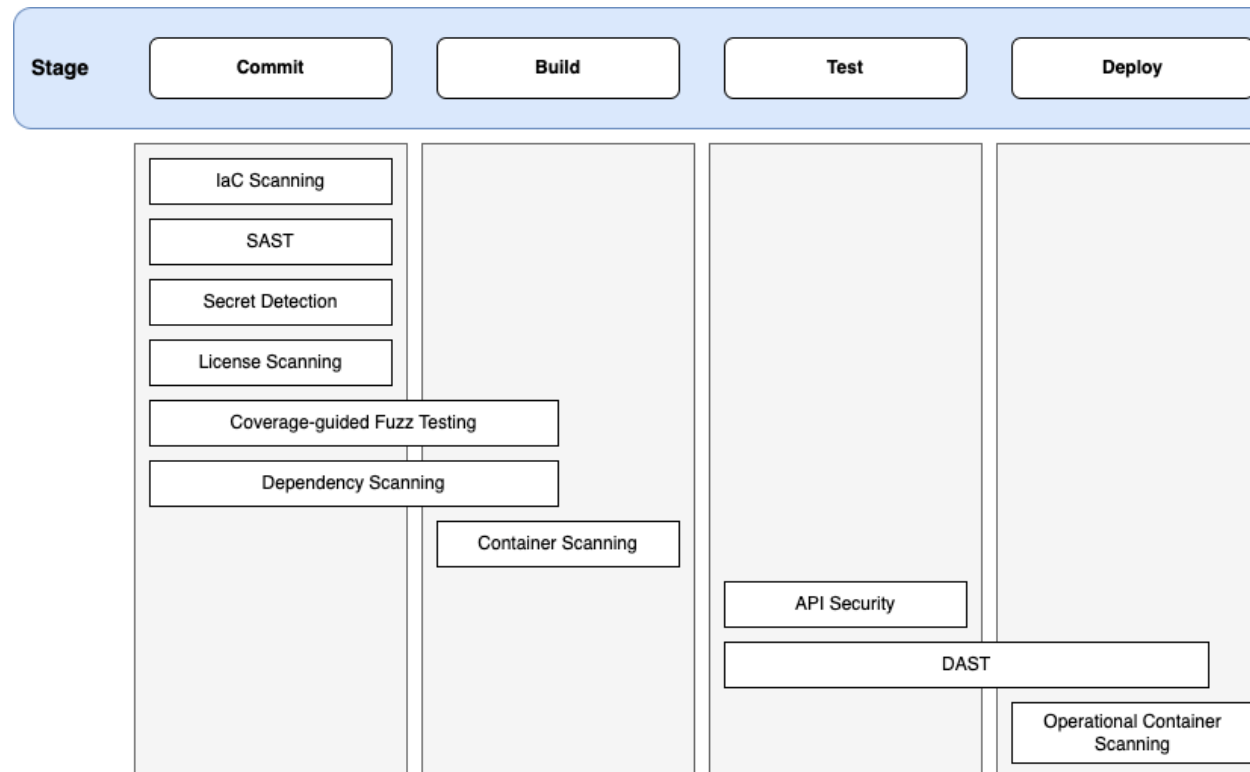
Cela couvre :

- Le **code source**  
*Static Application Security Testing (SAST) + Analyze du dépôt pour la détection de secrets*
- Les **dépendances** (Librairies, conteneurs)  
*Dependencies Scanning, Container Scanning*
- Les **vulnérabilités** dans une application Web en cours d'exécution.  
*Dynamic Application Security Testing (DAST), Analyse des APIs pour détection de vulnérabilités connus ou inconnues (API fuzzing)*
- **L'infrastructure** : Configuration de l'IAC (Infrastructure As Code)



# Étapes vs Outils

Chaque outil intervient à différentes étapes de la pipeline





# Mise en place

---

La mise en place de ces outils dépend fortement de la licence et certains outils ne sont disponibles qu'en version payante.

Les résultats des outils peuvent être publiés dans l'interface si une directive ***artifacts:reports <mot-clé>*** est présente dans *.gitlab-ci.yml*



# Licenses

---

*Gitlab* permet également de fixer des politiques transverses quant à l'utilisation de produit tiers.

L'analyse de code permet de détecter les licences associées aux dépendances et d'afficher des rapports si le projet utilise des dépendances non-permises.



# Outils Free

---

## **SAST : Static Application Security Testing**

```
stages:  
- test  
sast:  
  stage: test  
include:  
- template: Security/SAST.gitlab-ci.yml
```

## **Secret Detection**

```
include:  
- template: Security/Secret-Detection.gitlab-ci.yml
```

## **Infrastructure as Code (IaC) Scanning**

```
include:  
- template: Security/Secret-Detection.gitlab-ci.yml
```

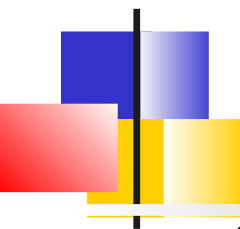




# Phases des pipelines

---

Construction et tests développeur  
Analyses statiques  
**Dépôts d'artefacts**  
Déploiement et release  
Tests post-déploiements  
Gestion de l'infrastructure



# 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 en fait un projet Gitlab sans dépôt de source associé.

Les packages publiés héritent de la visibilité du projets

Pour y publier, les autres projets doivent s'authentifier auprès du registre via des jetons (personnel ou job) et configurer leur outil de build afin de publier vers la bonne URL



# 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



# Exemple : Construction puis push vers un registre

---

```
stage: build
before_script:
  - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" $CI_REGISTRY
# Default branch leaves tag empty (= latest tag)
# Other branches are tagged with the escaped branch name (commit ref slug)
script:
  - |
    if [[ "$CI_COMMIT_BRANCH" == "$CI_DEFAULT_BRANCH" ]]; then
      tag=""
      echo "Running on default branch '$CI_DEFAULT_BRANCH': tag = 'latest'"
    else
      tag=":$CI_COMMIT_REF_SLUG"
      echo "Running on branch '$CI_COMMIT_BRANCH': tag = $tag"
    fi
  - docker build --pull -t "$CI_REGISTRY_IMAGE${tag}" .
  - docker push "$CI_REGISTRY_IMAGE${tag}"
```



# *Terraform Module Registry*

---

Avec ***Terraform Module Registry***, les projets GitLab peuvent devenir des registres privés pour les modules *terraform*<sup>1</sup>.

Les modules peuvent être publiés par des jobs CI/CD puis consommés par d'autre projet

Les autres projets s'authentifient auprès du registre par des jetons

*1. Les modules Terraform permettent la réutilisation d'infrastructure entre projets*



# Publication de modules

---

Plusieurs façons pour publier un module :

- Utiliser directement l'API

PUT

/projects/:id/packages/terraform/modules/:module-name/:module-system/:module-version/file

- Dans une pipeline CI/CD utiliser le gabarit ***Terraform-Module.gitlab-ci.yml***
- Dans une pipeline CI/CD, effectuer un appel d'API





# Phases des pipelines

---

Construction et tests développeur

Analyses statiques

Dépôts d'artefacts

**Déploiement et release**

Tests post-déploiements

Gestion de l'infrastructure



# Introduction

---

Les pipelines CI/CD déploient généralement 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 à un instant t
- Si on dispose d'un service de déploiement comme Kubernetes, cela offre d'autres possibilités



# Types d'environnement

---

Un environnement peut être :

– **Statique** :

- A un nom fixe comme « staging », « production ».
- Réutilisé par des déploiements successifs.
- Créé manuellement ou via une pipeline

– **Dynamique** :

- A un nom dynamique basé sur une variable de CI/CD..
- Typiquement associé à un branche de features. Permet de valider une fonctionnalité en live (Review apps).
- Arrêté puis supprimé à la suppression de la branche
- Créé par une pipeline



# 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`



# Exemple

---

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



# Déploiement manuel

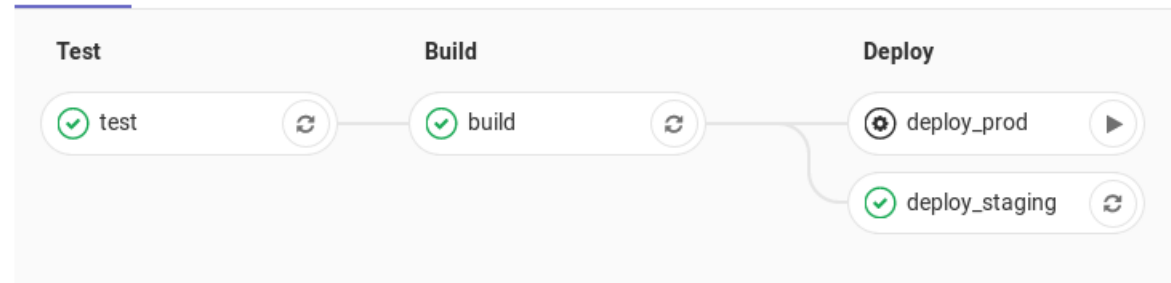
L'ajout de *when: manual* convertit le job en un job manuel et expose un bouton Play dans l'UI

## Use busybox

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

🔑 [ec75f5bf](#) ... 📄

Pipeline Jobs 4





# Environnements dynamiques

Dans le cas d'**environnements dynamiques**, les clés *name* et *url* peuvent 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*

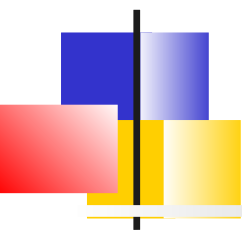


# Example

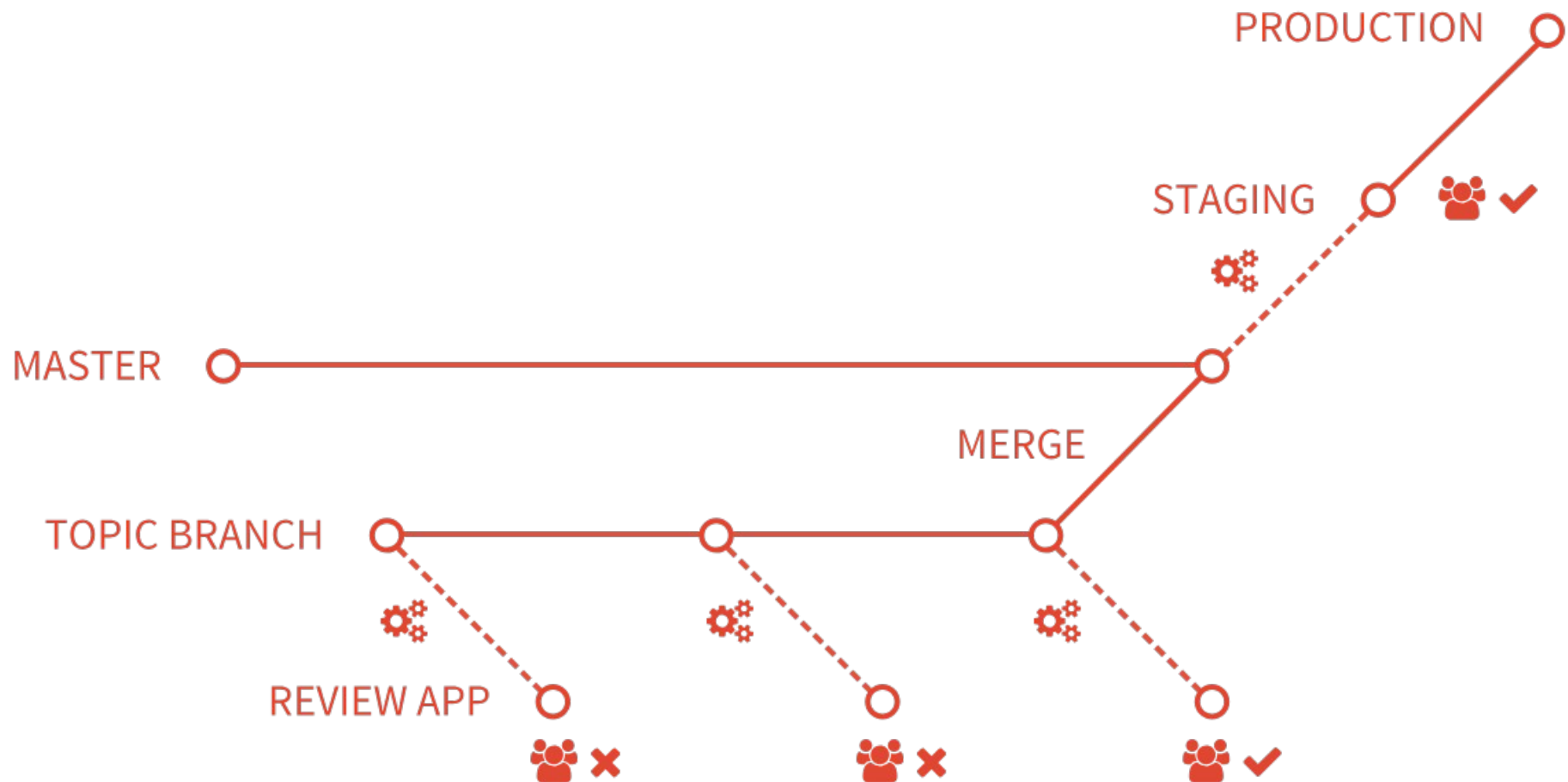
---

```
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:
    - main
```





# 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.



# Exemple

---

```
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
```



# 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 relase 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.
```



# Feature flags

---

Les **indicateurs de fonctionnalité** s'appuient sur des déploiements par petits lots.

Les fonctionnalités peuvent alors être activées ou désactivées pour des sous-ensembles d'utilisateurs.

Cette fonctionnalité s'appuie sur le projet OpenSource

[\*https://github.com/Unleash/unleash\*](https://github.com/Unleash/unleash)

Cela nécessite que l'application intègre l'API *unleash*



# Phases des pipelines

---

Construction et tests développeur

Analyses statiques

Dépôts d'artefacts

Déploiement et release

**Tests post-déploiements**

Gestion de l'infrastructure



# Introduction

---

Après un déploiement, des tests de post-déploiement peuvent être inclus dans la pipeline.

Ces tests sont métiers et peuvent être faits avec tout type d'outil.

De son côté, Gitlab propose d'intégrer des outils de surveillance dans son interface.



# Monitoring gitlab

---

De nombreuses fonctionnalités Gitlab concernant le monitoring des applications sont en cours de dépréciation.

Reste :

- La gestion des alertes et des incidents
- La traque des erreurs



# Gestion des incidents

---

La gestion des incidents nécessite :

- L'intégration dans Gitlab des outils de monitoring.
- La gestion des astreintes dans le système de notifications des alertes.
- La classification des Alertes et Incidents.
- Informer les utilisateurs avec une page de Status.



# Intégration

---

Alertes : Gitlab peut recevoir des alertes via des webhooks

Le menu **Settings > Monitor** permet d'activer les endpoints et de configurer les champs de l'alerte

Le menu **Monitor > Alerts** permet de les visualiser

Incidents : Les incidents sont créés manuellement dans l'interface

Gestion des astreintes **Premium**

Page de statut **Ultimate +** Compte AWS

# Status page



[Contact support](#)

## Incidents

**April 1, 2020**

Testing a new incident

Opened

[Full report](#)

**March 27, 2020**

2020-03-24: Last WALE backup was seen 20m 10s ago

Opened

[Full report](#)

**March 27, 2020**

2020-03-23: error burn-rate exceeding SLO across several services in CNY

Opened

[Full report](#)

**March 27, 2020**

2020-03-19: GitLab Appears to be throwing many 500s

Closed

[Full report](#)





# Phases des pipelines

---

Construction et tests développeur  
Analyses statiques  
Dépôts d'artefacts  
Déploiement et release  
Tests post-déploiements  
**Gestion de l'infrastructure**



# Introduction

---

La gestion de l'infrastructure avec Gitlab s'appuie sur ***Terraform***.

On peut alors définir des ressources versionnées, réutilisées et partagées :

- Composants de bas niveau : CPU/mémoire, stockage, réseau
- Haut-niveau : entrées DNS, fonctionnalités SaaS
- Adopter les déploiements ***GitOps***
- Utilisez GitLab comme stockage d'état Terraform.
- Stockez les modules Terraform



# Intégration Terraform

---

L'intégration s'effectue via Gitlab CI

Un gabarit est fourni qui :

- Utilise ***GitLab-managed Terraform state*** pour stocker les états Terraform
- Déclenche 5 phases de pipeline : ***init, test, validate, build, deploy***
- Exécute les commandes Terraform ***test, validate, plan***, et ***plan-json*** ainsi que ***apply*** dans la branche par défaut
- Effectue un ***laC scanning*** pour vérifier la sécurité



# Exemple *.gitlab-ci.yml*

---

include:

- # To fetch the latest template, use:
  - template: Terraform.latest.gitlab-ci.yml
- # To fetch the advanced latest template, use:
  - template: Terraform/Base.latest.gitlab-ci.yml
- # To fetch the stable template, use:
  - template: Terraform.gitlab-ci.yml
- # To fetch the advanced stable template, use:
  - template: Terraform/Base.gitlab-ci.yml

variables:

- TF\_STATE\_NAME: default
- TF\_CACHE\_KEY: default
- # If your terraform files are in a subdirectory, set TF\_ROOT accordingly. For example:
- # TF\_ROOT: terraform/production



# Kubernetes

---

*GitLab* peut se connecter à un cluster Kubernetes pour déployer, gérer et surveiller les applications

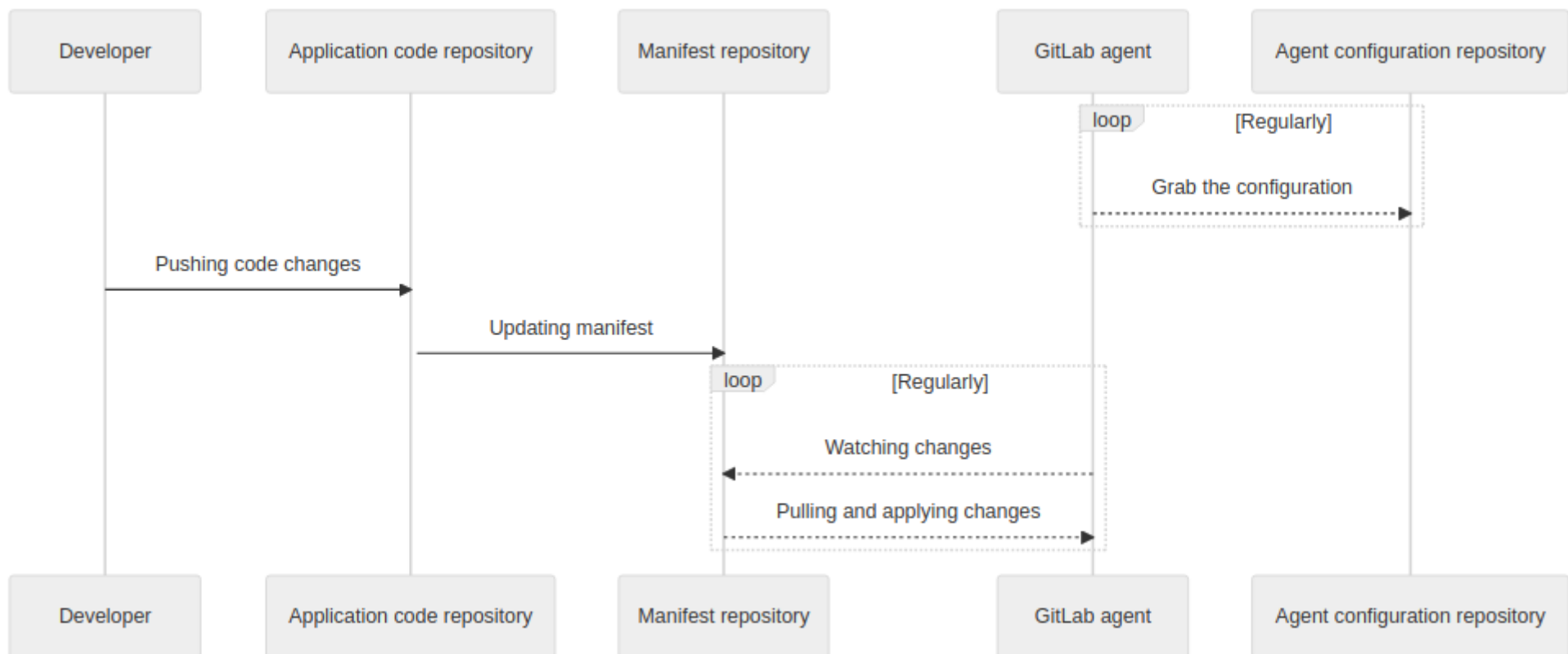
L'interface permet de créer des clusters vers 3 fournisseurs : Google GKE, Amazon EKS, Civo

La connexion s'effectue en installant un agent dans le cluster

2 workflows sont alors proposés :

- GitOps
- Gitlab CI/CD

# GitOps





# Gitlab CI/CD

---

Dans ce cas, les commandes de déploiement Kubernetes sont dans la pipeline.

La mise en place consiste à

- Enregistrer l'agent dans le projet
- Utiliser des commandes *kubectl* dans *.gitlab-ci.yml*



# Exemple

---

```
deploy:
  image:
    name: bitnami/kubectl:latest
    entrypoint: ['']
  script:
    - kubectl config get-contexts
    - kubectl config use-context path/to/agent/repository:agent-name
    - kubectl apply -f myManifest.yml
```





# Annexes

---

Rappels Git  
Auto DevOps



# 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

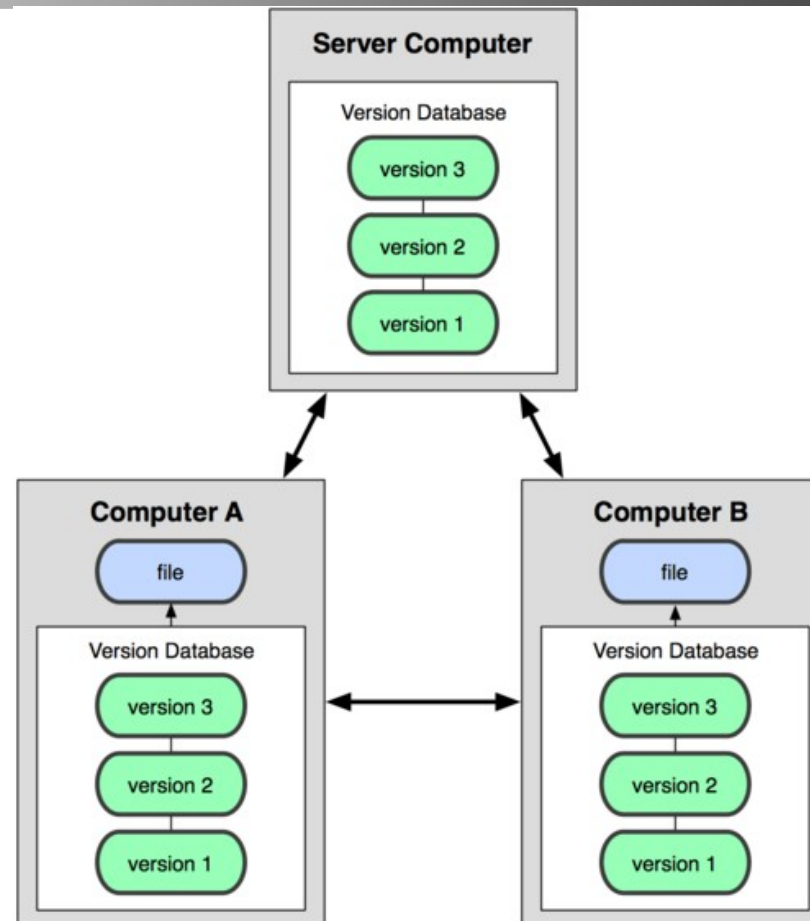
---

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

---

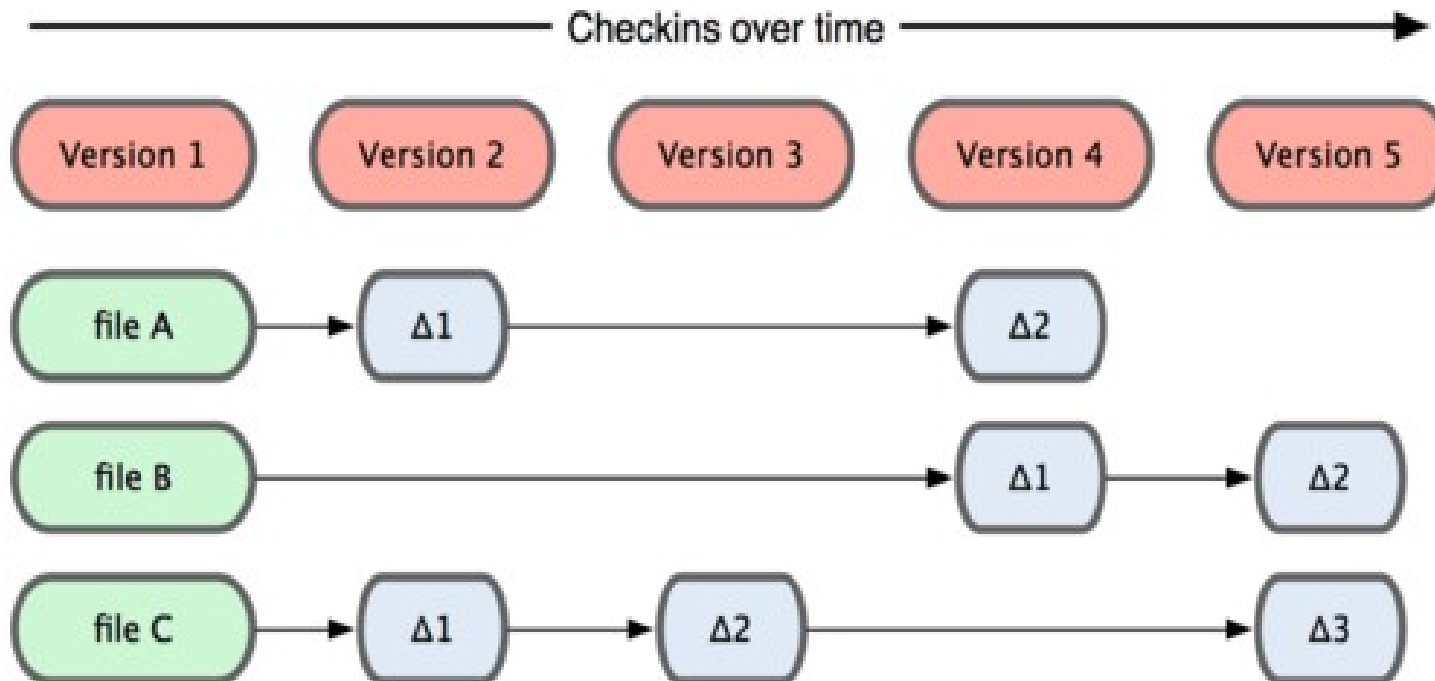
*Git* adopte une approche radicalement différente pour le stockage des données par rapport aux systèmes traditionnels comme Subversion

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

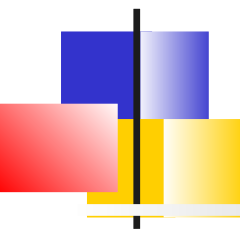
- A chaque commit, *Git* prend un instantané de l'état des fichiers et le stocke dans sa base.
- Pour être efficace, si un fichier est inchangé, son contenu n'est pas stocké une nouvelle fois mais plutôt une référence au contenu précédent

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

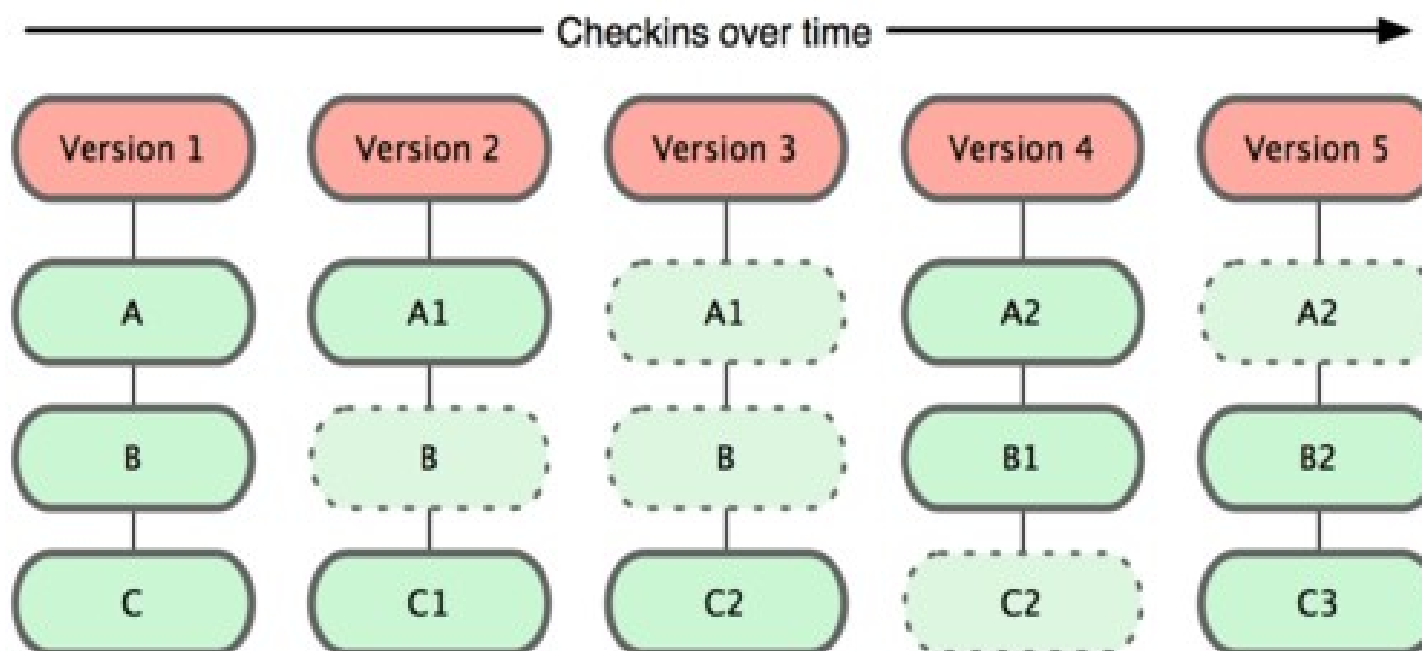
# Approche standard (CVS, SVN, ...)







# 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

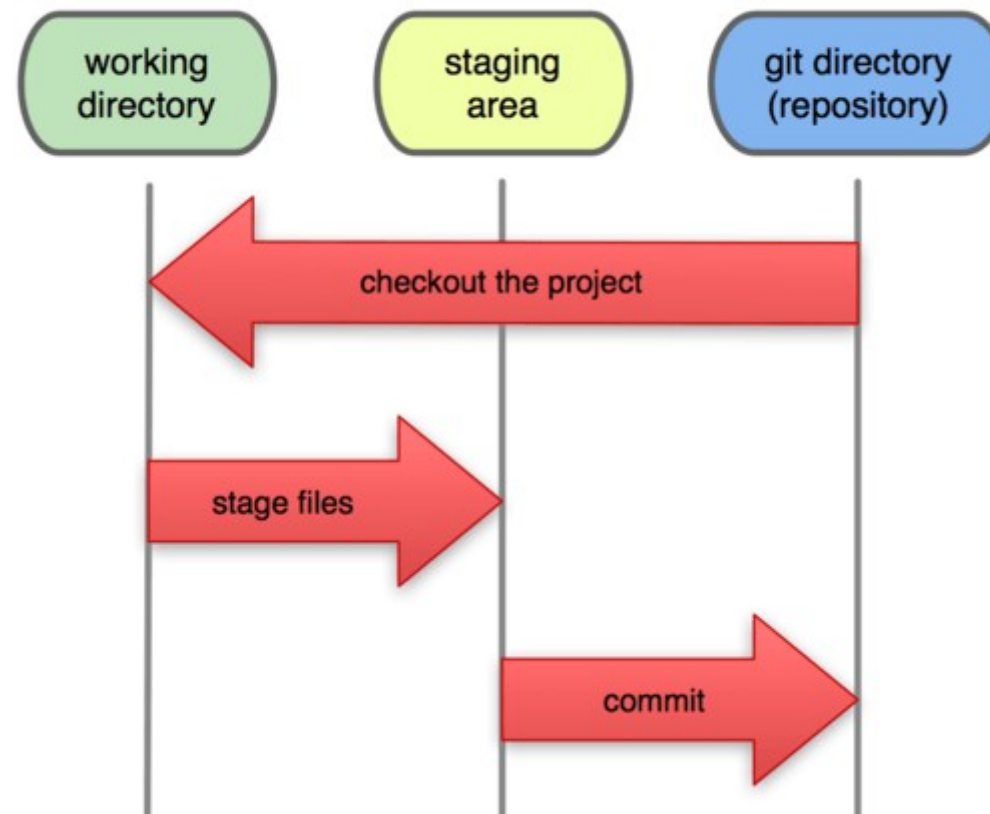
---

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

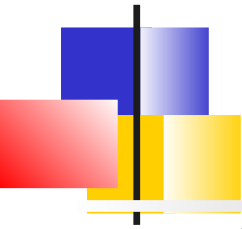
---

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

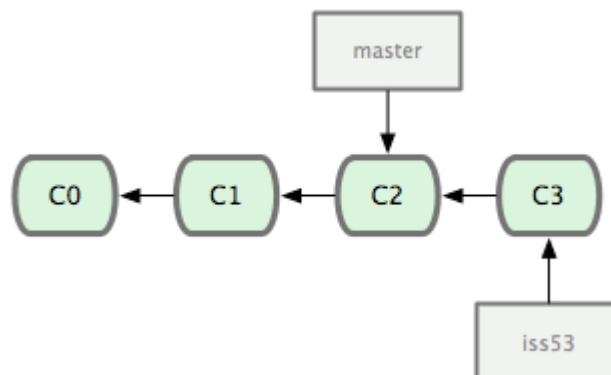
# Création branche locale

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

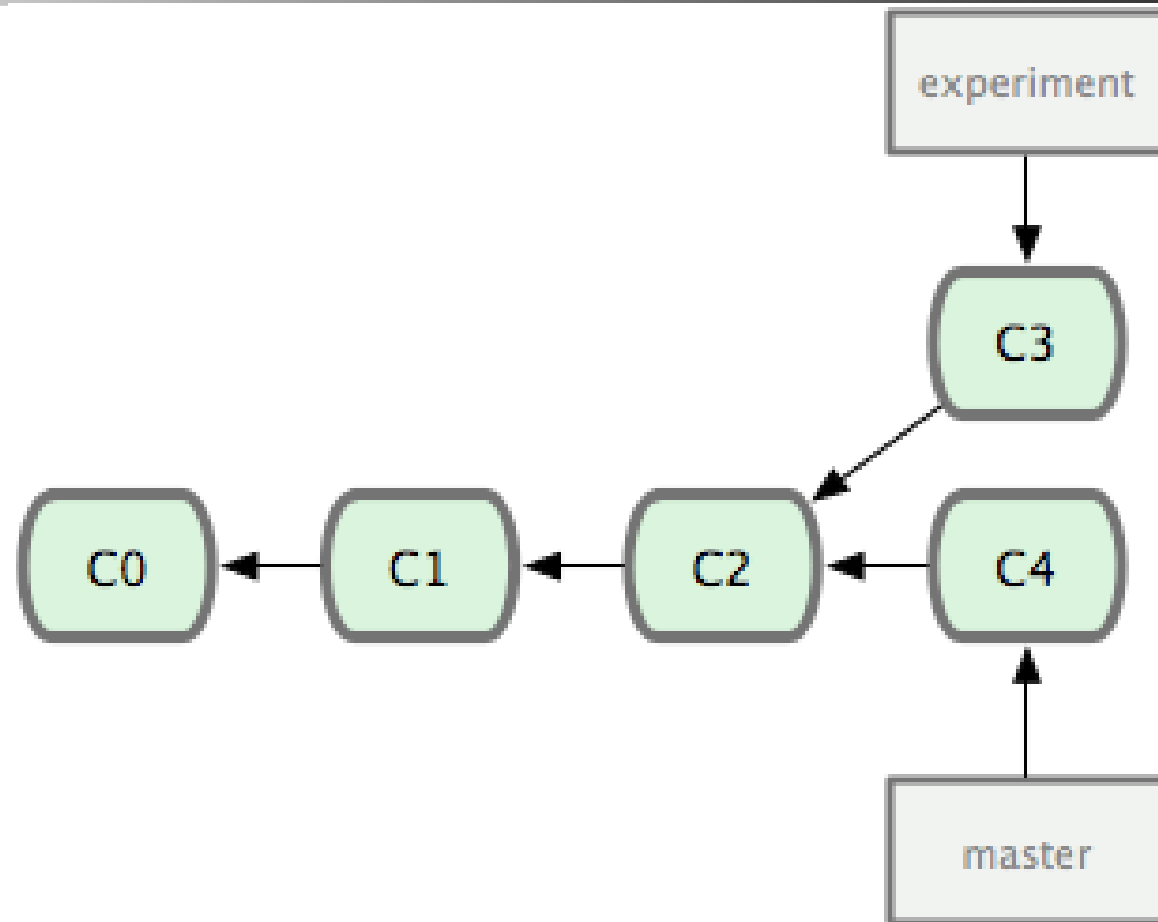
```
$ git checkout -b iss53
```

Switched to a new branch 'iss53'

Après quelques commits :

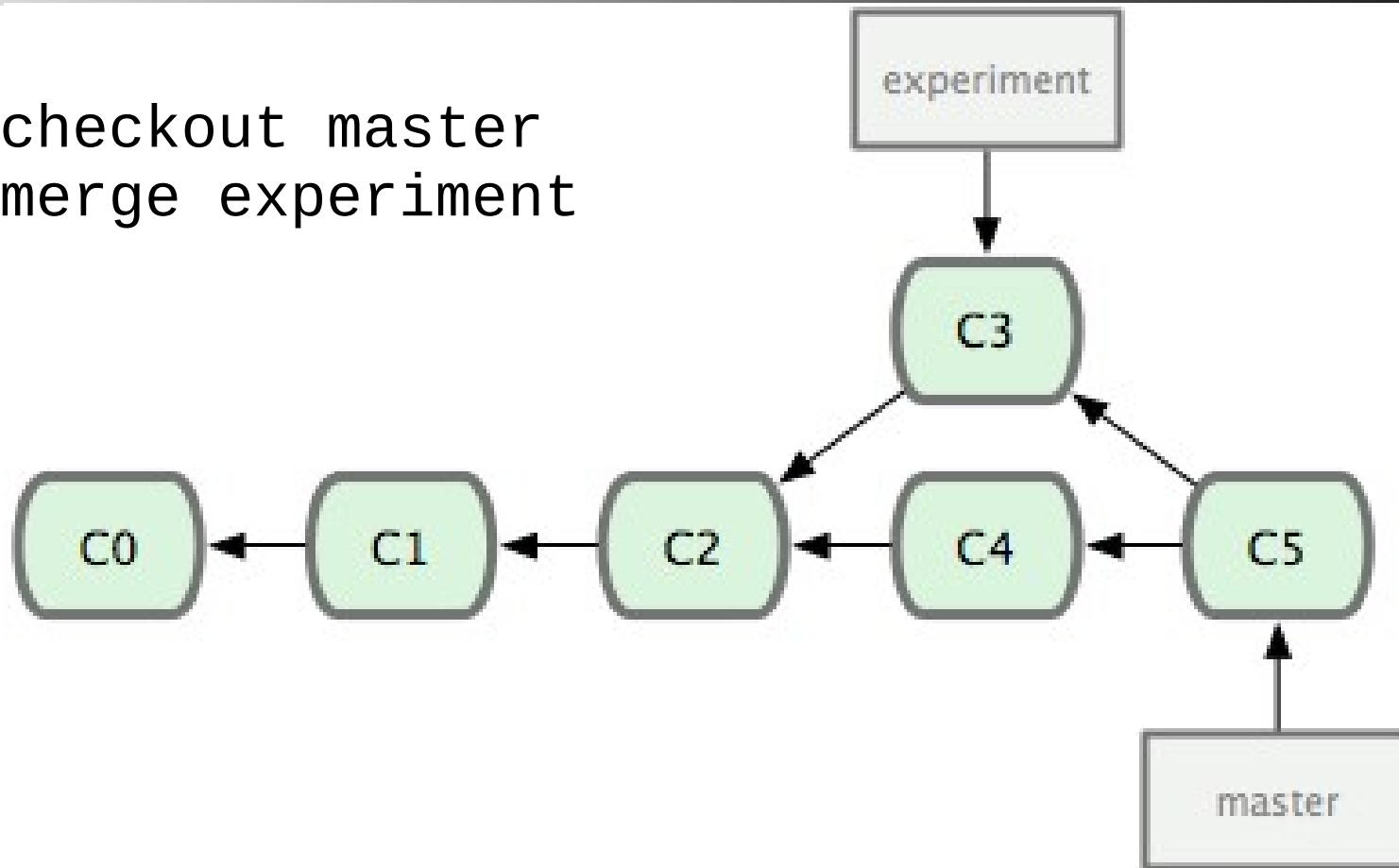


# 2 branches divergentes



# Résultat d'un merge

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





# Merge et conflit

---

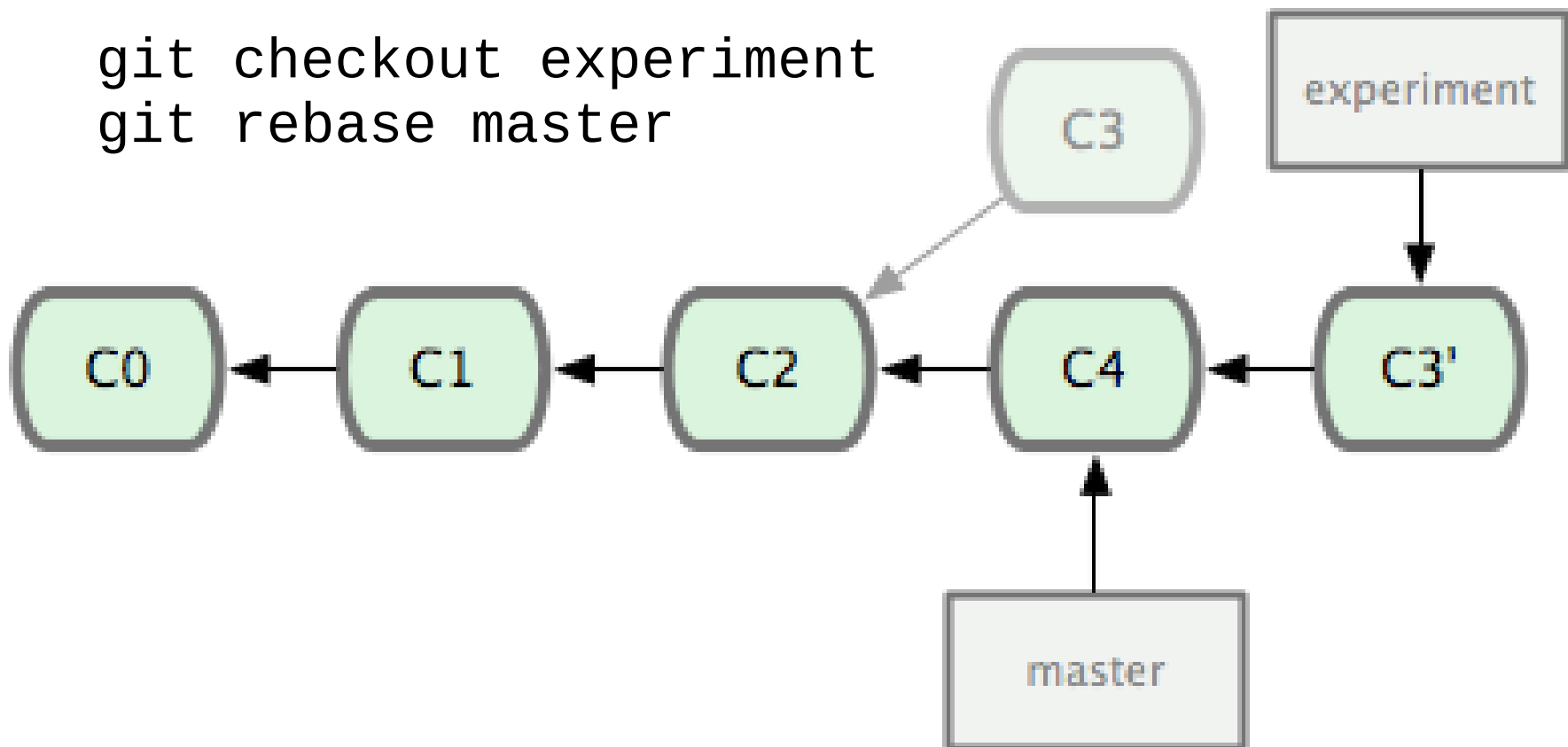
Si des conflits apparaissent lors de la fusion, l'opération s'interrompt

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.

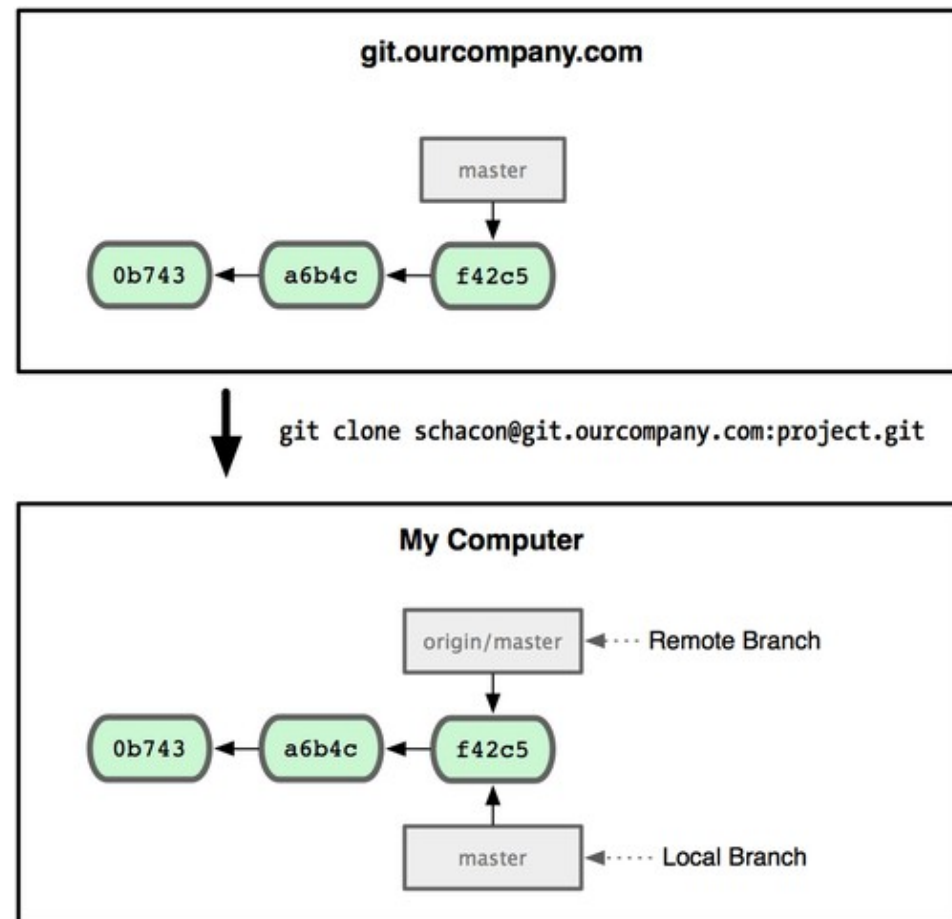
Ce sont des branches locales que l'on ne peut pas modifier.

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

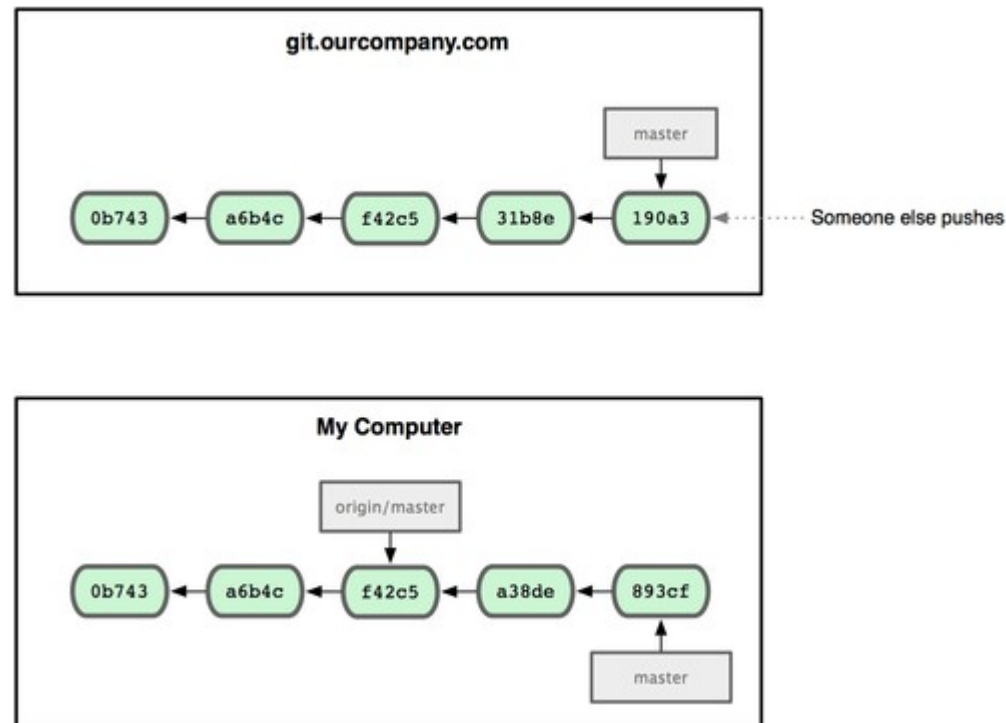
Elles sont référencées dans les commandes *Git* par **(remote)/(branch)**

# Exemple après clone



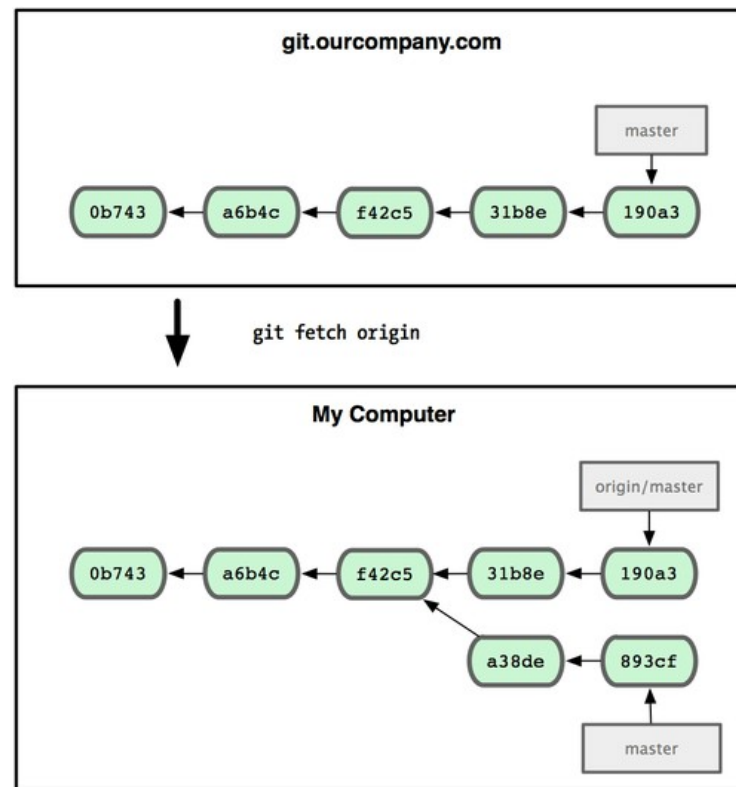
# Déplacement

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

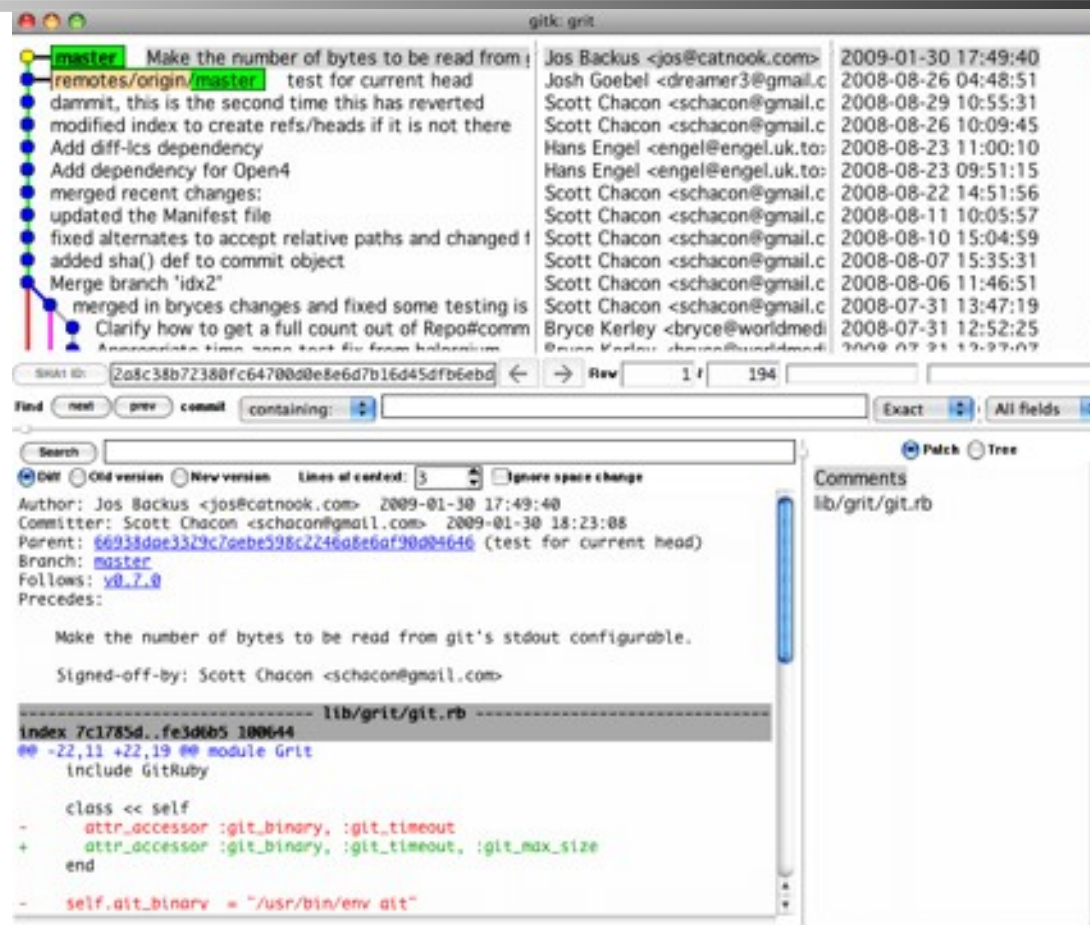


# Synchronisation

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*



The screenshot shows the gitk graphical user interface. The top window displays a commit history with a list of commits on the left and a detailed view of a specific commit on the right. The commit being viewed is 2a8c38b72380fc64700d0e8e6d7b16d45dfb6ebd, authored by Scott Chacon on 2009-01-30. The commit message is "test for current head". The diff view shows changes to the file lib/grit/git.rb, including the addition of a new class method attr\_accessor for :git\_timeout and :git\_max\_size.

```
gitk: gitk
master Make the number of bytes to be read from:
remotes/origin/master test for current head
dammit, this is the second time this has reverted
modified index to create refs/heads if it is not there
Add diff-ics dependency
Add dependency for Open4
merged recent changes:
updated the Manifest file
fixed alternates to accept relative paths and changed 1
added sha() def to commit object
Merge branch 'idx2'
merged in bryces changes and fixed some testing is
Clarify how to get a full count out of Repo#comm
Appendix: time, test fix from bryce

2a8c38b72380fc64700d0e8e6d7b16d45dfb6ebd ← Rev 1 / 194

Find next prev commit containing: Exact All fields

Search
Diff Old version New version Lines of context: 5 Ignore space change
Author: Jos Backus <jos@catnook.com> 2009-01-30 17:49:40
Committer: Scott Chacon <schacon@gmail.com> 2009-01-30 18:23:08
Parent: 66933d0e3329c70e8e598c2246a8e6af90004646 (test for current head)
Branch: master
Follows: v0.7.0
Precedes:

Make the number of bytes to be read from git's stdout configurable.

Signed-off-by: Scott Chacon <schacon@gmail.com>

lib/grit/git.rb
index 7c1785d..fe3d0b5 100644
@@ -22,11 +22,19 @@ module Grit
  include GitRuby

  class << self
    attr_accessor :git_binary, :git_timeout
+   attr_accessor :git_binary, :git_timeout, :git_max_size
  end

  self.git_binary = "/usr/bin/env git"
```



# 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 :

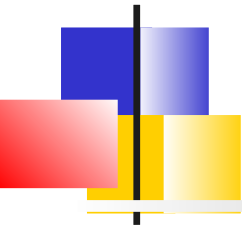
```
$ git checkout serverfix
```

Si l'on veut renommer la branche

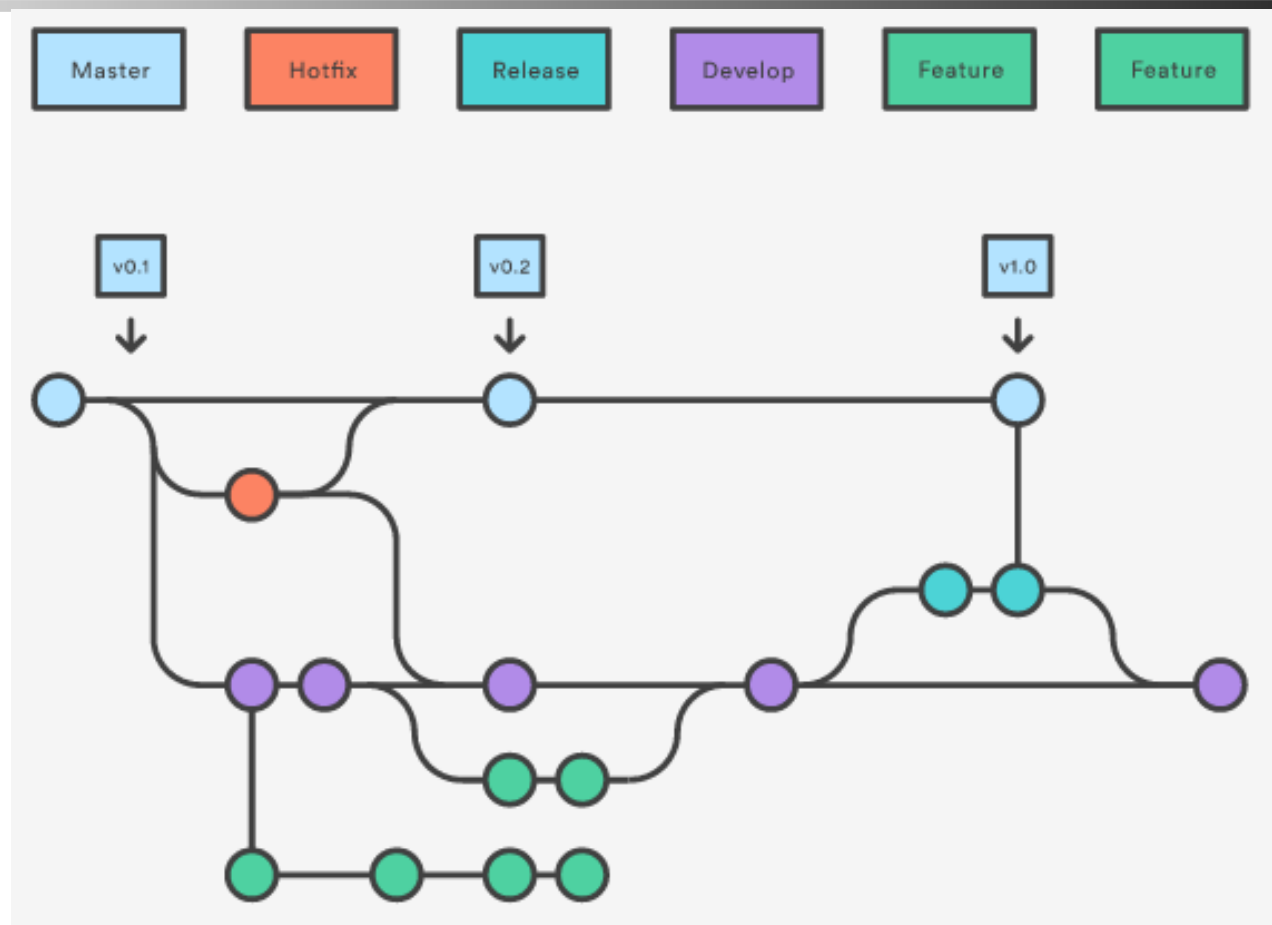
```
$ git checkout -b sf origin/serverfix
```

Enfin, si on veut utiliser une branche locale existante :

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



# Gitflow





# 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)





# Gitlab CI/CD

---

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



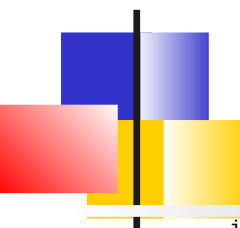
# Personnalisation

---

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
```