



GitLab CI/CD

David THIBAU - 2024
david.thibau@gmail.com



Agenda

Introduction

- DevOps et CI/CD
- La plateforme Gitlab

Workflow de collaboration

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

Concepts Gitlab-CI

- Introduction
- Jobs et Runners
- UI pipeline

Syntaxe .gitlab-ci.yml

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

Support Gitlab pour pipeline standard

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



Introduction

DevOps et CI/CD
La plateforme Gitlab



Objectif DevOps

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



Approche en continu

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

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

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

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



Pipelines

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

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



Distinction CI/CD





Pipeline et les containers

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

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



Introduction

DevOps et CI/CD

La plateforme Gitlab



Introduction

Gitlab se définit comme une plateforme DevOps complète qui inclut :

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



Community vs Enterprise

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

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

Les versions payantes apportent généralement :

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



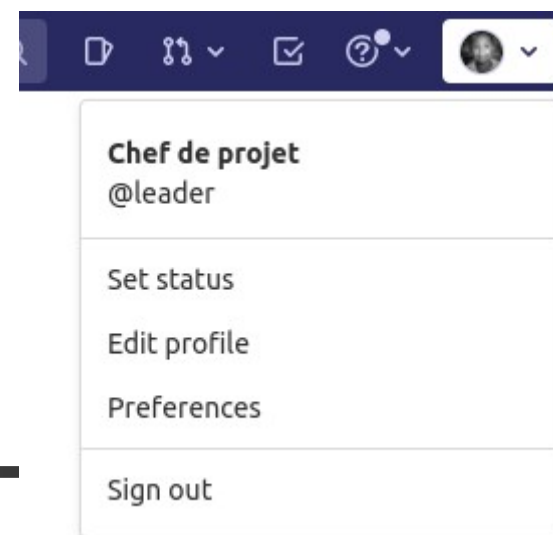
Interface utilisateur

2 grand profils utilisateurs accèdent à la plateforme :

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



Menu *User Settings*



Profile : Édition du profil utilisateur

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

Applications : Se connecter avec un fournisseur OAuth2

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

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

Emails : Possibilité d'associer plusieurs emails au compte

Password : Modification mot de passe

Notifications : Configurer le niveau de notifications de Gitlab

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

GPG Keys : Pouvoir signer des tags

Preferences : Personnalisation de l'UI

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

Authentication Log : Journal des authentifications



Mise en place clés ssh

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

2 étapes :

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



Mise en place

- Environnement Linux :

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

Ou

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

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

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



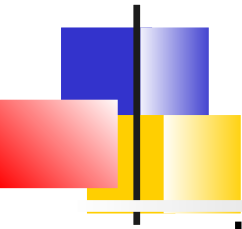
Workflows de collaboration

Projets et Membres

Repository Gitlab

Les MergeRequest

GitlabFlow et ses déclinaisons



Projets

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

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

3 visibilité sont possibles pour un projet :

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



Membres

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

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

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



Groupes

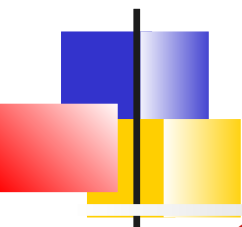
Afin de faciliter la gestion des projets et des membres associés, il est possible de définir des **groupes de projets**.

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

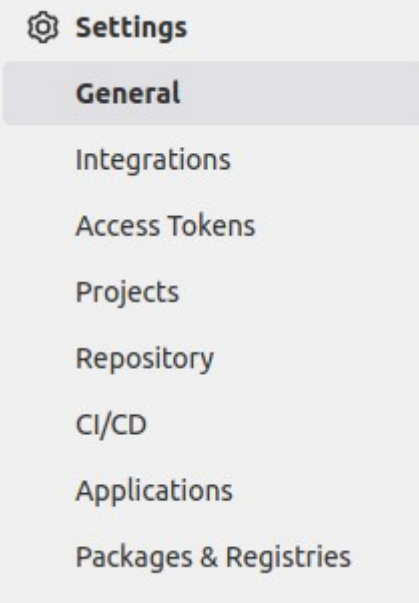
Il sera possible de visualiser toutes les issues et Merge Request des projets du groupe

Les groupes peuvent être hiérarchiques

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



Menu Groupe Settings



Group Information → Members : Ajout de membres

General : Nom et visibilité

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

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

Projects : Projets du groupe

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

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

Applications : Fournisseur OAuth2

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



Création de projet

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

Il peut s'agir :

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

Lors de la création, il faut définir :

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

Création projet vierge à partir d'un groupe



Create blank project

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

New project › Create blank project

Project name

Project URL



Project slug

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

Project description (optional)

Visibility Level

☒  Private

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

Project Configuration

☒ Initialize repository with a README

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



Menu Projet

Projects : Activité, Labels et membres

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

Issues : Gestion des issues, tableau de bord Kanban

Merge requests : Travaux en cours

CI/CD : Historique d'exécution des pipelines

Security & compliance : Rapports sur les détections de vulnérabilités

Deployments : Gestion des environnements de déploiement

Monitor : Information de surveillance du projet

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

Packages et registries : Accès aux dépôts d'artefacts

Wiki : Documentation annexe

Snippets : Bouts de code

Settings : Configuration

D Delivery Service

Project information

Repository

Issues 0

Merge requests 0

CI/CD

Security & Compliance

Deployments

Monitor

Infrastructure

Packages & Registries

Analytics

Wiki

Snippets

Settings

Menu Projet → Settings

General :

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

Integrations : Intégration application tierces

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

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

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

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

Monitor : Configuration du monitoring

Usage Quotas : Définition de quotas de stockage



Workflows de collaboration

Projets et Membres

Repository Gitlab

Les MergeRequest

GitlabFlow et ses déclinaisons



Particularités *Gitlab*

On peut interagir avec les dépôts GitLab via :

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

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

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

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

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

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

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



Clonage d'un repo

Plusieurs options pour cloner un dépôt :

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



Commits

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



Analytiques proposées

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

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

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

Un graphique dédié aux contributeurs



Vues proposées

Settings → Contributors : Les contributeurs au code

Repository → Commits : Historique des commits

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

Repository → Graph : Vue graphique des commits et merge

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



Les branches

Dans Gitlab, les branches peuvent avoir des caractéristiques particulières :

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

Le mainteneur est donc responsable de :

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



Branche par défaut

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

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

La branche par défaut a certaines particularités :

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



Branches protégées

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

Des permissions sont associées à une branche protégée :

- ***Allow to Merge*** : Qui peut y fusionner une autre branche.
- ***Allow to Push*** : Qui peut y faire un push
- ***Force Push*** : Les personnes ayant le droit push peuvent elles faire des force push¹.

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

-stable, production/

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



Création de branche

Plusieurs façons de créer des branches avec Gitlab :

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



Workflows de collaboration

Projets et Membres
Repository Gitlab

Les MergeRequest

GitlabFlow et ses déclinaisons



Patterns de workflow

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

Cependant, certains patterns de collaboration sont documentés :

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



Merge Request gitlab

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

A chaque démarrage, d'une nouvelle tâche,

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



Création de Merge Request

Plusieurs façons pour créer une MR :

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



Cycle de vie d'une MR

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



Propriétés d'une MR

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

- Une description rich text
- Une ou plusieurs^u personnes assignées
- Un ou plusieurs^u reviewers
- Un milestone
- Un ou plusieurs labels
- Des options de merge :




Onglets d'une MR

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

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

Resolve "Implémenter l'interaction avec banques"

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

 Merged **mouhamadou bamba badjinka** requested to merge [6-implementer-l-interacti...](#) into [main](#) 5 days ago

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

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



Pipeline #1231758956 failed

Pipeline failed for [18dc461e](#) on [6-implementer-l-interaction-avec-banques](#) 3 days ago



8✓ Approval is optional



 Merged by  [David THIBAU](#) 3 days ago

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

Merge details

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

Pipeline #1231763196 failed

Pipeline failed for [2253fd14](#) on [main](#) 3 days ago



Activity

All activity  

- **mouhamadou bamba badjinka** requested review from [@badjinka](#) 5 days ago
- **mouhamadou bamba badjinka** assigned to [@dthibau](#) 5 days ago
- **mouhamadou bamba badjinka** added 1 commit 5 days ago
 - [af44fecd](#) - inter action avec banque
[Compare with previous version](#)
- **mouhamadou bamba badjinka** added 1 commit 5 days ago
 - [5f8ab2cf](#) - inter action avec banque V2
[Compare with previous version](#)
- **David THIBAU** added 1 commit 5 days ago
 - [65d26fd4](#) - Configuration eureka et config
[Compare with previous version](#)



Assignee

[Edit](#)

 [David THIBAU](#)

Reviewer

[Edit](#)

 [mouhamadou bamba badjinka](#) 

Labels

[Edit](#)

None

Milestone

[Edit](#)

None

Time tracking



No estimate or time spent

2 Participants





Commentaires et discussions

Des **commentaires** peuvent être associés aux MR

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

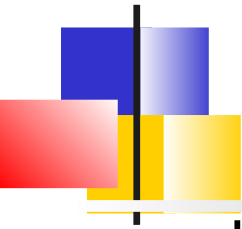
Un commentaire peut être transformé en **discussion/thread**.

Une discussion/thread regroupe plusieurs échanges et a un statut

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

Il est possible de

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



Revue de code

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

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

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

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



Configuration des MR

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

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



Workflows de collaboration

Projets et Membres

Repository Gitlab

Les *MergeRequest*

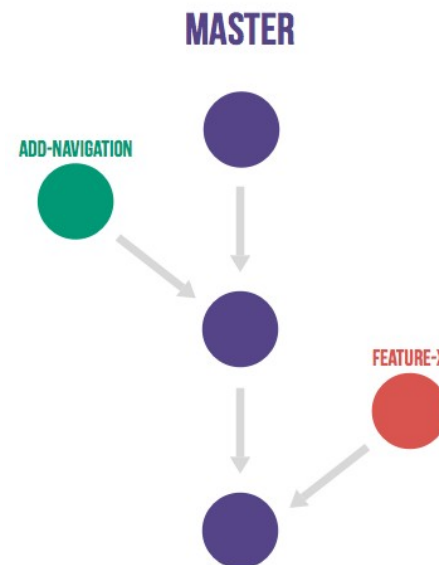
GitlabFlow et ses déclinaisons



Gitlab Flow

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

Ce type de workflow peut convenir à des projet DevOps simple





Déclinaisons DevOps

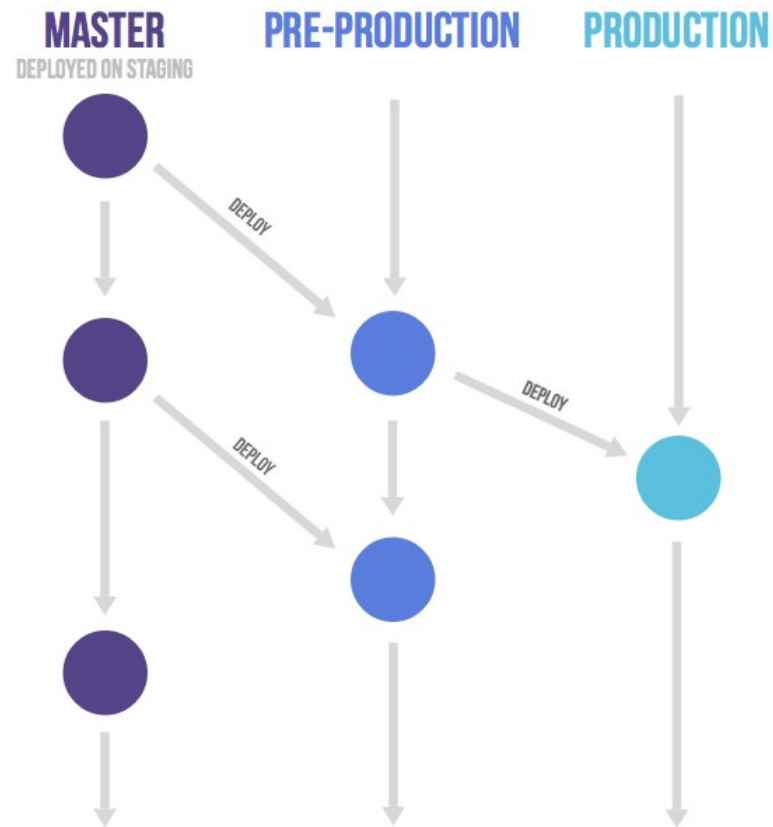
Il est cependant possible de modifier la configuration par défaut en créant d'autres branches et en identifiant les moyens de mettre à jour ces branches.

Dans un projet DevOps¹, on peut introduire par exemple :

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

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

Déclinaison avec qa



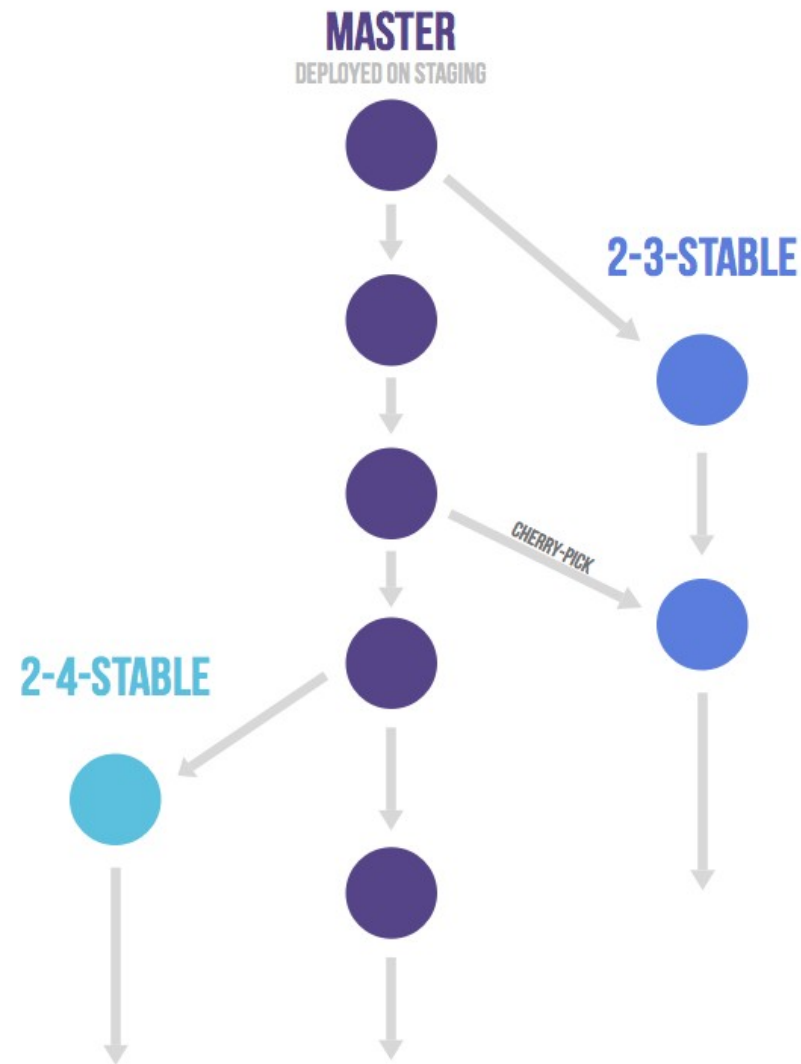


Déclinaisons Release

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

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

Branches de releases (Gitflow)



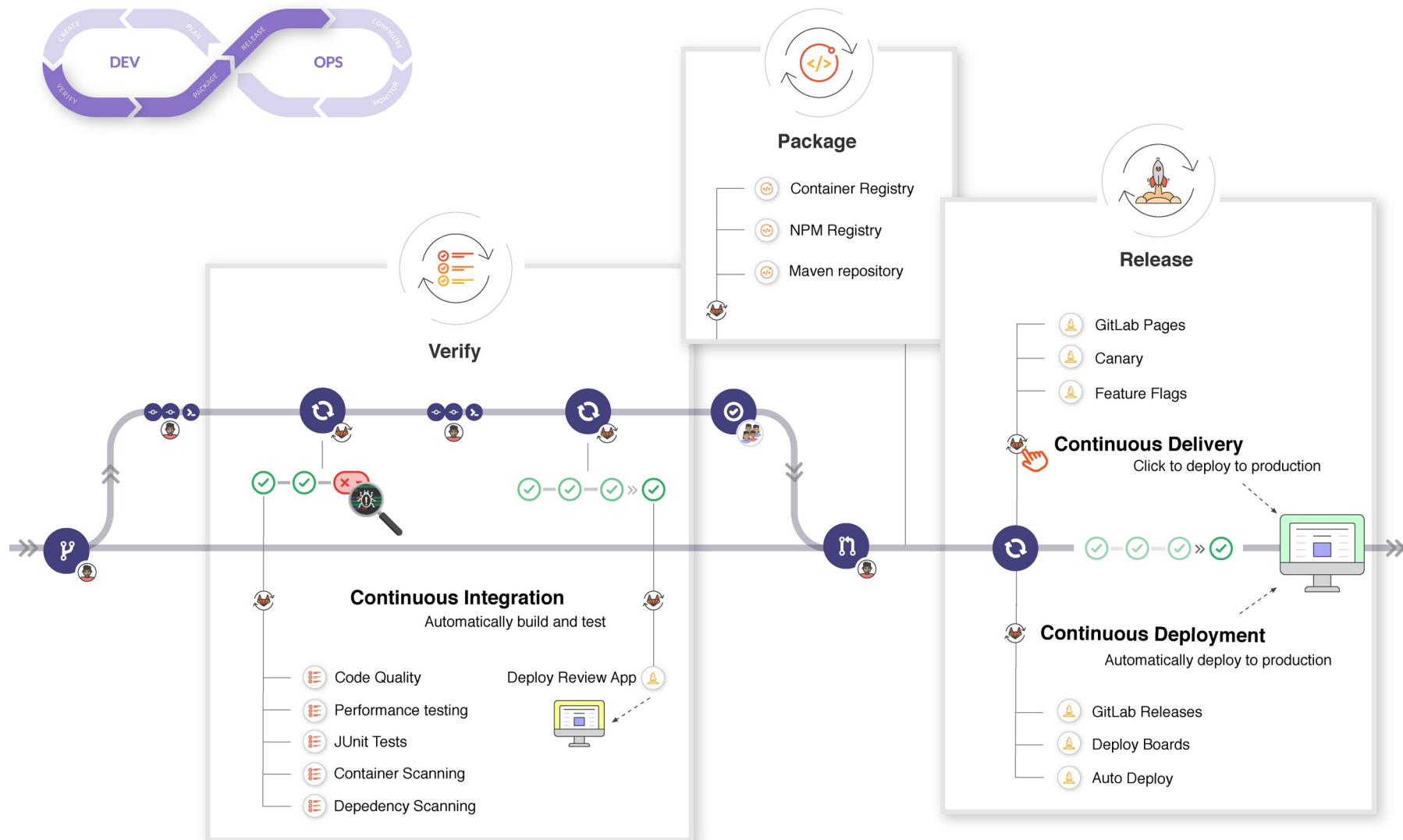


Fusions entre branche

La fusion entre branches peut s'effectuer via :

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

Exemple pipeline





Concepts Gitlab-ci

Jobs et Runners
UI Pipelines



Runner

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

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

Un runner peut être

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



Type de Runners

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

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

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

Pour disposer d'un runner :

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



Installation GitlabRunner

L'installation s'effectue :

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



Enregistrement

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

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

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



Exécuteurs

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

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



Affectation d'un runner et tags

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

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

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



Concepts Gitlab-ci

Jobs et Runners
UI Pipelines



Editeur

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

Il permet une validation de la syntaxe

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

Des gabarits sont également disponibles pour la plupart des technologies

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



Exécution des pipelines

Les pipelines s'exécutent
automatiquement à chaque push

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

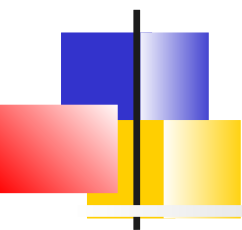


Tableau de bord d'exécution

Status	Pipeline	Triggerer	Stages	
<div>⏸ blocked</div> <div>⌚ 00:13:00</div>	<div>Adding probe</div> <div>#123967683 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>#123967122 1-changer-hello-en-bonjour d9111678 </div> <div>latest Auto DevOps</div>		<div>✓ ✓ ✓ ✓ ! ✓</div>	<div>▶ ⌵ ↺ ⬇ ⌵</div>

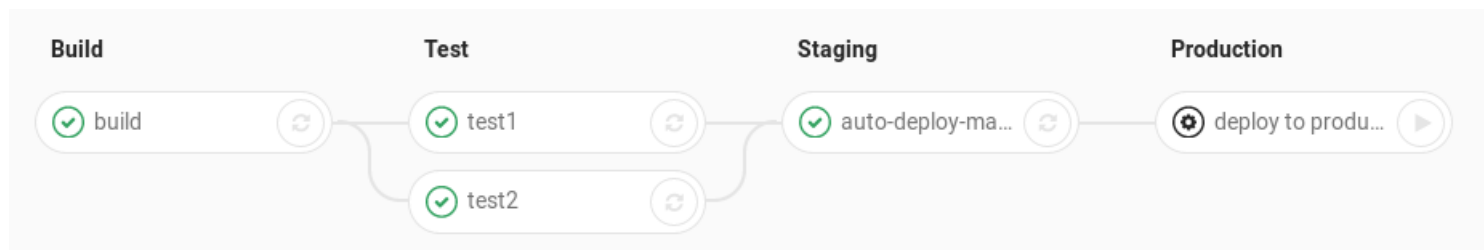
Un tableau de bord permet de voir les dernières exécutions de pipeline, leurs status, le commit, le déclenchement et l'exécution des tâches

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



Visualisation d'une pipeline

Le détail d'une pipeline est affichée graphiquement.



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

De déclencher une tâche manuelle



Syntaxe gitlab-ci.yml

Basiques *.gitlab-ci.yml*

Principales Directives

Réutilisation

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.

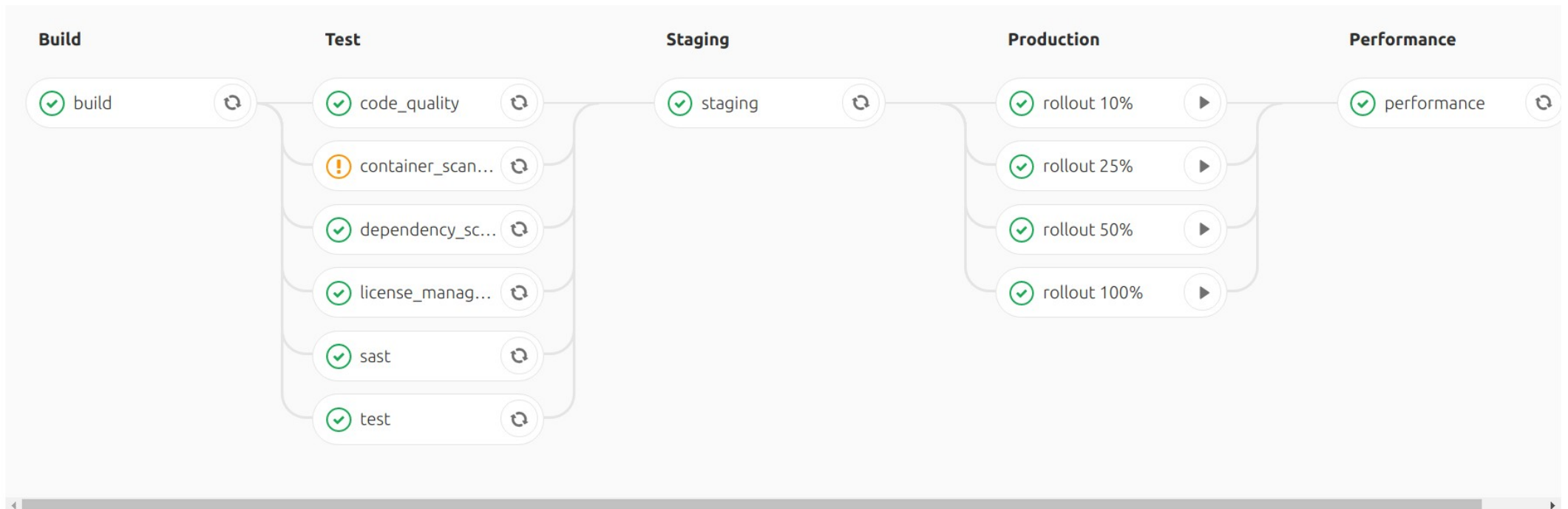
PréRequis :

- 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 dans un environnement de staging
 - Production : Roll-out manuel de la production
 - Performance : Test de performance en prod

AutoDevops sur branche main





Jobs / phases / tâches

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

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

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

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

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



stages

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

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

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

- Exemple :

stages:

- build
- test
- deploy



Jobs

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

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

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



Exemple

stages:

- Build
- Test
- Staging
- Production

build:

stage: Build

script: make build dependencies

test1:

stage: Test

script: make build artifacts

test2:

stage: Test

script: make test

auto-deploy-ma:

stage: Staging

script: make deploy

deploy-to-production:

stage: Production

script: make deploy



Directive needs

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

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

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



Contexte des directives

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

Les principales directives globales sont :

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

Variables

Le job peut accéder à un ensemble de variables :

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

Une variable peut être configurée comme étant masquée ou protégée¹

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

Ex :

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

1. Accessible seulement des branches protégées



Syntaxe gitlab-ci.yml

Basiques *.gitlab-ci.yml*

Principales Directives

Réutilisation

Environnements et déploiements

Intégration docker



cache

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

Les caches sont :

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



Sous-directive de cache

Les sous-directives possibles sous cache sont :

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



Examples

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




Artifacts

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

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

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

Ils sont conservés 1 semaine (par défaut)

Ils sont téléchargés par défaut par les jobs en aval

pdf:

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

```
artifacts:
```

```
  paths:
```

```
    - mycv.pdf
```

```
  expire_in: 1 week
```



Sous-directives artifacts

Sous la directive artifacts peuvent être précisés :

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



Exemples

```
artifacts:
  expose_as: 'Exécutable'
  paths:
    - binaries/
    - .config
  exclude:
    - binaries/**/*.*
---
job:
  artifacts:
    access: 'developer'
---
rspec:
  stage: test
  script:
    - bundle install
    - rspec --format RspecJunitFormatter --out rspec.xml
  artifacts:
    reports:
      junit: rspec.xml
---
```



Réutilisation des artefacts

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

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

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



Réutilisation des artefacts (2)

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

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

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



GIT_STRATEGY

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

La variable peut prendre 3 valeurs :

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



GIT_CHECKOUT

La variable ***GIT_CHECKOUT*** peut être utilisée lorsque *GIT_STRATEGY* est définie à *clone* ou *fetch*

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

Si false :

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

variables:

```
GIT_STRATEGY: clone
```

```
GIT_CHECKOUT: "false"
```

script:

- git checkout -B master origin/master
- git merge \$CI_COMMIT_SHA



Control Flow

allow_failure permet à une tâche d'échouer sans impacter le reste de la pipeline.

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

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

tags : Liste de tags pour sélectionner un runner

parallel : Nombre d'instances du jobs exécutés en parallèle

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



Conditions

when conditionne l'exécution d'un job. 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 (Déprécié !!)



rules

La directives **rules** permet d'inclure ou d'exclure des jobs de la pipeline.

- C'est la même finalité que *only*, *except* mais en plus puissant

Les règles sont évaluées lors de la création du pipeline dans l'ordre de définition

Lorsqu'une correspondance est trouvée, la tâche est incluse ou exclue du pipeline, selon la configuration.

Les règles sont définies avec les mots-clés *if*, *changes*, *exists*, *allow_failure*, *variables*, *when*



Exemple rules:if

job:

script: echo "Hello, Rules!"

rules:

On interdit pour une MR d'une branche feature

vers un autre branche que celle par défaut

- if: \$CI_MERGE_REQUEST_SOURCE_BRANCH_NAME =~ /^feature/ &&
\$CI_MERGE_REQUEST_TARGET_BRANCH_NAME != \$CI_DEFAULT_BRANCH

when: never

Si démarrage manuel, on autorise les échecs

- if: \$CI_MERGE_REQUEST_SOURCE_BRANCH_NAME =~ /^feature/

when: manual

allow_failure: true

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

- if: \$CI_MERGE_REQUEST_SOURCE_BRANCH_NAME



Exemple rules:variables

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



workflow

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

- **auto_cancel** permet d'annuler une pipeline en cours d'exécution
 - **on_new_commit** : Annulation si un nouveau commit survient
 - **on_job_failure** : Nécessite une configuration de l'administrateur, permet d'annuler certains jobs de la pipeline lorsqu'un job est en échec
- **name** permet de nommer la pipeline
- **rules** permet de conditionner l'exécution en fonction des variables prédéfinies fournies par Gitlab



Exemples

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

```
workflow:
```

```
  auto_cancel:
```

```
    on_new_commit: interruptible
```

```
job1:
```

```
  interruptible: true
```

```
  script: sleep 60
```

```
job2:
```

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

```
  script: sleep 60
```



Syntaxe gitlab-ci.yml

Basiques *.gitlab-ci.yml*

Principales Directives

Réutilisation

Intégration docker

Environnements et déploiements



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
```



Syntaxe gitlab-ci.yml

Basiques *.gitlab-ci.yml*

Principales Directives

Réutilisation

Intégration docker

Environnements et déploiements



Docker

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

Il peut être global à la pipeline (directive default) ou spécifique à un job.

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



Syntaxe image

2 syntaxes sont possibles pour image

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

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

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



Docker services

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

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

services:

- tutum/wordpress:latest

alias : wordpress

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



Test du service

Lors de l'exécution du build, le Runner:

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

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

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




Options pour service

4 options disponibles :

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



Variables

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

Exemple service Postgres :

```
services:
```

```
- postgres:latest
```

```
variables:
```

```
POSTGRES_DB: nice_marmot
```

```
POSTGRES_USER: runner
```

```
POSTGRES_PASSWORD: ""
```



Construction d'image

Un scénario désormais classique du CI/CD est:

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

En commande docker :

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



Configuration du runner

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

- Avec l'exécuteur shell et une pré-installation de docker sur le runner
- Avec l'exécuteur docker et :
 - l'image docker (image contenant le client docker),
 - ainsi que le service docker-in-docker permettant de disposer d'un daemon docker
- Avec l'exécuteur docker, une pré-installation 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



Exécuteur shell

1. Enregistrer un exécuteur Shell sur le runner :

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

2. Installer docker sur la machine hébergeant le runner

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

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

4. Vérifier que gitlab-runner a accès à docker

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

5. Tester la pipeline :

```
before_script:
```

```
- docker info
```

```
build_image:
```

```
script:
```

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

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



Docker in Docker (1)

Enregistrer un exécuteur docker en mode
privilège

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



Docker in Docker (2)

Tester dans un *.gitlab-ci.yml*

image: **docker:stable**

variables:

DOCKER_HOST: tcp://docker:2375/

DOCKER_DRIVER: overlay2

services:

- **docker:dind**

before_script:

- docker info

...



Association de socket (1)

Enregistrer un runner avec une association de socket :

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




Association de socket (2)

image: **docker:stable**

before_script:

- docker info

build:

stage: build

script:

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



Registre Gitlab

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

Gitlab dans sa version entreprise propose un registre de conteneur.

Pour l'utiliser, il faut :

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



Authentification auprès du registre Gitlab

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

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



Exemple

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



Syntaxe gitlab-ci.yml

Basiques *.gitlab-ci.yml*

Principales Directives

Réutilisation

Intégration docker

Environnements et déploiements



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.

Operate → 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

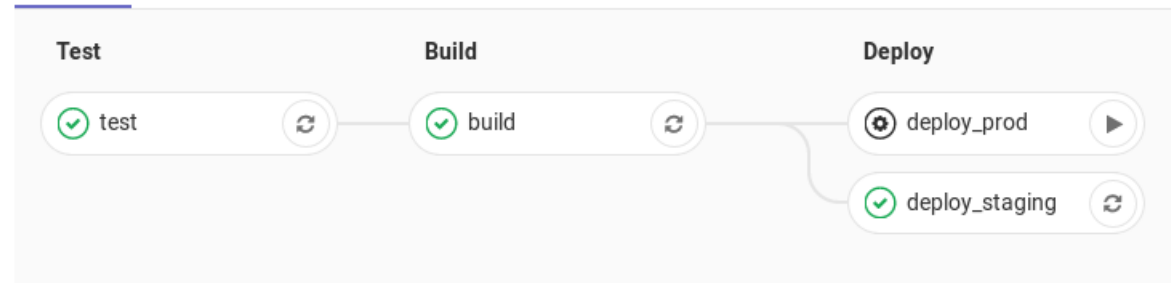
L'ajout de *when: manual* convertit le job en un job manuel et expose un bouton Play dans l'UI

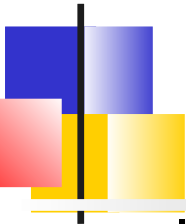
Use busybox

🕒 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
```





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
```



Support Gitlab pour pipeline standard

Construction et tests développeur

Analyses statiques

Dépôts d'artefacts

Release

Gestion de l'infrastructure

AutoDevOps



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
- ...

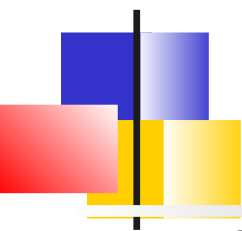
Gitlab s'appuie alors sur des outils de la pile technologique qui sont soit préinstallés sur des runners ou dans des images docker



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

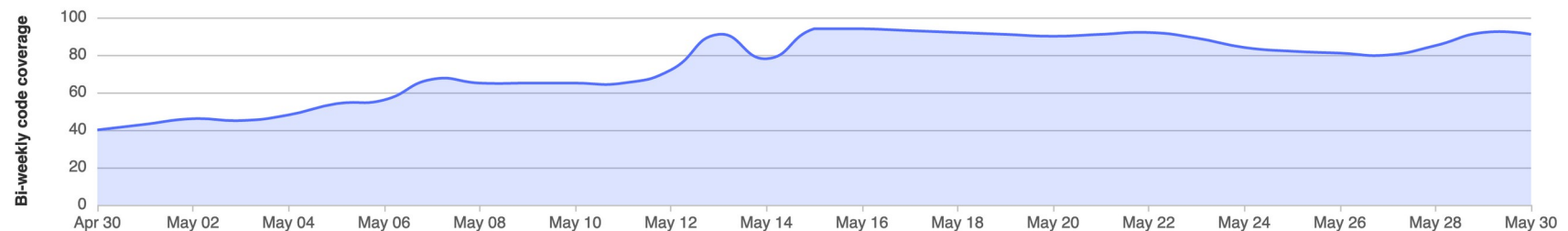
91.98%



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

Release

Gestion de l'infrastructure

AutoDevOps



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**¹ et ses plugins pour analyser le code source.

Les résultats sont disponibles² :

- 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.

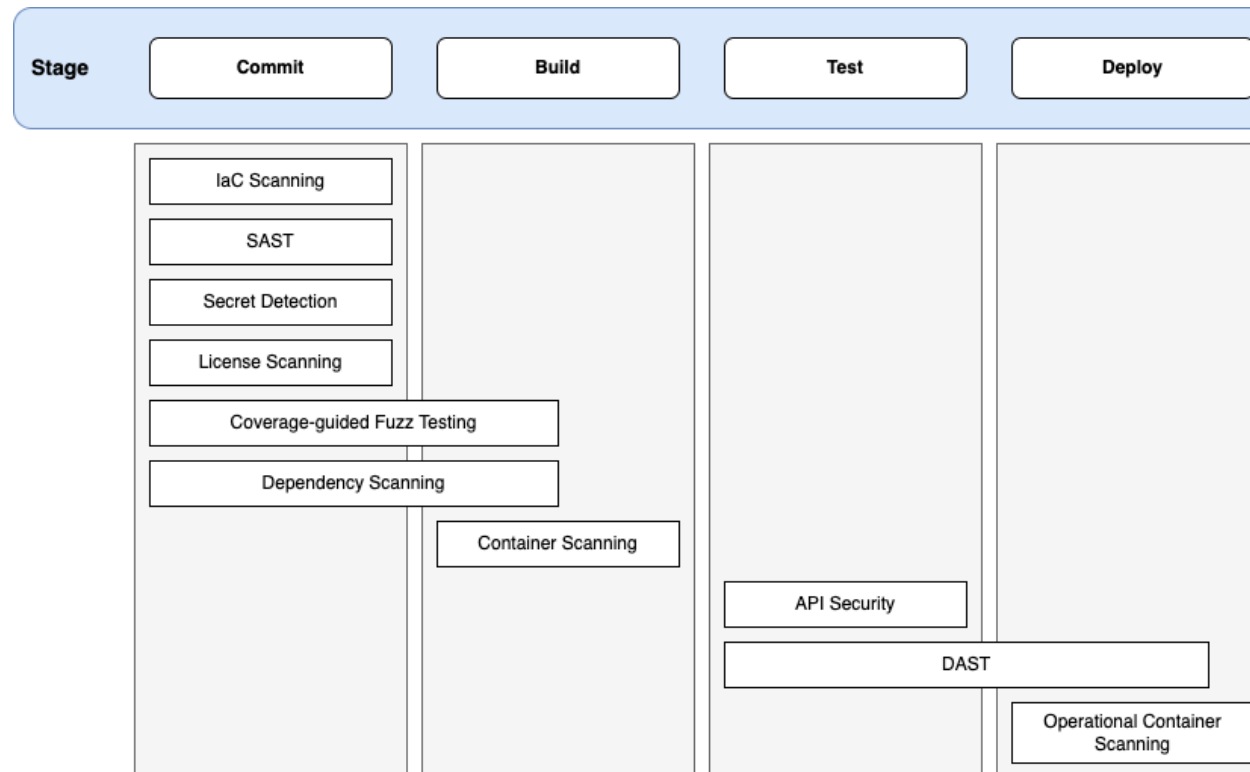
Cela couvre :

- 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
```

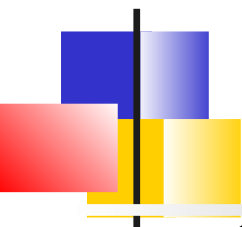
Infrastructure as Code (IaC) Scanning

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



Phases des pipelines

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



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



Phases des pipelines

Construction et tests développeur

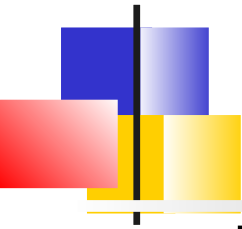
Analyses statiques

Dépôts d'artefacts

Release

Gestion de l'infrastructure

AutoDevOps



Release

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

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



Contenu et création d'une release

Une release peut contenir :

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



Création : Edition

Une *release* peut être créée

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

Après avoir créé une release, on peut :

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



Création de release via un job CI/CD

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

3 méthodes typiques :

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



Exemple

Création lors fusion dans la branche par défaut

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



Phases des pipelines

Construction et tests développeur

Analyses statiques

Dépôts d'artefacts

Release

Gestion de l'infrastructure

AutoDevOps



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

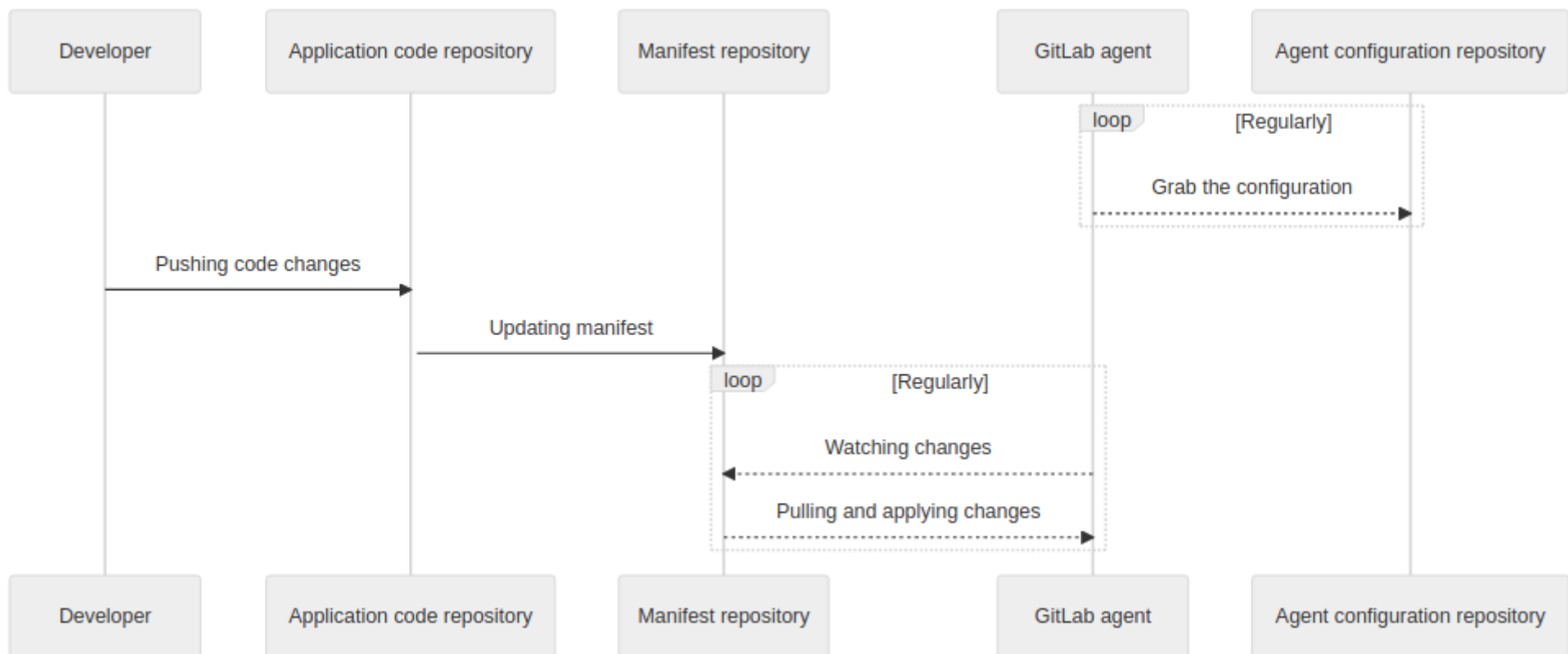
L'interface permet de créer des clusters vers 3 fournisseurs : Google GKE, Amazon EKS, Civo

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
```



Phases des pipelines

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



Introduction

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

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

Auto DevOps peut également être explicitement activé



Pré-requis

Pré-requis nécessaire :

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



Stratégies

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

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



Phases (1)

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

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

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

Auto SAST : Exécute une analyse statique pour détecter des vulnérabilités

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

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

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



Phases (2)

Auto Review Apps : Déploie vers un cluster Kubernetes

Auto DAST : Test dynamique de la sécurité avec OWASP ZAPProxy

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

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

Migrations : Possibilité de configuration de scripts de migration Postgres

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

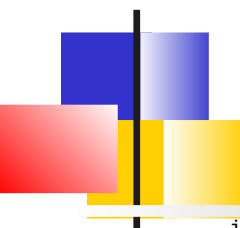


Personnalisation

Il est possible de personnaliser la pipeline via

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

New File → Template AutoDevOps



Template AutoDevOps (1)

```
image: alpine:latest
```

```
variables:
```

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

```
POSTGRES_USER: user
```

```
POSTGRES_PASSWORD: testing-password
```

```
POSTGRES_ENABLED: "true"
```

```
POSTGRES_DB: $CI_ENVIRONMENT_SLUG
```

```
POSTGRES_VERSION: 9.6.2
```

```
KUBERNETES_VERSION: 1.11.10
```

```
HELM_VERSION: 2.14.0
```

```
DOCKER_DRIVER: overlay2
```

```
ROLLOUT_RESOURCE_TYPE: deployment
```

```
stages:
```

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



Template *AutoDevOps* (2)

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

```
# Override DAST job to exclude master branch
```

```
dast:
  except:
    refs:
      - master
```



Annexes

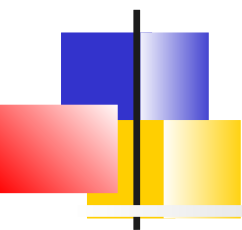
Rappel Kubernetes



Auto-correctif

Kubernetes va TOUJOURS essayer de diriger le cluster vers son état désiré.

- **Moi**: «Je veux que 3 instances de Redis toujours en fonctionnement.»
- **Kubernetes**: «OK, je vais m'assurer qu'il y a toujours 3 instances en cours d'exécution. »
- **Kubernetes**: «Oh regarde, il y en a un qui est mort. Je vais essayer d'en créer un nouveau. »

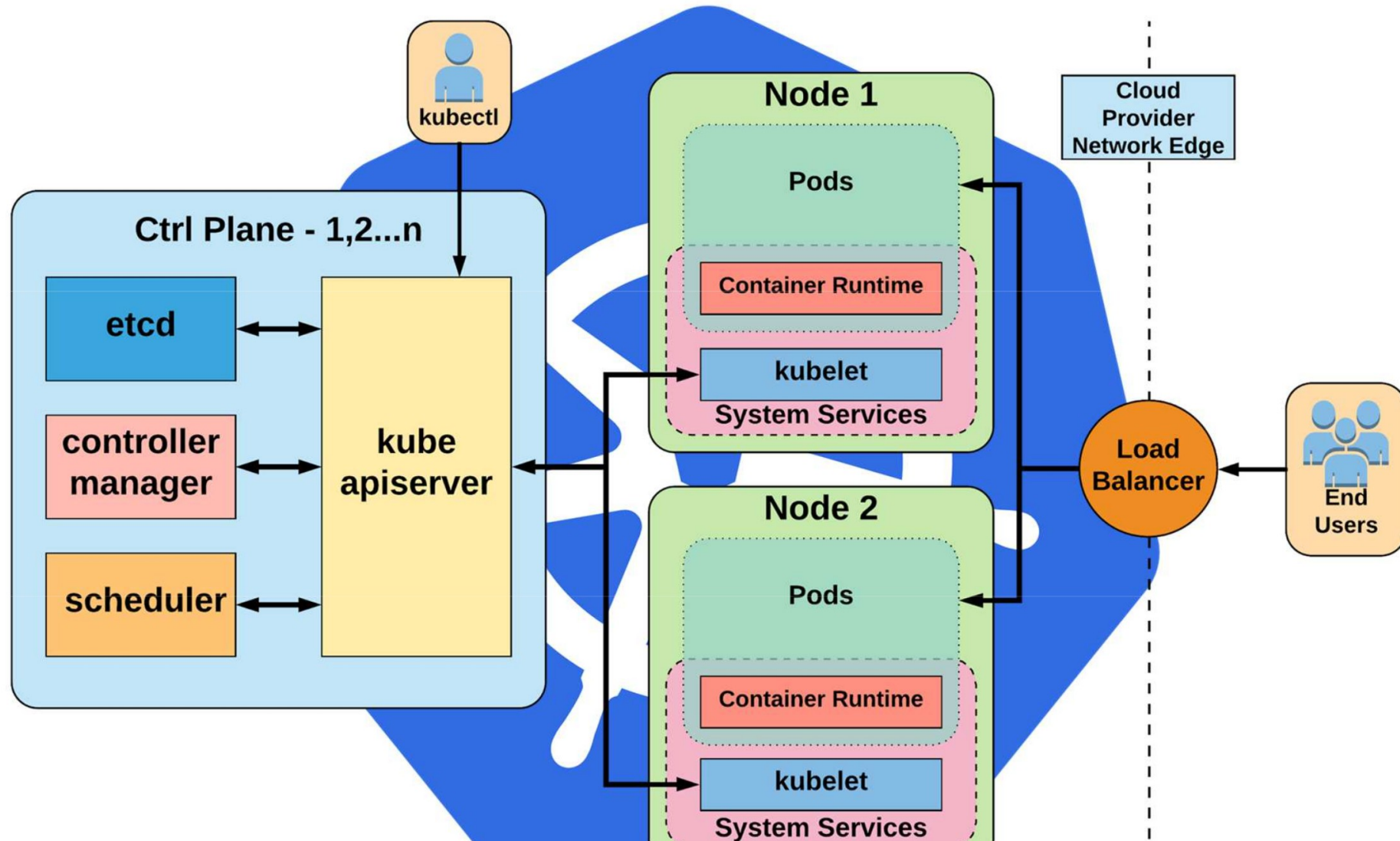


Fonctionnalités applicatives

- Scaling automatique
- Déploiements Blue/Green
- Démarrage de jobs planifiés
- Gestion d'application Stateless et Stateful
- Méthodes natives pour la découverte de services
- Intégration et support d'applications fournies par des tiers (*Helm*)

pod = 1 ou plusieurs conteneurs co-localisés

Architecture cluster





API

L'interaction se fait par une API Rest très riche.

L'API est très cohérente et tous les appels suivent le même format

Format:

`/apis/<group>/<version>/<resource>`

Examples:

`/apis/apps/v1/deployments`

`/apis/batch/v1beta1/cronjobs`

L'outil ***kubectl*** et le format ***yaml*** sont les plus appropriés pour effectuer les requêtes REST

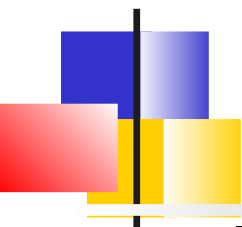


Principes

L'API est une API Rest, elle permet principalement des opérations CRUD sur des **ressources**

En particulier, le client *kubectl* propose les commandes :

- **create** : Créer une ressource
- **get** : Récupérer une ressource
- **edit/set** : Mise à jour d'une ressource
- **delete** : Suppression d'une ressource



Ressources applicatives

Les principales ressources d'une application sont :

- **deployment** : Un déploiement, les déploiements font référence à des *ReplicaSet*, ils peuvent être historisés
- **replicaSet** : Ils définissent le nombre d'instances maximales pour une image de conteneur applicative
- **pod** : Ce sont des conteneurs qui s'exécutent, ils sont distribués sur les nœuds par le scheduler de *Kubernetes*
- **service** : Ce sont des point d'accès stable à un service applicatif

pod

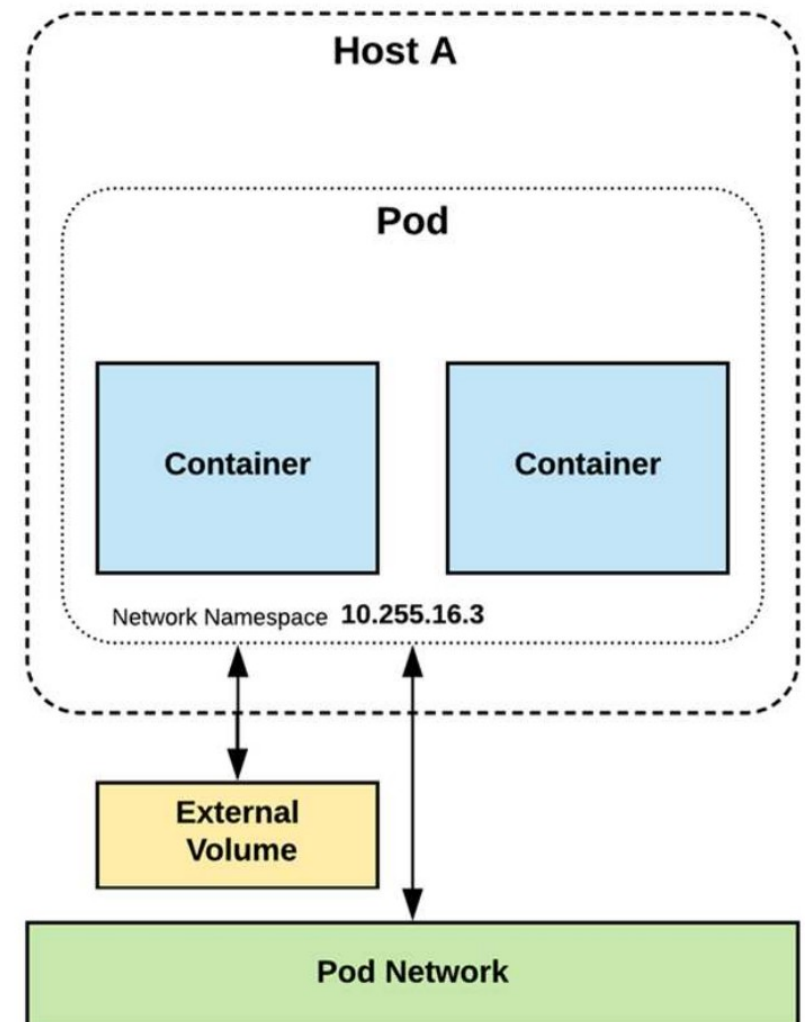
Un **pod** est la plus petite unité de travail

Un *pod* regroupe un ou plusieurs conteneurs qui partagent :

- Une adresse réseau
- Les mêmes volumes

Les pods sont éphémères. Ils disparaissent lorsqu'ils :

- Sont terminés
- Ont échoués
- Sont expulsés par manque de ressources





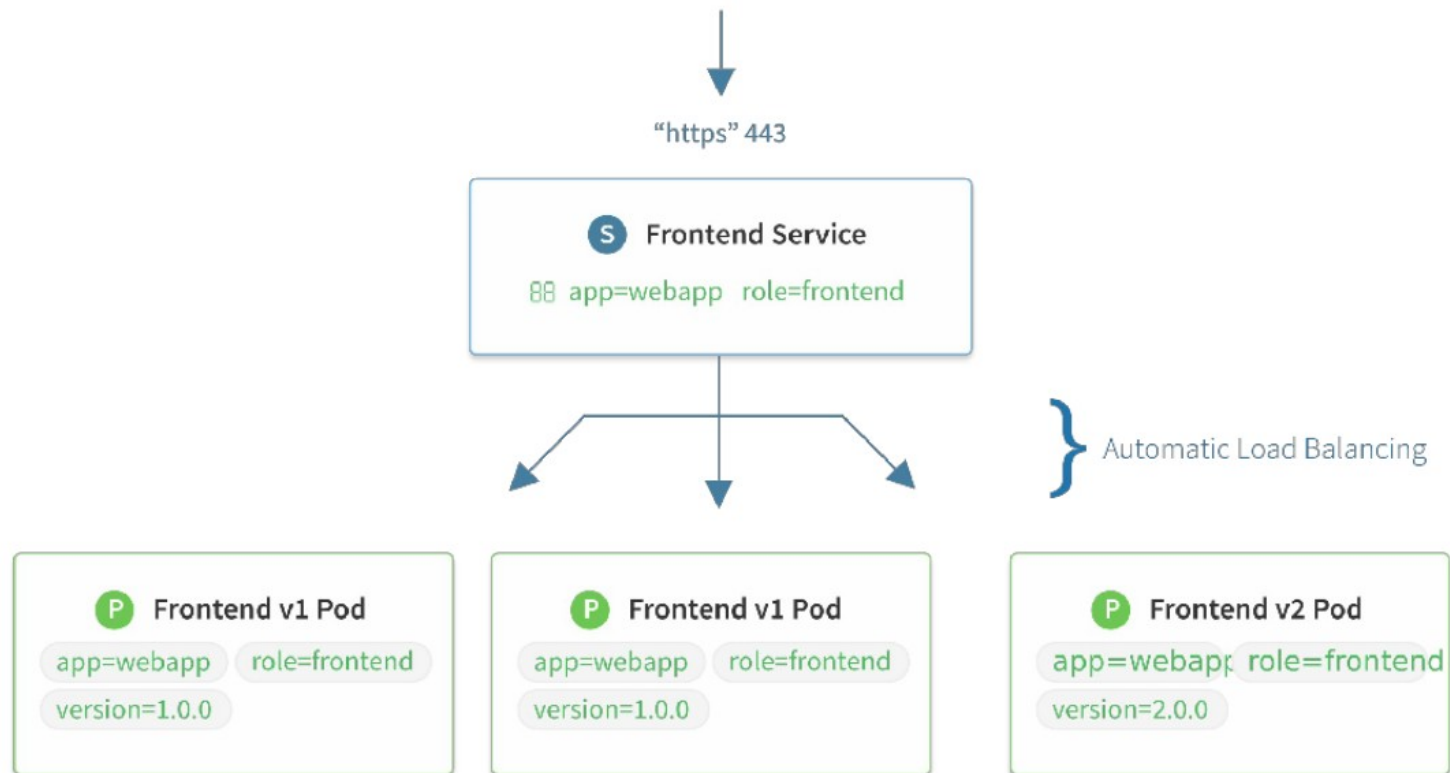
Services

Un **service** est une méthode unifiée d'accès aux charges de travail exposées des *Pods*.

Ressource durable. Les services ne sont pas éphémères :

- IP statique du cluster
- Nom DNS statique (unique à l'intérieur d'un espace de nom)

Service





Ressource *deployment*

Exemple description d'un déploiement:

```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  spec:
    containers:
      - image: dthibau/annuaire
        name: annuaire
```

A partir de ce type de fichier *.yaml*, on peut créer la ressource via :

kubectl create -f ./my-manifest.yaml



Exemple service

Un service nommé *my-service* qui représente tous les pods ayant le **label *app=MyApp*** et qui mappe son port 80 vers le port 80 des pods

```
kind: Service
  apiVersion: v1
  metadata:
    name: my-service
  spec:
    selector:
      app: MyApp
    ports:
      - protocol: TCP
        port: 80
        targetPort: 80
```



Type de service

Un service peut avoir plusieurs types :

- **ClusterIP (défaut)** : Expose le service sur une IP interne au cluster. Le service n'est pas accessible de l'extérieur
- **NodePort** : Expose le service sur un port statique créé automatiquement sur chaque nœud du cluster. Le service est accessible de l'extérieur via `<ClusterIP>:<NodePort>`
- **LoadBalancer** : Expose le service en externe à l'aide de l'équilibreur de charge d'un fournisseur de cloud.
- **ExternalName** : Mappe le service au contenu du champ `externalName` (par exemple `foo.bar.example.com`),



Commandes *kubectl*

get : Afficher 1 ou plusieurs ressources

describe : Afficher les détails sur une ou plusieurs ressources

create : Crée une ressource à partir d'un fichier ou de stdin.

set : Mettre à jour des attributs sur une ressource

edit : Éditer une ressource

delete : Supprimer des ressources

logs : Afficher les logs d'un container

expose : Exposer un déploiement en tant que service

execute : Exécuter une image particulière sur le cluster

attach : S'attacher à un container qui s'exécute

exec : Exécuter une commande dans un container

port-forward : Forward un ou plusieurs ports d'un pod

cp : Copier des fichiers entre conteneurs

auth : Inspecter les autorisations

...



Exemples

Affiche les paramètres fusionnés de *kubeconfig*

kubectl config view

Liste tous les services du namespace par défaut

kubectl get services

Liste tous les pods de tous les namespaces

kubectl get pods --all-namespaces

Description complète d'un pod

kubectl describe pods my-pod

Supprime les pods et services ayant le noms "baz"

kubectl delete pod,service baz

Affiche les logs du pod (stdout)

kubectl logs my-pod

S'attacher à un conteneur en cours d'exécution

kubectl attach my-pod -i

Exécute une commande dans un pod existant (un seul conteneur)

kubectl exec my-pod -- ls /

Visualiser la consommation mémoire et CPU des pods

kubectl top pod

Écoute le port 5000 de la machine locale et forward vers le port 6000 de my-pod

kubectl port-forward my-pod 5000:6000



La commande *apply*

Dans la pratique, la commande ***apply*** avec en paramètre un fichier *.yaml* décrivant la ressource est la plus adaptée pour des déploiements via *kubectl* :

- Elle peut créer ou modifier la ressource
- Les fichiers *.yaml* décrivant les ressources à déployer sont committés, versionnés dans le dépôt des sources

```
kubectl apply -f ./my-manifest.yaml
```



Déploiement

La ressource **deployment** permet de manipuler un ensemble de *Replicaset* (*ensemble de conteneurs répliqués*)

Les principales actions que l'on peut faire sur un déploiement sont :

- Le **rollout**: Création/Mise à jour entraînant la création des pods en arrière-plan
- Le **rollback**: Permet de revenir à une ancienne version des *ReplicaSets*
- La **scalabilité** horizontale : Permet de mettre en échelle l'application horizontalement
- La mise en pause
- La suppression de vieilles versions



Commandes de déploiement *kubectl*

Mettre à jour une image dans un déploiement existant

Enregistrer la mise à jour

```
kubectl set image deployment/nginx-deployment  
  nginx=nginx:1.9.1 --record
```

Regarder le statut d'un rollout

```
kubectl rollout status deployment/nginx-deployment
```

Obtenir l'historique des révisions

```
kubectl rollout history deployment/nginx-deployment
```

Roll-back sur la version précédente

```
kubectl rollout undo deployment/nginx-deployment
```

Scaling

```
kubectl scale deployment/nginx-deployment --replicas=10
```




Scheduler et Workload

Les actions de l'API sont souvent asynchrones

Pour *Kubernetes*, ces ordres sont considérés comme des **workloads** à exécuter via le scheduler.

Les *workload* sont visibles via l'API, elles comportent 2 blocs de données :

- **spec** : La spécification de la ressource
- **status** : Est géré par *Kubernetes* et décrit l'état actuel de l'objet et son historique.



Autres ressources du cluster

ClusterRole : Rôle avec permissions sur l'API

VolumePersistent : Système de stockage

PersistentVolumeClaims : Demande d'usage d'un volume persistant

ConfigMaps : Stockage clé-valeur pour la configuration

Secrets : Stockage de crédeniels



Namespace

Kubernetes prend en charge plusieurs clusters virtuels soutenus par le même cluster physique.

Ces clusters virtuels sont appelés **espaces de noms**.

- Les noms des ressources doivent être uniques dans un espace de noms, mais pas entre les espaces de noms.
- Chaque ressource *Kubernetes* ne peut être que dans un seul *namespace*

Les *namespaces* sont généralement utilisés dans des clusters utilisés par différentes équipes



Labels et sélecteurs

Les **labels** sont des paires clé / valeur attachées à des objets, tels que des pods, des services, des déploiements

Ils sont utilisés pour organiser et sélectionner des sous-ensembles d'objets.

Les **sélecteurs** permettent de rechercher des objets ayant des labels spécifiques.

Il y a 2 types de sélecteurs: égalité ou ensemble.

- Ils sont utilisés par les opérations *LIST* et *WATCH* de l'API
- Les services et les ReplicaSet utilisent les labels et les sélecteurs pour sélectionner les pods



Annotations

Les **annotations** (*metadata*) permettent d'attacher des métadonnées arbitraires non identifiables à des objets.

- Les clients tels que les outils et les bibliothèques peuvent récupérer ces métadonnées.



Écosystème *Kubernetes*

De nombreux outils peuvent compléter une installation cœur de Kubernetes :

- **CoreDNS** : Permet de déclarer dans un DNS interne les services (qui deviennent accessibles via leur nom)
- **Helm** : Système de gestion de package permettant d'automatiser l'installation d'autres outils (ressources *Kubernetes*)
- **Prometheus** : Monitoring du cluster, généralement associé à Grafana
- **Ingress** : Permettant d'exposer les services à l'extérieur du cluster
- **Istio** : Maillage de service (services mesh), gère les communications inter-pods



Distribution Kubernetes

Kubernetes est disponible en OpenSource mais une installation nécessite encore beaucoup d'expertise ... et beaucoup de ressources

Kubernetes est donc proposé par les acteurs du cloud

- Amazon Elastic Container Service for Kubernetes
- Azure Kubernetes Services
- Google Kubernetes Engine
- Digital Ocean
- ...

Il est également disponible en version « dev » mono-nœud : *microk8s*, *minikube*, *kind*

Des versions en ligne comme : <https://labs.play-with-k8s.com/>

L'outil *Rancher* permet de gérer graphiquement plusieurs installation

Terraform permet de provisionner des cluster (et services) as Code