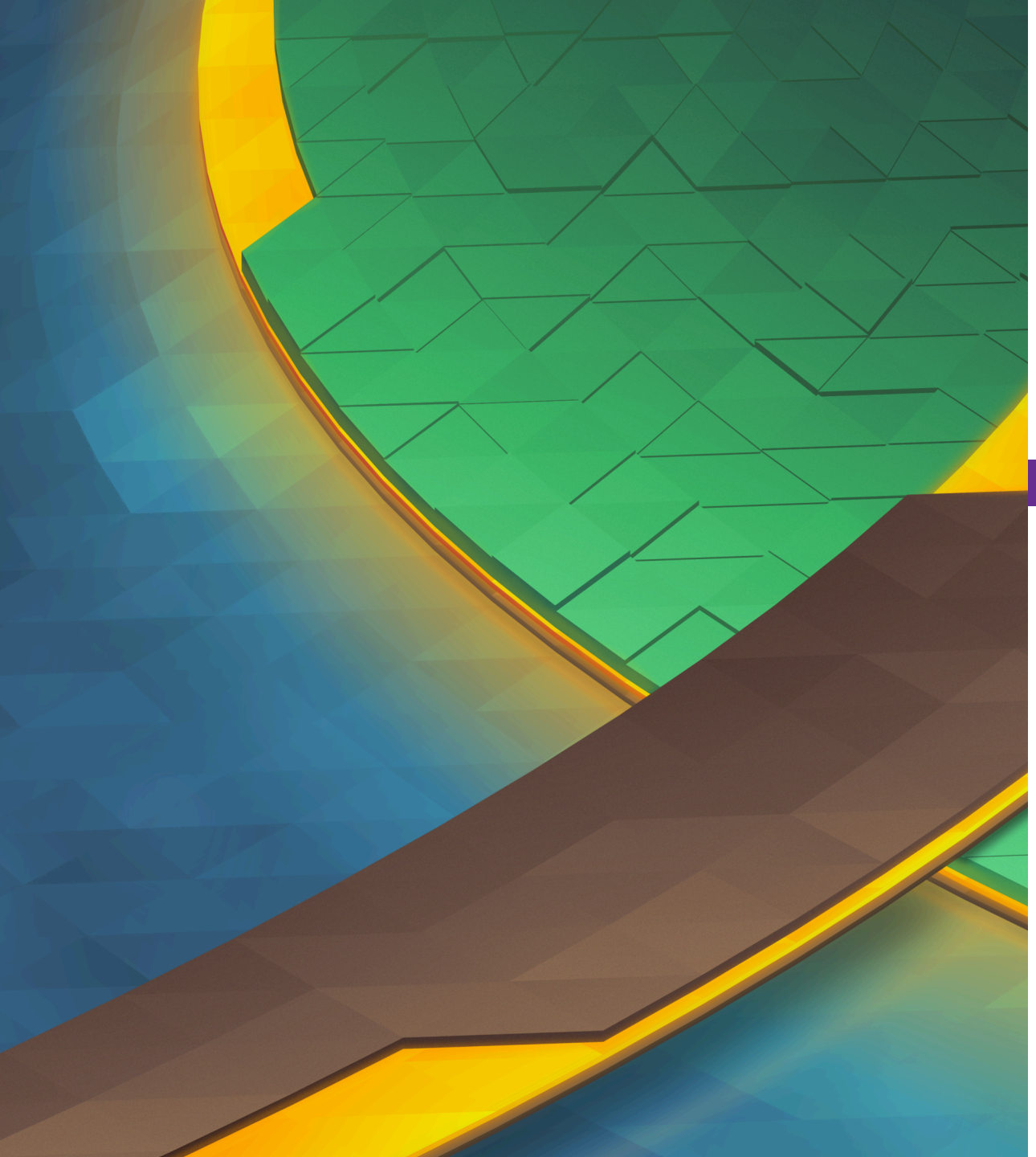
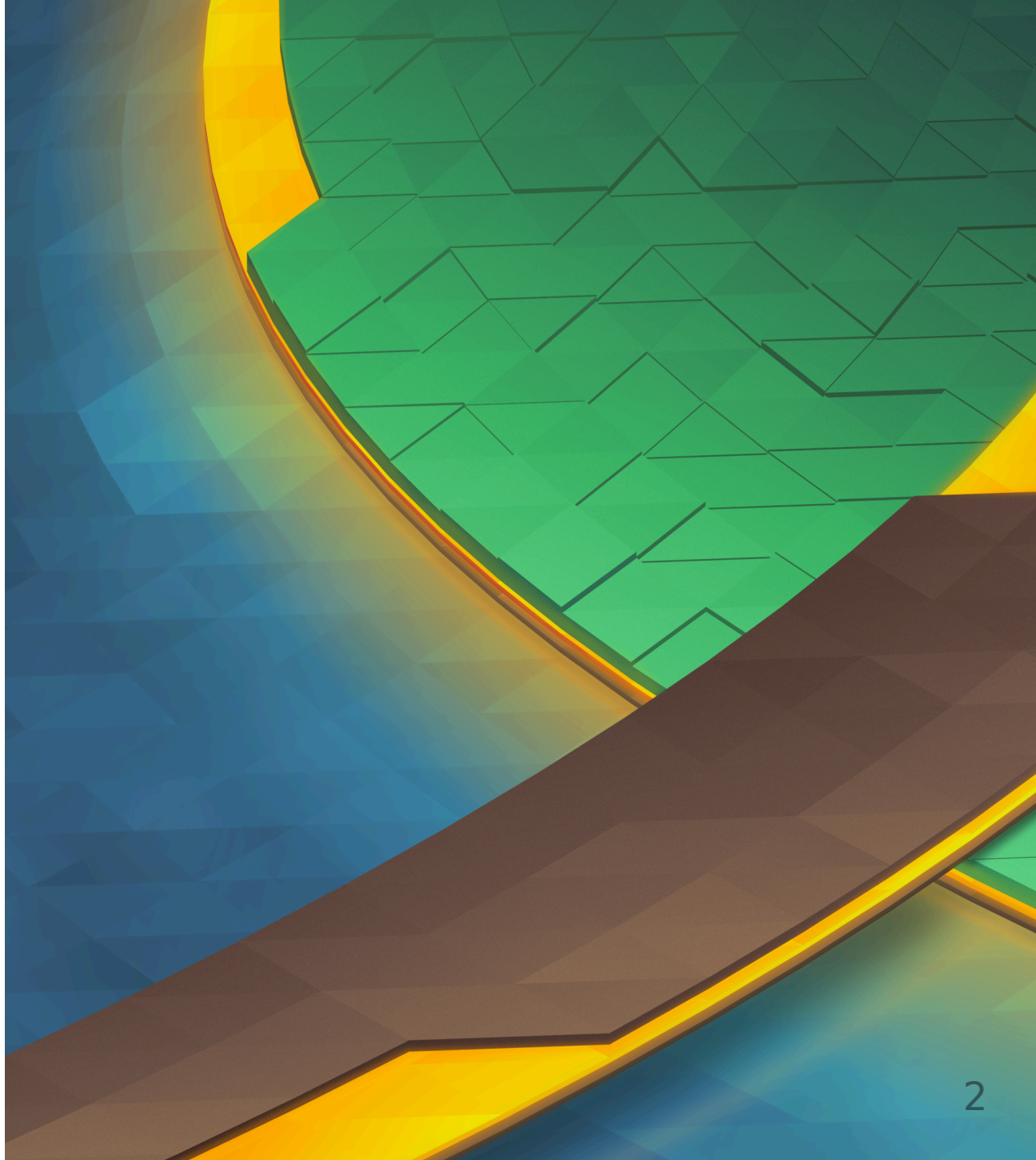


Gitlab et Gitlab-CI

Glenn Y. Rolland
<teaching@glenux.net>

Préambule



Objectifs de la formation

- Maîtriser les fondamentaux de GitLab comme un système de gestion de versions et de collaboration ;
- Structurer et organiser vos projets pour favoriser la réutilisabilité et la maintenance à long terme ;
- Collaborer efficacement au sein d'une équipe avec les outils de tracking et de revue de code intégrés ;
- Comprendre et implémenter des pipelines de GitLab-CI pour le CI/CD ;
- Intégrer et orchestrer des infrastructures complexes en utilisant les capacités de déploiement automatique de GitLab-CI ;
- Créer, tester et déployer des applications de manière agile et sécurisée ;
- Apprendre à monitorer et optimiser vos pipelines de CI/CD pour garantir des performances optimales et une livraison rapide.

Qui êtes vous ?

Petit tour de présentation... avec 3 questions (1 minute chacun)

Passé

Quelle est **votre expérience** ?

Quelles sont **vos compétences** ?

Quel est **votre "bagage"** sur ce sujet ?

Présent

Quelles circonstances vous amènent ici ?

Pourquoi participez-vous à cette formation aujourd'hui ?

En quoi cette formation **est-elle importante** pour votre équipe ?

Futur

Quel est **votre objectif** avec ce cours ?

Comment utiliserez-vous ces nouvelles compétences ?

...et **d'ici 2 ans** ? 5 ans ?

Qui suis-je ?

2021 → aujourd'hui	**Auteur, conférencier et co-fondateur de CRYPTO-CHEMISTS** Formation et conseil sur l'impact des technologies emergentes (Blockchain, AI…)
2018 → aujourd'hui	**Directeur technique et co-fondateur de BOLDCODE** Développement et audit logiciel, web et mobile, offshoring éthique au Népal.
2017 → 2022	**Directeur technique et co-fondateur de DATA-TRANSITION** Gestion éthique des données, audit des SI, conformité au RGPD.
2010 → 2017	**Gérant et co-fondateur de NETCAT (GNUSIDE)** Infrastructures & systèmes en réseau, optimisation de la fiabilité, de la sécurité et de la performance.
2006 → 2010	**Ingénieur de recherche chez BEWAN (Pace Group)** Conception de systemes embarqués, et automatisation de la qualité logicielle.

Déroulement de la formation & règles du jeu

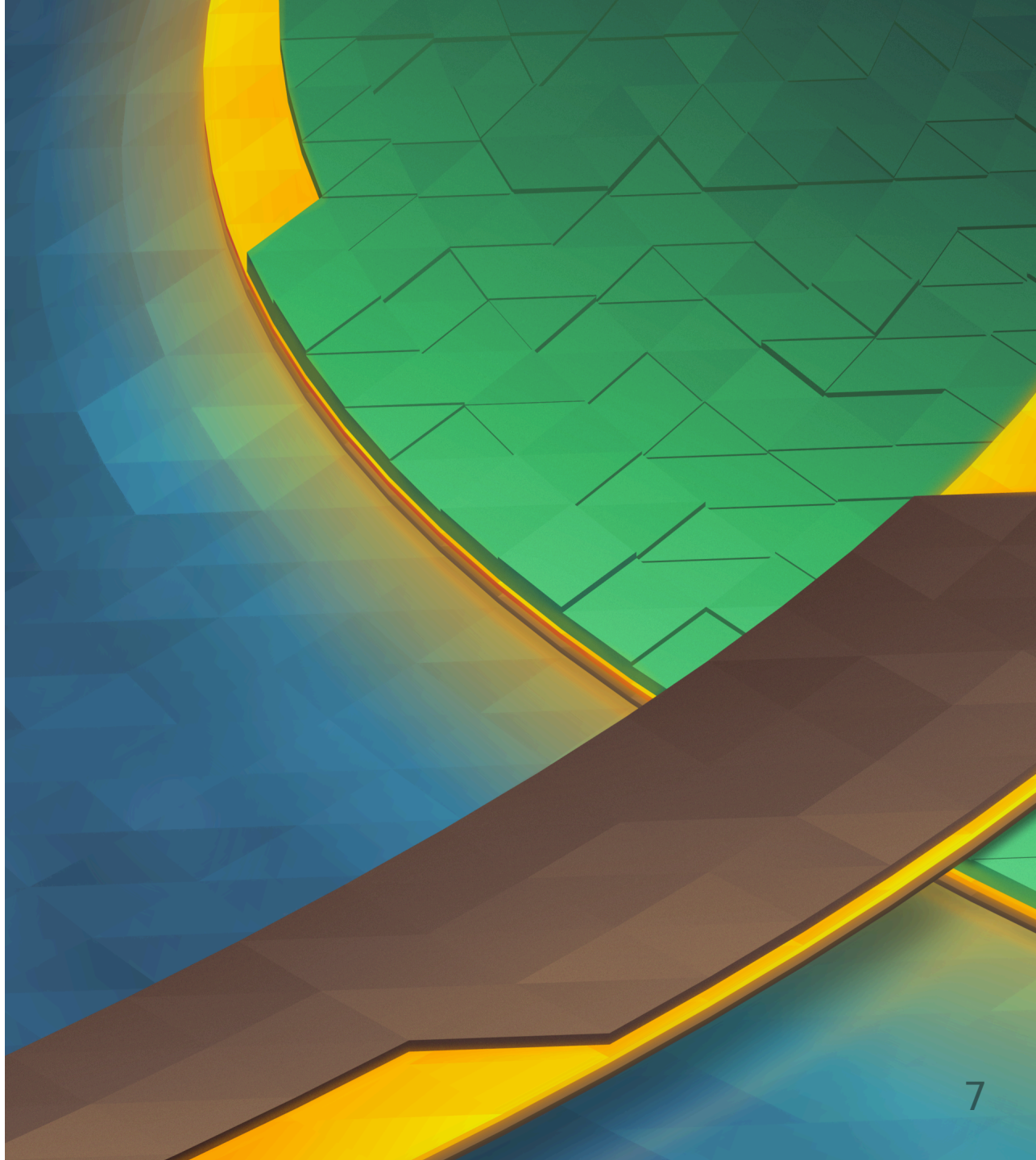
Horaires

- 9h00 - 12h30
- 13h30 - 17h00
- Des pauses le matin et l'après midi

Le cadre

- Liberté de parole dans le respect des autres et des objectifs de la formation
- Bienveillance, nous sommes dans un espace d’apprentissage
- Confidentialité de l’animateur et des participants sur les échanges

Gitlab



Rappels Git

Introduction à Git et son histoire

Brève Histoire de Git:

- Développé par **Linus Torvalds en 2005**
- Nécessité d'un système de **gestion de versions distribué, performant, et sécurisé** pour le développement du noyau Linux

Philosophie et Principes Clés:

- Rapidité, simplicité, support pour le développement non linéaire (milliers de branches parallèles)
- Système distribué, chaque utilisateur possède une copie complète du dépôt

Concepts clés et architecture de Git

Système de stockage

- Blobs, trees, et commits
- Des identifiants basés sur le SHA-1 du contenu.

Les trois états de Git

- Work directory
- Stage
- History
  - Remote tracking branch (locale)
  - Remote branch

Commandes Git de base

Configuration

```
git config
```

Initialisation

```
git init
```

```
git clone
```

Travail local

```
git add
```

```
git commit
```

Information

```
git status
```

```
git diff
```

```
git blame
```

Collaboration

```
git fetch
```

```
git pull
```

```
git push
```

Gestion des branches

```
git checkout
```

```
git branch
```

```
git merge
```

Gestion des snapshots

```
git rebase
```

```
git cherry-pick
```

```
git bissect
```

Workflows Git

…à base de fork de dépots (par défaut)

- Copie du repository
- Demande d'intégration des changements
  - `git send-email`
  - Merge/Pull request

…à base de branches

- Travail local ou distant
- Répartition du travail dans des branches distinctes
  - Pour la collaboration
  - Pour la gestion de la qualité
  - etc.
- Fusion locale
- Avec ou sans outils
  - ex: `git flow`
- Merge/Pull request d'une branche spécifique dans *upstream*

Workflow à base de branches

Workflow simple

- Cloner, Modifier, Commiter, et Pousser
- Utilisation de branches pour le développement parallèle

Stratégies de Branchement

- Feature Branch Workflow, Git Flow, Forking Workflow

Importance des Branches

- Séparation du développement, facilitation de la revue de code, et intégration continue

Références

- Git Documentation: send-email
- email + git = <3
- Roger Dudler : Git, petit guide

À propos de Gitlab

Qu’est ce que GitLab ?

- Logiciel libre
- Fournissant
  - un hébergement de dépôts Git
  - des outils pour gérer le dépôt
  - des outils pour gérer les projets et la collaboration
- Développé initialement en Ruby / Rails
  - Réécrit en partie en Go par la suite
  - Frontend en VueJS

De (très) nombreuses fonctionnalités

- Gestion du versioning de code
- Suivi de bugs et collaboration
  - Milestones
  - Kanban
- Documentation interne
- CI/CD
  - Construction automatique
  - Gestion de pipelines (flux d'execution)
  - Gestion des livrables
  - Gestion des environnements de livraison
- etc.

Historique du logiciel (1)

La vie de Gitlab

- 2011: **débuts** de GitLab
  - Créé par Dmitiy Zaporozhets, puis Veriy Sizov et Sytse Sijbrandij
  - Inspiré par Gitorious
- 2012: beta de *GitLab.com* et annonce sur HackerNews
- 2013: « I want to work on GitLab full time »
- 2014: GitLab Inc
- 2015: Incubation au Y Combinator
- 2016: Croissance
- 2020: The world's largest all-remote company
  - 1200 salariés dans 65 pays
- ...

Historique du logiciel (2)

À souligner

- premiere licorne (en partie) Ukrainienne valorisée à plus de 1 milliard de $ en 2018
- Utilisé par IBM, Sony, NASA, Alibaba, Oracle, le CERN, Boing, SpaceX, etc.

Les différentes distributions

Community Edition (CE)

Open-source, sous licence MIT

Fonctionnalités clés :

- Gestion de versions avec Git
- Suivi des issues
- Revue de code
- CI/CD intégrée

➡️ Pour les petites équipes, projets open-source, et toute organisation cherchant une solution robuste sans coût supplémentaire.

Enterprise Edition (EE)

Propriétaire, nécessite un abonnement.

Fonctionnalités supplémentaires :

- Support pour les approbations de merge request
- Authentification et intégration plus poussées (LDAP, Kerberos)
- Analyse avancée de code et sécurité
- Gestion de projets et portefeuilles
- Assistance et support premium

➡️ Pour les organisations nécessitant des fonctionnalités avancées, un support prioritaire, et une intégration dans des environnements complexes.

Les modalités de fonctionnement (1)

On premise (soit Gitlab CE, soit Gitlab EE)

Avantages

- Contrôle total sur l'instance GitLab, y compris la configuration, la sécurité et l'accès.
- Possibilité de personnalisation et d'intégration avec des systèmes internes.
- Aucune dépendance à une connexion internet externe pour l'accès interne.

Défis

- Assurer la maintenance, les mises à jour, les sauvegardes, la sécurité.
- Coûts de l'infrastructure physique ou virtuelle nécessaire.
- Compétences techniques pour l'installation et la gestion.

Les modalités de fonctionnement (1)

Gitlab.com (basé sur GitLab EE)

Avantages

- Accessibilité instantanée sans configuration de serveur.
- Maintenance, mises à jour et sauvegardes gérées par GitLab.
- Intégration facile avec d'autres services et extensions GitLab.

Limitations et considérations

- Dépendance à une connexion internet stable.
- Moins de contrôle sur la configuration système et les politiques de sécurité.
- Possibles limitations sur les ressources disponibles gratuitement.

GitLab vs GitHub

Similarités

- Outil « intégré »
- Navigation dans le code
- Gestion des tâches
- Gestion des livrables
- Sites statiques
- Intégration continue

GitLab vs GitHub

Différences

	Gitlab	Github
Licence et business-model	OpenSource (Open Core)	Propriétaire
Hébergement	On premise ou SaaS	SaaS
Permissions	Organisation, Groupes (sous-groupes, etc.), Équipes	Organisation, Equipe
Terminologie	Merge Request Snippet Project	Pull Request Gist Repository
Integration continue	Depuis le début	Fonctionalité tardive (d'abord externe, puis Github Actions)

Utilisateurs, rôles et projets

Premiers pas

- Création d'un compte utilisateur
- Création d'une clé SSH

```
ssh-keygen -t rsa -C moi@projet_rsa -f moi@projet_rsa
```

- Autorisation de la clef (publique) SSH sur Gitlab

Tour d'horizon des projets

- Page par défaut
  - Configurable selon le projet
- Groupes / Projets
- Activité
- Création
- Forks

Gestion des utilisateurs

- Gestion des équipes
- Invitation d’utilisateurs
- Les rôles
  - Définition des rôles
  - Droits associés aux roles

Fonctionalités par projet

- Dépot de code (repository)
  - Editeur de code intégré
  - Readme en markdown
- Gestion de tâches
- Intégration continue
- Registres de releases, packages, et de containers
- Sites statiques (pages)
- Wiki

Dépot de code (repository)

Navigateur de fichiers

- Changement de branches
- Création de fichiers
- Création de branches
- Recherche de fichiers
- Téléchargement du code
- Clone

Dépot de code (repository)

Historique des commits

- Changement de branches
- Recherche par auteur
- Recherche par message
- Navigation à la version X

Dépot de code (repository)

Branches

- Niveau d’activité
- Comparaison
- Demande de fusion
- Téléchargement
- Création / Suppression

Tags

- Listing
- Filtrage par nom
- Tri par date

Contributeurs

- Timeline - Activité globale
- Timeline - Activité par contributeur
- Vue par branche

Graphe

- Suivi des développements en parallele
- Suivi des fusions

Comparaison de versions

Des réglages spécifiques

Pour garantir un workflow

- Branche par défaut
- Branches protégées
- Tags protégés

Pour le déploiement

- Tokens de déploiement
- Clefs de déploiement

Et aussi

- Mirroir automatique d’un dépot
- Nettoyage automatique

🟦 Dans **Settings > Repository**

Gestion des taches

Tâches

- Création / Mise à jour / Suppression
- Filtres
  - Taches ouvertes / fermées
  - Par responsable...
  - Toutes les tâches
- Import / Export

Points d’étapes (milestones)

- Création
- Listing + Filtres (par date, etc.)

Labels et issue board

- Project Information > Labels

Références

Vidéos

- LinkedIn Learning: Josh Samuelson - Learning Gitlab
- LinkedIn Learning: Josh Samuelson - Continuous integration and continuous delivery with GitLab

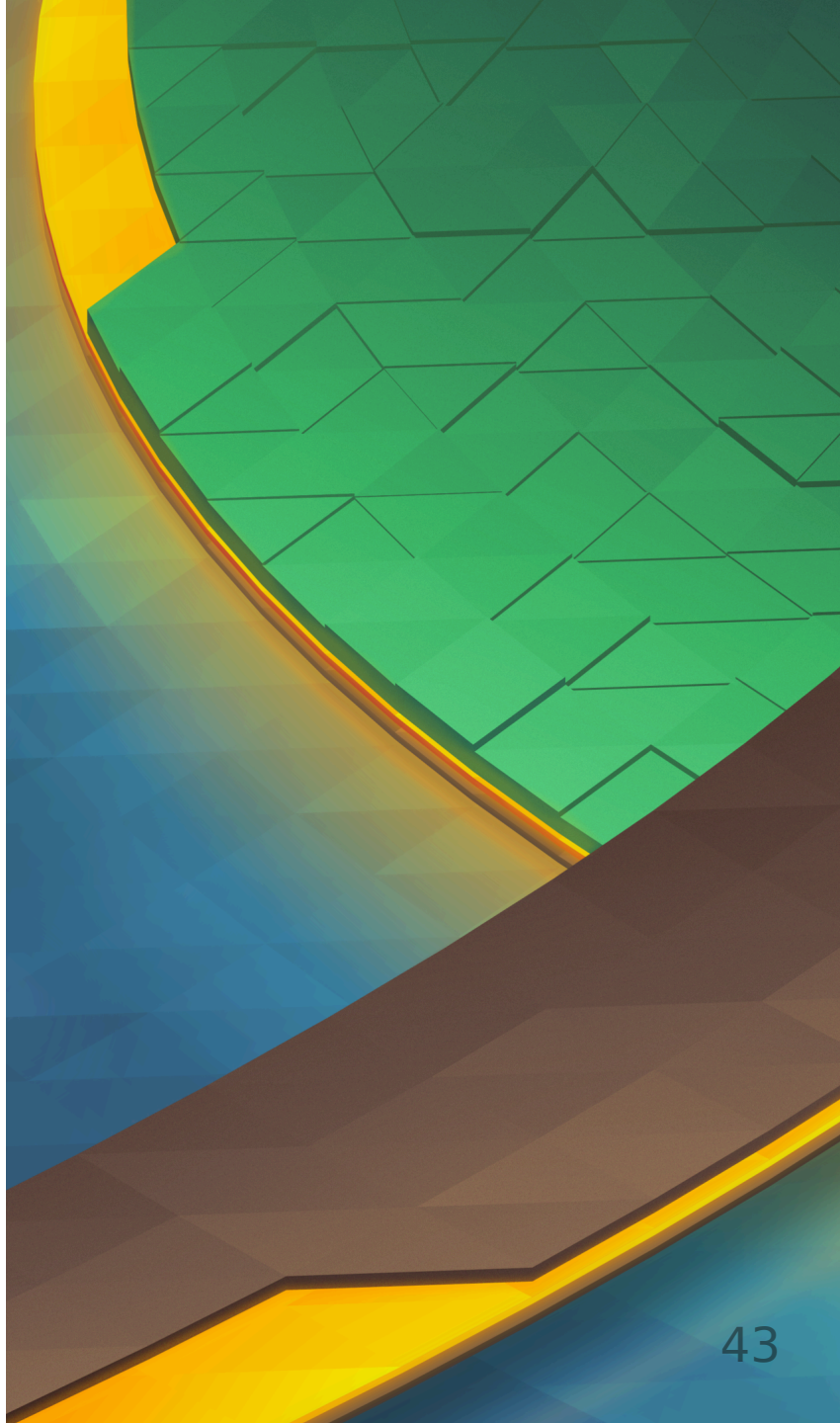
Cours

- GitLab: pour bien commencer
- Groupe IPSL GitLab IN2P3

Awesome Gitlab

- andorka/awesome-gitlab
- fkromer/awesome-gitlab

Gitlab-CI



Architecture de Gitlab-CI

Définition et concepts (1)

Pipeline

- Un flux passant à travers une série d'étapes (*Stage*)
- Ils sont déclenchés par `git push` par défaut
- Ils peuvent aussi être déclenchés en externe (HTTP POST), utile pour le CI multi-projets
- Ils peuvent être programmés à l'avance et de façon régulière (*cron-like*)

Stage

- Une étape du *Pipeline*
- Les *Stages* sont évalués de façon séquentielle
- Un *Stage* exécute des *Jobs*

Définition et concepts (2)

Job

- Le plus petit composant d'un *Pipeline*
- Les *Jobs* d'un même *Stage* sont exécutés en parallèle.
- Chaque *Job* contient une ou plusieurs commandes qui doivent être exécutées (séquentiellement)
- Il peut être paramétrée par des variables
- Il peut être limité ou contraint par règles (certains tags, certaines branches spécifiques, etc.)
- Il produit des fichiers « résultats »
- L’exécution des jobs est réalisée dans des *conteneurs* sur n’importe quel machine ou *Pod Kubernetes* disponible et enregistrés comme *GitLab Runner*

Définition et concepts (3)

Artifacts

- Définit les fichiers ou dossiers à conserver après l'exécution d'un job.
- Utile pour passer des binaires ou des rapports entre les stages.

Cache

- Utilisé pour définir les dossiers à mettre en cache entre les exécutions de jobs.
- Accélère les builds en réutilisant les données entre les jobs/stages.

Composants de Gitlab-CI (1)

Composants obligatoires

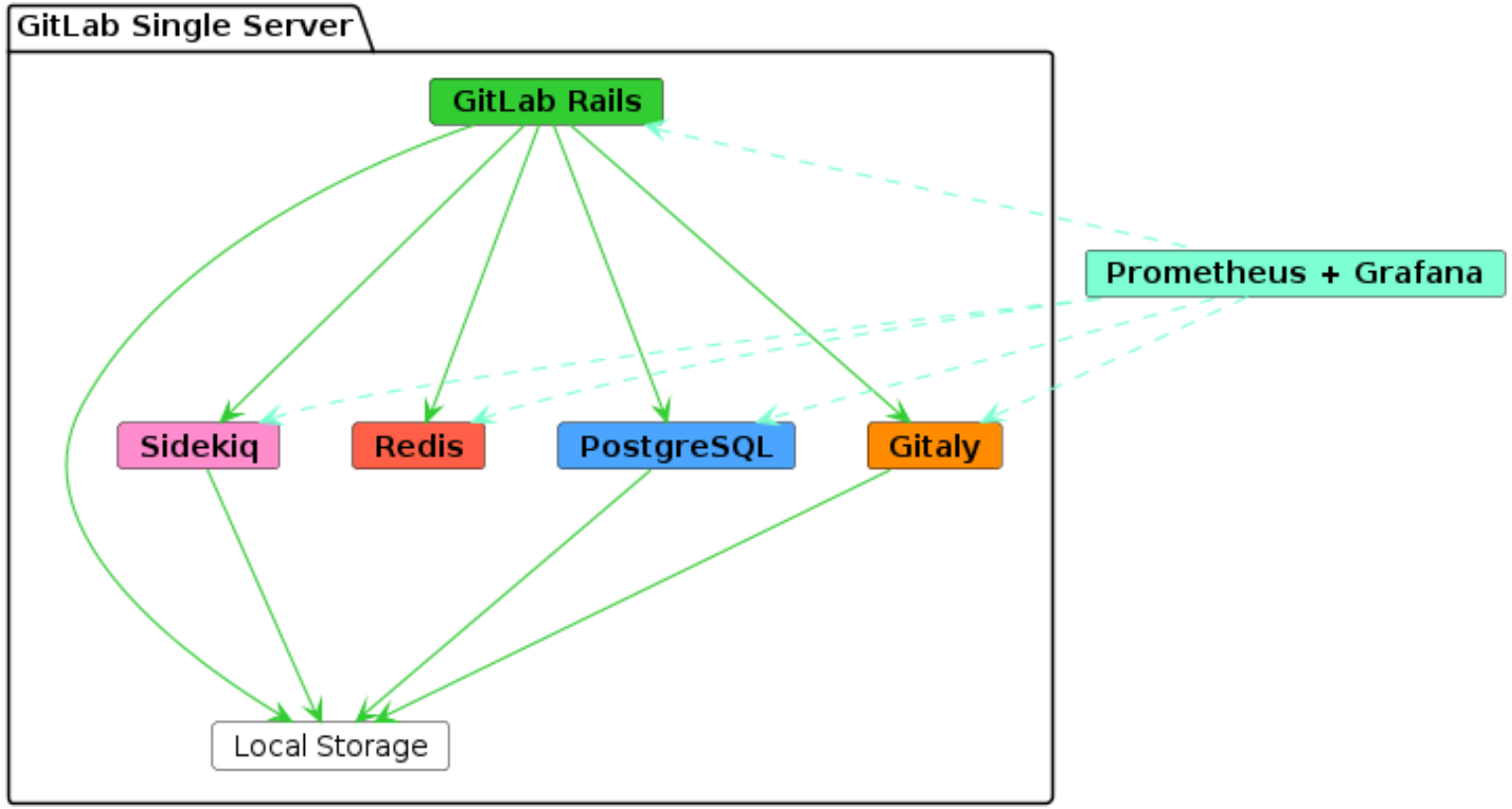
- GitLab Coordinator
- GitLab Runners
  - Runner Executors
- Composants tiers
  - Base de données (PostgreSQL)
  - Cache (Redis)
  - Gestion des dépôts Git (Gitaly)
  - Message Queue (Sidekiq)

Composants optionnels

- Composants tiers
  - Configuration (Consul)
  - Stockage Objet (S3)
  - Load-Balancer Internes (HAProxy)
  - Reverse-proxy PostgreSQL (PGBouncer)

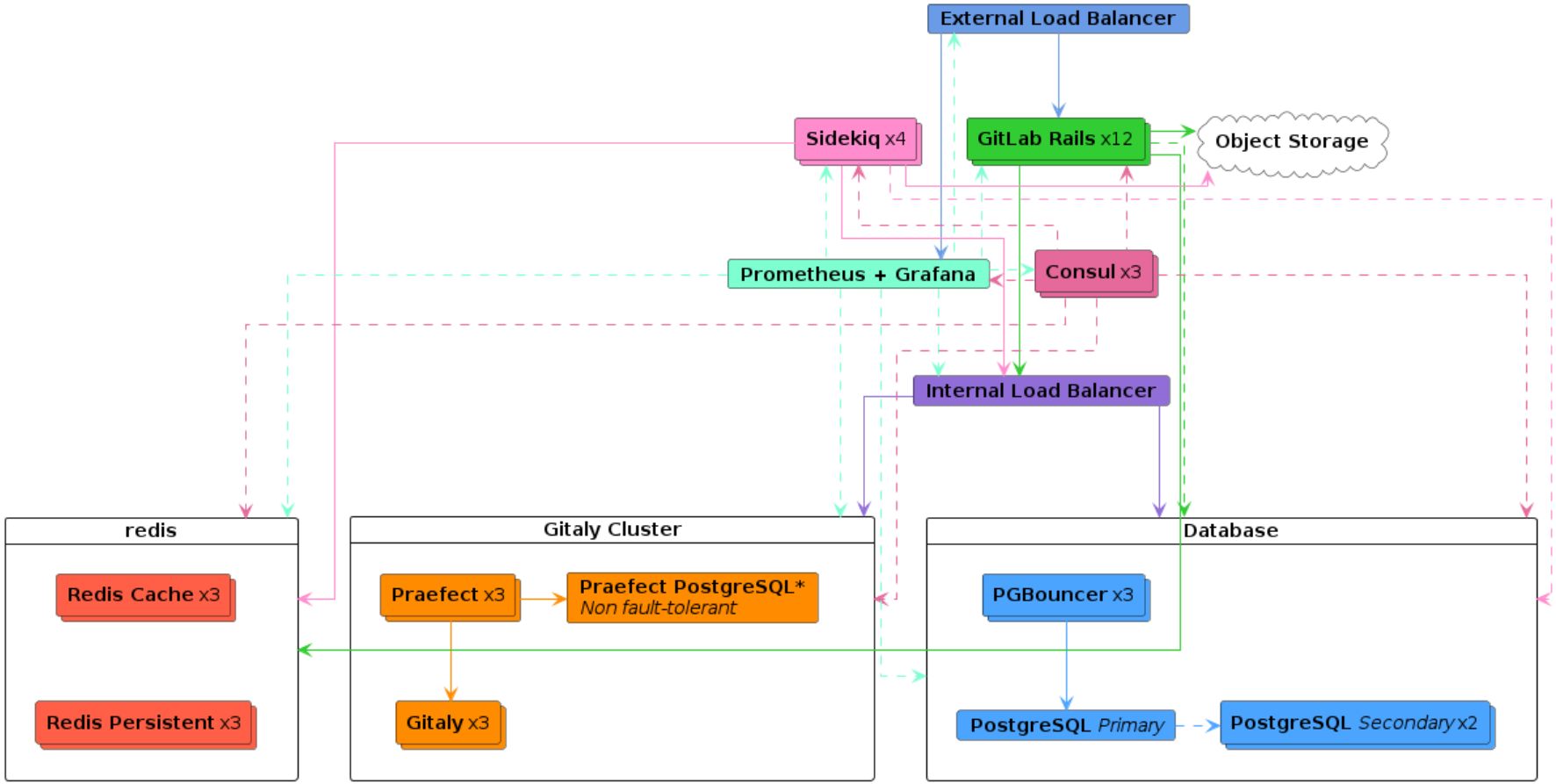
ℹ️ GitLab: Reference architecture

Composants de Gitlab-CI (2)



- Jusqu'à 20 RPS / 1.000 utilisateurs

Composants de Gitlab-CI (3)



- Jusqu'a 1000 RPS / 50.000 utilisateurs

GitLab Runner

- Agent léger et redimensionnable
- Récupère une tâche CI via l'API du coordinateur de GitLab CI/CD
- Exécute la tâche
- Renvoie le résultat à l'instance GitLab
- Il est très déconseillé de faire fonctionner les *Gitlab Runners* sur la même machine que le serveur

Rôle des *GitLab Runner*

- Exécution des jobs définis dans `.gitlab-ci.yml`.
- Isolation de l'environnement d'exécution pour chaque job.
- Flexibilité et adaptabilité élevées selon les besoins du projet.

Types de *GitLab Runners*

Shared Runners

- Disponibles pour tous les projets d'une instance GitLab.
- Hébergés et gérés par l'administrateur de GitLab.
- Avantages : flexibilité élevée.
- Inconvénients : sécurité (ressources partagées).

Specific Runners

- Associés à des projets spécifiques.
- Contrôle granulaire sur l'environnement d'exécution.
- Utilisés pour des besoins de sécurité ou configurations spécifiques.

Group Runners

- Associés à un groupe de projets.
- Disponibles pour tous les projets dans un groupe GitLab donné.

Configuration et gestion des GitLab Runners

Installation

- Varie selon le système d'exploitation.
- Peut nécessiter des scripts d'installation ou l'utilisation de Docker.

Enregistrement

- Nécessaire après l'installation.
- Implique la fourniture de l'URL de GitLab et d'un token d'enregistrement.

Surveillance et mise à jour

- Surveillance régulière de l'état et de la performance.
- Maintenance des runners avec les dernières versions pour sécurité et performance.

Runner Executor

- L'environnement dans lequel le travail sera exécuté par le *GitLab Runner*
- Permet la personnalisation de l'environnement d'exécution.
  - Flexibilité dans le choix de l'environnement adapté aux besoins spécifiques des projets.
- Chaque *GiLab Runner* définit au moins un exécuteur

Types de *Runner Executors*

Docker Executor

Utilise des conteneurs Docker pour l'exécution des jobs.

Favorise la reproductibilité et l'isolation des environnements.

Shell Executor

Exécute les jobs directement sur la machine hôte.

Utilisé pour des tâches simples ou quand un accès complet au système est nécessaire.

Kubernetes Executor

Exécute les jobs dans un cluster Kubernetes.

Permet l'élasticité et la scalabilité dans des environnements cloud ou microservices.

Autres Executors

Virtual Machine (VM) Executors, par exemple avec VirtualBox.

Permettent d'utiliser des machines virtuelles personnalisées pour les jobs.

Choix et configuration des *Runner Executors*

Critères de sélection

- Nature des tâches à exécuter (besoin d'isolation, accès aux ressources, etc.).
- Infrastructure existante (on-premise, cloud, Kubernetes, etc.).
- Exigences de sécurité et de performance.

Configuration

- Définie dans le fichier de configuration du GitLab Runner.
- Nécessite souvent des ajustements spécifiques au projet et à l'infrastructure.

Installation de Gitlab-CI

Installation de Gitlab

Ressources recommandées

- CPU : 4
- RAM : 4 GB
- du stockage en conséquence

ℹ️ Ressources minimales recommandées selon la documentation de Gitlab (2022-07)

Méthodes d'installation

GitLab peut être installé de différentes manières :

- à partir des sources: Cette méthode offre le plus de contrôle et de flexibilité, mais elle est plus complexe et nécessite des connaissances techniques approfondies.
- avec Docker: Docker simplifie l'installation et la configuration de GitLab, en encapsulant l'application et ses dépendances dans un conteneur.
- avec un package Omnibus: GitLab fournit des packages Omnibus pré-construits pour différentes distributions Linux, ce qui simplifie l'installation et la configuration.

Le choix de la méthode d'installation dépend des besoins et des préférences de chaque utilisateur.

Installation à partir des sources

L'installation à partir des sources est la méthode la plus flexible, mais elle nécessite des connaissances techniques approfondies.

Prérequis:

- Un serveur Linux avec les dépendances nécessaires installées (Ruby, Go, Node.js, PostgreSQL, Redis, etc.).
- Un utilisateur système dédié pour GitLab.
- Une base de données PostgreSQL configurée pour GitLab.

Installation à partir des sources

Étapes d'installation:

- Cloner le dépôt GitLab: git clone `https://gitlab.com/gitlab-org/gitlab.git`
- Installer les dépendances: `bundle install`
- Configurer GitLab: `cp config/gitlab.yml.example config/gitlab.yml` et modifier le fichier de configuration selon vos besoins.
- Initialiser la base de données: `bundle exec rake gitlab:setup RAILS_ENV=production`
- Démarrer GitLab: `bundle exec rails start`

Installation avec Docker

Docker simplifie l'installation et la configuration de GitLab en utilisant un conteneur.

Prérequis

- Docker installé sur votre système.
- Un fichier docker-compose.yml configuré pour GitLab.

Installation avec Docker

Étapes d'installation

- Créer un fichier `docker-compose.yml` avec la configuration fournie (page suivante)
- Démarrer GitLab avec Docker Compose: `docker-compose up -d --build`

ℹ️ Il est important de personnaliser le fichier `docker-compose.yml` en fonction de vos besoins, notamment en configurant les volumes de données, les ports, et les variables d'environnement.

Installation avec Docker

```yaml
version: "3.6"
services:
  gitlab:
    image: gitlab/gitlab-ce:latest
    ports:
      - "22:22"
      - "80:80"
      - "443:443"
    volumes:
      - /srv/gitlab/data:/var/opt/gitlab
      - /srv/gitlab/logs:/var/log/gitlab
      - /srv/gitlab/config:/etc/gitlab
    environment:
      GITLAB_OMNIBUS_CONFIG: "from_file('/omnibus_config.rb')"
    configs:
      - source: gitlab
        target: /omnibus_config.rb
    secrets:
      - gitlab_root_password
  gitlab-runner:
    image: gitlab/gitlab-runner:alpine
    # pour swarm
    deploy:
      mode: replicated
      replicas: 4
# pour swarm
configs:
  gitlab:
    file: ./gitlab.rb
secrets:
  gitlab_root_password:
    file: ./root_password.txt
```

Installation avec un package *Omnibus*

GitLab fournit des packages Omnibus pré-construits pour différentes distributions Linux. Cette méthode est la plus simple et la plus rapide.

Prérequis:

- Un serveur Linux compatible avec les packages Omnibus.
- Accès à Internet pour télécharger le package.

Installation avec un package *Omnibus*

Étapes d'installation:

- Télécharger le package Omnibus pour votre distribution Linux à partir de https://about.gitlab.com/install/.
- Installer le package en utilisant le gestionnaire de paquets de votre distribution (ex: `dpkg -i gitlab-ce_15.10.4-ce.0_amd64.deb` pour Debian/Ubuntu).
- Configurer GitLab en modifiant le fichier `/etc/gitlab/gitlab.rb`.
- Reconfigurer GitLab: gitlab-ctl reconfigure
- Démarrer GitLab: gitlab-ctl start

Configuration de GitLab

Après avoir installé GitLab, vous devez le configurer en fonction de vos besoins. Les options de configuration courantes incluent:

- **Nom de domaine:** Définissez le nom de domaine que GitLab utilisera pour accéder à l'interface web.
- **SSL/TLS:** Activez SSL/TLS pour sécuriser les connexions à GitLab.
- **Authentification:** Configurez les méthodes d'authentification des utilisateurs (ex: mot de passe, LDAP, OAuth).
- **Email:** Configurez les paramètres de messagerie pour les notifications et les rappels.
- **Sauvegardes:** Configurez les sauvegardes régulières de votre instance GitLab.

ℹ️ La configuration de GitLab se fait généralement en modifiant le fichier de configuration principal (`/etc/gitlab/gitlab.rb` pour les installations Omnibus, `config/gitlab.yml` pour les installations à partir des sources) et en relançant GitLab.

Installation du registre Docker (Container Registry)

GitLab inclut un registre Docker intégré (Container Registry) pour stocker les images Docker.

Prérequis

- GitLab installé et configuré.
- Un volume de stockage dédié pour le registre Docker.

Installation du registre Docker (Container Registry)

Étapes d'installation

- Activer le registre Docker dans la configuration de GitLab:
  - Pour Omnibus, modifier `/etc/gitlab/gitlab.rb` et ajouter `registry_external_url 'https://registry.example.com'` (en remplaçant `registry.example.com` par votre nom de domaine).
  - Pour les installations à partir des sources, modifier `config/gitlab.yml` et ajouter `registry_external_url: "https://registry.example.com"`.
- Reconfigurer GitLab: `gitlab-ctl reconfigure` (pour Omnibus).
- Démarrer le registre Docker: `gitlab-ctl start registry` (pour Omnibus).

ℹ️ Le registre Docker est accessible via l'interface web de GitLab, dans le menu "Packages & Registries".

Références

- Gitlab: Install gitlab
- Gitlab: Update gitlab
- GitLab Docker images

Intégration continue (*continuous integration*)

Comment construire un pipeline ?

- Un cycle d’intégration continue dans Gitlab CI est défini à partir d’un fichier de configuration écrit en YAML.
- Le fichier est placé à la racine du projet sous le nom réservé de `.gitlab-ci.yml`.

Définition d'un pipeline dans `gitlab-ci.yml`

Exemple :

```
---

# ... pipeline n°1 ....

---

# ... pipeline n°2 ....
```

Définition et rôle des différents *stages* dans le *pipeline*

- L'ordre et les noms des *stages* sont arbitraires et fonction du projet.

Exemple :

```
# ...

stages:
  - build
  - test
  - deliver
  - deploy

# ...
```

Image Docker dans laquelle s'exécutent les scripts

- Peut être globale (par défaut pour tous les jobs), ou spécifique à un job

Exemple :

```
# ...

image: node:latest

job1:
  # ici on définit une image spécifique au job
  image: crystal:1.12.0
  scripts:
    - shards install
    - shards build

job2:
  # ici l'image globale est utilisée
  scripts:
    - npm install
    - npm run build
# ...
```

Structure de base des *Jobs*

- Un job doit indiquer à quel *stage* il appartient
- Un job contient (au moins) un *script*

Exemple :

```
# ...

deliver:dockerimage:
  stage: deliver
  # ...
  script:
    - export VERSION="$(cat ./_artifacts/VERSION)"
    - echo "Delivering docker image for $CI_REGISTRY_PROJECT:$VERSION"

# ...
```

Introduction aux services dans GitLab CI/CD

Définition

- Services Docker qui fonctionnent aux côtés du job dans un pipeline
- Utilisés pour fournir des applications ou des bases de données nécessaires aux tests

Utilité

- Simuler des environnements de production
- Fournir des dépendances externes nécessaires au cours des tests

Exemples de services couramment utilisés

Bases de données

PostgreSQL, MySQL, Redis, ElasticSearch...

```
services:
  - postgres:latest
  - mysql:5.7
```

Docker (in Docker)

Docker in Docker

```
image: docker:20.10
services:
  - docker:dind
```

Configuration des services dans `.gitlab-ci.yml` (1)

Syntaxe de base

- Définir les services sous le mot-clé `services` dans `.gitlab-ci.yml`

```yaml
test:
  services:
    - redis:latest
```

Accès aux Services

- Les services sont accessibles via `localhost` depuis le job principal
- Les ports standards sont utilisés pour se connecter aux services

Configuration des services dans `.gitlab-ci.yml` (2)

Variables d'environnement pour les services

Passer des variables d'environnement aux services

```
services:
  - name: mysql:5.7
    command: ["--default-authentication-plugin=mysql_native_password"]
    variables:
      MYSQL_DATABASE: "example"
      MYSQL_ROOT_PASSWORD: "password"
```

Configuration des services dans `.gitlab-ci.yml` (3)

Alias pour les services

Utiliser `alias` pour référencer les services sous un nom différent

```yaml
services:
  - name: mysql:5.7
    alias: db57
  - name: mysql:5.6
    alias: db56
```

Bonnes pratiques pour l'utilisation des services

- **Choisir des Versions Spécifiques**
  - Utiliser des tags de version spécifiques pour éviter les surprises dues aux mises à jour
- **Nettoyage des Ressources**
  - Assurer la fermeture et le nettoyage des connexions aux services pour éviter les fuites de ressources
- **Sécurité**
  - Être conscient des implications de sécurité lorsque vous exposez des ports ou utilisez des services en réseau

Artefacts

> « (nom mascular, du latin *artis facta*) : en anthropologie, produit ayant subi une transformation, même minime, par l'homme, et qui se distingue ainsi d'un autre provoqué par un phénomène naturel. » Larousse (2024)

- Fichiers générés par les jobs dans Gitlab CI, que l'on souhaite conserver.
- Peuvent posséder une durée d'expiration pour éviter de surcharger le stockage avec des fichiers anciens.

```yaml
job24:
  # ...
  artifacts:
    paths:
      - ./_artifacts
    expire_in: 1 hour
```

ℹ️ **Note :** utiliser des chemins spécifiques pour les artifacts pour ne pas conserver plus que nécessaire.

⚠️ **Attention :** En anglais (dans la configuration en YAML), on écrit « artifacts »

Cache

- Permet de réutiliser entre les jobs ou les pipelines les données qui ne changent pas souvent
- L'utilisation du cache permet d'accélérer les builds.

```yaml
job24:
  # ...
  cache:
    paths:
      - ./node_modules
    key: ${CI_COMMIT_REF_SLUG}
```

⚠️ **Attention**:

- Prudence avec le cache cross-project pour éviter les conflits ou la pollution du cache.
- Utiliser des clés de cache spécifiques à la branche, au commit ou au job pour optimiser la réutilisation du cache.
- Cachez les dépendances plutôt que les résultats de build pour éviter de cacher des données incorrectes.

Syntaxe et mots-clés

- `script`
- `before_script` et `after_script`
- `image`
- `stages`
- `only` et `except`
- `only` avec `schedules`
- `when`
- `allow_failure`
- `tags`
- `services`
- `environment`
- `variables`
- `caches`
- `artifacts`
- `retry`

Le fichier .gitlab-ci.yml

- Exemple avec gitbook
- Exemple avec Jekyll
- Exemple avec MkDocs
- Exemple avec Docker

Contrôle des Jobs

Dépendances entre les *jobs*

Définir l'ordre d'exécution

Le mot-clé `needs` permet de spécifier les *jobs* dont un *job* dépend.

```yaml
build:
  stage: build
  script:
    - echo "Construire l'application"

test:
  stage: test
  script:
    - echo "Exécuter les tests"
  needs:
    - build
```

Dans cet exemple, le *job* test ne s'exécutera qu'après la réussite du *job* build.

Dépendances entre les *jobs*

Optimiser les performances

- Exécuter des *jobs* en parallèle lorsque cela est possible pour accélérer le pipeline.
- Définir les dépendances permet d'éviter d'attendre inutilement la fin de *jobs* non-essentiels.

`needs:` vs `dependencies:`

- `dependencies:` est l'ancienne syntaxe, dépréciée mais toujours fonctionnelle.
- `needs:` est la nouvelle syntaxe recommandée, offrant plus de flexibilité et de contrôle.

Exécution conditionnelle des *jobs*

En fonction de branches spécifiques

- Le mot-clé `rules` permet de définir des règles d'exécution pour les *jobs*, basées sur des conditions spécifiques.
- GitLab CI/CD fournit des variables prédéfinies pour accéder aux informations du contexte d'exécution (ex: `$CI_COMMIT_BRANCH`).

```yaml
deploy_to_prod:
  stage: deploy
  script:
    - echo "Déploiement en production"
  rules:
    - if: $CI_COMMIT_BRANCH == "master"
      when: always
```

Exécution conditionnelle des *jobs*

En fonction de branches spécifiques

- Il est possible d'utiliser des expressions régulières pour définir des règles plus complexes.

```yaml
deploy_to_staging:
  stage: deploy
  script:
    - echo "Déploiement en staging"
  rules:
    - if: $CI_COMMIT_BRANCH =~ /^release\/.*/
      when: always
```

- Ce *job* s'exécute pour toutes les branches commençant par `release/`.

ℹ️ **Note:** Les règles `rules` remplacent les anciens mots-clés `only` et `excepta`.

Exécution conditionnelle des *jobs*

En fonction de tags spécifiques

```yaml
release:
  stage: release
  script:
    - echo "Créer une nouvelle release"
  rules:
    - if: $CI_COMMIT_TAG
      when: always
```

Ce job `release` ne s'exécute que lorsqu'un *tag* est créé.

ℹ️ **Note:** `$CI_COMMIT_TAG` est une variable prédéfinie qui contient le nom du *tag* associé au *commit*.

Exécution conditionnelle des *jobs*

En fonction de tags spécifiques

```yaml
deploy_production:
  stage: deploy
  script:
    - echo "Déployer en production"
  rules:
    - if: $CI_COMMIT_TAG =~ /^v[0-9]+\.[0-9]+\.[0-9]+$/
      when: always
```

Ce job `deploy_production` ne se déclenche que pour les *tags* respectant le format `vX.Y.Z`.

- Permet de déclencher des actions spécifiques pour des versions taguées de votre code.
- Assure un contrôle précis sur les déploiements en production.
- Facilite la gestion des versions et des releases.

Exécution conditionnelle des *jobs*

En fonction des modifications dans les fichiers

- Évite d'exécuter des *jobs* inutilement si les fichiers concernés n'ont pas été modifiés.
- Accélère le pipeline en limitant les *jobs* aux tâches essentielles.

```
build_frontend:
  stage: build
  script:
    - echo "Construire l'interface utilisateur"
  rules:
    - if: $CI_COMMIT_BRANCH == "main" && $CI_PIPELINE_SOURCE == "push"
      changes:
        - frontend/**/*
      when: on_success
```

Ce job `build_frontend` ne s'exécute que si des fichiers dans le dossier `frontend/` ont été modifiés sur la branche `main` lors d'un *push*.

Exécution conditionnelle des *jobs*

En fonction des modifications dans les fichiers

- `changes:` permet de spécifier les fichiers ou dossiers à surveiller.
- `when:` permet de définir le moment d'exécution du job en fonction des changements.

ℹ️ **Note:** L'utilisation de changes: nécessite de comprendre la structure de votre projet et de définir des règles précises pour un fonctionnement optimal.

Exemple de cas d'usages

- Déclencher la construction d'une documentation seulement si les fichiers sources ont changé.
- Exécuter des tests spécifiques uniquement pour les parties du code impactées par les modifications.

Bonnes pratiques pour le Job Control

- L'utilisation excessive de dépendances peut complexifier le pipeline et le rendre difficile à maintenir.
- Privilégiez la simplicité et la clarté.

Job Control : Références

- Gitlab CI: Migrer depuis les only / exept vers les rules
- Gitlab CI: Lancer un job seulement sur des tags spécifiques
- Gitlab CI: Lancer une job seulement quand certains fichiers sont modifiés
- Job control

Éviter la duplication de code

Les mots-clés `before_script` permettent de factoriser une séquence de commandes

```yaml
image: ubuntu:latest

before_script:
  - apt-get update
  - apt-get install -y gcc g++
  - mkdir _build

phase1:
  script:
    - gcc -o _build/toto toto.c

phase2:
  script:
    - gcc -o _build/titi titi.c
```

Éviter la duplication de code

Utiliser des ancres et des références YAML

YAML permet de définir des ancres et des références pour réutiliser des séquences de commandes ou des configurations complexes.

```yaml
.build_config: &build_config
  image: ubuntu:latest
  before_script:
    - apt-get update && apt-get install -y gcc g++
    - mkdir _build

phase1:
  <<: *build_config
  script:
    - gcc -o _build/toto toto.c

phase2:
  <<: *build_config
  script:
    - gcc -o _build/titi titi.c
```

Dans cet exemple, l'ancre `&build_config` définit une configuration de *build* commune.

Les *jobs* phase1 et phase2 utilisent ensuite la référence `<<: *build_config` pour hériter de cette configuration.

Éviter la duplication de code

Étendre les images docker

- Créer une image Docker de base contenant les dépendances communes à plusieurs projets
- Publier cette image sur le *Container Registry* de Gitlab
- Utiliser cette image de base dans la définition de image: dans `.gitlab-ci.yml`
- Ajouter les dépendances spécifiques au projet dans chaque *job*

```yaml
image: registry.gitlab.com/mon-organisation/mon-image-de-base:latest

job1:
  # ...
  before_script:
    - apt-get install -y mon-dependance

job2:
  # ...
  before_script:
    - npm install mon-autre-dependance
```

Éviter la duplication de code

Utiliser des outils comme `make` ou `maven`

- Définir des cibles (= tâches) avec un script prédéfini, mais indépendant du CI/CD
- Réutiliser ces cibles dans le CI/CD

```yaml
image: ubuntu:latest

before_script:
  - apt-get update
  - apt-get install -y make

build:
  script:
    - make build

test:
  script:
    - make test
```

Autres sujets intéressants

- Accélérer le pipeline
- Optimisation du cache
- Executer un job en local
- Mise en place d'un GitLab Runner
- Images Docker avec Gitlab-CI

Livraison continue (*continuous delivery*)

Introduction aux registres GitLab

Container Registry

Stockage sécurisé pour les images Docker

Intégration avec Docker et GitLab CI/CD

Package Registry

Gestion centralisée des paquets pour divers langages et outils (NPM, Maven, Conan, etc.)

Simplification de la publication et du partage de paquets au sein de projets GitLab

Utilisation de la Container Registry dans GitLab CI/CD

- **Configuration**
  - Activation du registre de conteneurs dans les paramètres du projet
  - Authentification Docker pour push/pull
- **Intégration CI/CD**
  - Construction d'images Docker dans les jobs CI et push vers le registre
  - Utilisation d'images du registre comme environnement d'exécution des jobs
- **Avantages**
  - Flux de travail automatisé pour la création et le déploiement d'images
  - Sécurité renforcée avec des contrôles d'accès et des scans de vulnérabilités

Utilisation du Package Registry dans GitLab CI/CD

- **Configuration**
  - Définition de variables CI pour l'authentification au registre de paquets
  - Configuration des fichiers de projet (par exemple, `.npmrc`, `pom.xml`) pour utiliser le registre
- **Intégration CI/CD**
  - Publication de paquets depuis les jobs CI
  - Installation de paquets dans les jobs CI ou les environnements de déploiement
- **Avantages**
  - Gestion simplifiée des dépendances au sein de l'écosystème GitLab
  - Visibilité et contrôle sur l'utilisation des paquets dans les projets

Gestion des versions et contrôles d'accès

- **Gestion des versions**
  - Historique complet des versions pour chaque paquet ou image
  - Facilité de rollback en cas de nécessité
- **Contrôles d'accès**
  - Configuration des permissions au niveau du projet ou du groupe
  - Sécurité améliorée par la restriction de l'accès aux ressources sensibles

Déploiement continu (*continuous deployment*)

Gestion des secrets

Introduction

- **Secrets:** Informations sensibles (mots de passe, clés API, jetons) à ne pas stocker dans le code source.
- **GitLab CI/CD:** Fournit des mécanismes pour gérer ces secrets de manière sécurisée et les rendre accessibles aux jobs de votre pipeline.

Gestion des secrets

Variables CI/CD

- **Fonction:** Stockage et gestion des secrets.
- **Configuration:** Définition au niveau projet/groupe, marquage comme "masquées" pour la protection.
- **Utilisation:** Dans les scripts CI/CD via `${variable}`.

Sécurité des Variables

- **Confidentialité:** Non affichées dans les logs de *job*.
- **Contrôle d'accès:** Accès restreint en fonction des rôles et des environnements.
- **Protection:** GitLab chiffre les variables stockées dans sa base de données.

Gestion des secrets

Bonnes Pratiques

- **Stockage:** Évitez de stocker les secrets directement dans `.gitlab-ci.yml`.
- **Masquage:** Utilisez des variables CI/CD masquées pour les informations sensibles.
- **Restriction d'accès:** Limitez l'accès aux variables en fonction des besoins.
- **Solutions externes:** Envisagez l'utilisation d'un gestionnaire de secrets externe (ex: HashiCorp Vault) pour une sécurité renforcée.

Automatiser les déploiements (bonnes pratiques)

Introduction

- L'automatisation des déploiements est un élément crucial du CI/CD, permettant d'accélérer la livraison de logiciels tout en minimisant les erreurs humaines.
- Cette section présente les bonnes pratiques pour automatiser efficacement les déploiements avec GitLab CI/CD.

Automatiser les déploiements (bonnes pratiques)

Utilisation des Scripts de Déploiement

- Définir des scripts clairs et reproductibles pour chaque étape du déploiement.
- Utiliser des outils d'automatisation comme Ansible, Chef, Puppet, ou des scripts shell.
- Versionner les scripts de déploiement dans Git pour un suivi des modifications.

```
deploy_to_prod:
  stage: deploy
  script:
    - ansible-playbook -i inventory.ini deploy.yml
```

Automatiser les déploiements (bonnes pratiques)

Gestion des Environnements

- Utiliser les environnements GitLab pour séparer les déploiements de développement, de test et de production.
- Configurer des variables d'environnement spécifiques pour chaque environnement.
- Limiter les déploiements automatiques à des environnements spécifiques en fonction des branches ou des tags.

```
deploy_to_staging:
  stage: deploy
  environment:
    name: staging
  only:
    - develop
```

Automatiser les déploiements (bonnes pratiques)

Intégration avec les Outils d'Infrastructure

- Intégrer GitLab CI/CD avec des plateformes cloud comme AWS, Azure, ou GCP pour automatiser la création et la configuration d'infrastructures.
- Utiliser des outils d'infrastructure as code (IaC) comme Terraform ou CloudFormation pour gérer les ressources cloud.

```
terraform_apply:
  stage: infra
  script:
    - terraform apply
```

Automatiser les déploiements (bonnes pratiques)

Surveillance et Journalisation

- Mettre en place une surveillance continue des déploiements pour détecter les erreurs ou les problèmes de performance.
- Collecter et analyser les journaux pour identifier les causes des erreurs et améliorer les processus de déploiement.

```
monitor_deployment:
  stage: post-deploy
  script:
    - ./scripts/monitor_application.sh
```

Automatiser les déploiements (bonnes pratiques)

Tests Automatisés

- Intégrer des tests automatisés à chaque étape du déploiement pour garantir la qualité et la stabilité des applications.
- Exécuter des tests d'intégration, de performance et de sécurité pour valider les déploiements.

```
integration_tests:
  stage: test
  script:
    - ./scripts/run_integration_tests.sh
```

Automatiser les déploiements (bonnes pratiques)

Rollback et Récupération

- Prévoir des mécanismes de rollback pour revenir à une version précédente en cas de problème.
- Automatiser le processus de rollback pour une récupération rapide.
- Définir des stratégies de récupération en cas d'échec de déploiement.

```
rollback_deployment:
  stage: manual
  when: manual
  script:
    - ./scripts/rollback_deployment.sh
```

Automatiser les déploiements (bonnes pratiques)

Sécurité

- Sécuriser les scripts de déploiement et les secrets utilisés pour l'accès aux environnements.
- Mettre en place des contrôles d'accès stricts pour limiter les autorisations de déploiement.
- Intégrer des outils de sécurité pour analyser les vulnérabilités et les failles de sécurité.

```
security_scan:
  stage: security
  script:
    - ./scripts/run_security_scan.sh
```

Automatiser les déploiements (bonnes pratiques)

Documentation et Communication

- Documenter clairement les processus de déploiement et les configurations.
- Mettre en place des canaux de communication efficaces pour informer les équipes des déploiements et des problèmes éventuels.

Environnements de déploiement

- Les environnements dans GitLab CI/CD permettent de gérer les différents espaces où votre code sera déployé, comme le développement (dev), le test (staging), et la production (prod).
- Utiliser des environnements permet de contrôler et de suivre le déploiement de vos applications dans ces différents contextes.

Définition et Configuration des Environnements

Déclaration des Environnements

- Dans votre fichier .gitlab-ci.yml, vous pouvez définir des environnements en utilisant le mot-clé environment.
- Chaque environnement peut avoir des configurations spécifiques, telles que des URLs uniques pour l'accès à l'application déployée.

```
deploy_to_dev:
  stage: deploy
  script: echo "Déploiement en environnement de développement"
  environment:
    name: dev
    url: https://dev.example.com
```

Utilisation des Environnements

- Les environnements peuvent être utilisés pour déclencher des déploiements conditionnels, en fonction de la branche ou des tags.
- Par exemple, vous pouvez décider de déployer uniquement sur l'environnement de production lorsque des tags sont poussés.

```
deploy_to_prod:
  stage: deploy
  script: echo "Déploiement en environnement de production"
  environment:
    name: prod
    url: https://example.com
  only:
    - tags
```

Visualisation et Suivi des Déploiements

- GitLab fournit une interface utilisateur pour visualiser les déploiements dans chaque environnement.
- Prmet de suivre facilement l'état actuel de votre application dans les différents environnements.
- Accès à cette vue via l'onglet Environments dans le projet GitLab.

Stratégies de Déploiement

Déploiement Continu

- GitLab CI/CD supporte le déploiement continu, vous permettant de déployer automatiquement votre code dans un environnement après le passage réussi des tests.
- Cela assure que votre code est toujours dans un état déployable.

Rollback

- En cas de problème avec un déploiement, GitLab permet de revenir facilement à une version précédente de votre application.
- Pour cela, vous pouvez utiliser la fonctionnalité de rollback disponible pour chaque environnement.

Feature flags

Introduction

- Technique pour gérer le déploiement de nouvelles fonctionnalités.
- Implémentation de feature flags (interrupteurs) dans le code source.
- Activation/désactivation de fonctionnalités à la demande, sans modifier le code source.

Implémentation des Feature Flags

- De nombreuses bibliothèques open-source sont disponibles pour faciliter l'implémentation des feature flags (ex: Unleash, FeatureFlags.io).

ℹ️ https://docs.gitlab.com/ee/operations/feature_flags.html

Feature flags

Fonctionnement

- Activation/désactivation via l'interface web de Gitlab (Deploy > Feature Flags).
- Contrôle précis : Activation/désactivation par environnement (dev, staging, prod).
- Ciblage d'utilisateurs : Création de listes d'utilisateurs pour des tests A/B.
- Gestion centralisée : Interface unique pour gérer tous les feature flags du projet.

Avantages des Feature Flags

- Déploiements progressifs: Introduction progressive à des groupes d'utilisateurs.
- Rollback facile: Désactivation rapide en cas de problème.
- Tests A/B: Comparaison de différentes versions d'une fonctionnalité.
- Contrôle granulaire: Ciblage précis des utilisateurs.

Bonnes pratiques

- Nommage clair: Utiliser des noms descriptifs pour les feature flags.
- Documentation: Documenter la raison d'être et le fonctionnement de chaque feature flag.
- Suppression progressive: Supprimer les feature flags obsolètes pour éviter la dette technique.
- Surveillance: Surveiller l'utilisation des feature flags et leur impact sur l'application.

Releases

Introduction

- Les Releases dans GitLab permettent de créer des points de repère immuables dans l'historique de votre projet.
- Elles marquent un ensemble spécifique de modifications
- Elle offrent une vue d'ensemble claire des livrables à un instant T.

Releases

Fonctionnalités Clés

- Versions Immuables: Une fois créées, les Releases ne peuvent pas être modifiées, garantissant l'intégrité des versions publiées.
- Association aux Tags: Chaque Release est associée à un tag Git, simplifiant la navigation et la gestion des versions.
- Informations Détaillées: Les Releases peuvent inclure:
  - Notes de Version: Décrivent les changements importants inclus dans la Release.
  - Liens vers des Artefacts: Permettent de télécharger les fichiers associés à la Release (ex: binaires, packages).
  - Informations de Déploiement: Indiquent où et comment la Release a été déployée.

Utilisation des Releases avec GitLab CI/CD

GitLab CI/CD permet d'automatiser la création et la gestion des Releases. Voici quelques cas d'usage courants:

- Automatisation: Créer des Releases automatiquement à chaque nouveau tag Git.
- Déclenchement: Utiliser la création d'une Release pour déclencher automatiquement des actions, comme un déploiement vers un environnement spécifique.
- Traçabilité: Suivre l'historique des Releases pour comprendre l'évolution du projet et identifier les versions déployées.

Créer des Releases Automatiquement

L'outil release-cli de GitLab permet de créer des Releases automatiquement dans votre pipeline CI/CD.

```
release:
  stage: release
  image: registry.gitlab.com/gitlab-org/release-cli:latest
  script:
    - release-cli create --name "Release $CI_COMMIT_TAG" --tag $CI_COMMIT_TAG
  only:
    - tags
```

Ce script s'exécute uniquement lorsqu'un nouveau tag est poussé vers le dépôt. Il crée une Release nommée "Release [nom du tag]" et l'associe au tag correspondant.

Avantages des Releases

- Communication Améliorée: Facilite la communication des changements et des versions aux équipes et aux utilisateurs.
- Gestion Simplifiée: Centralise la gestion des versions et des livrables.
- Traçabilité Accrue: Offre un historique clair des modifications et des déploiements.

Bonnes Pratiques

- Nommage Sémantique: Utilisez des noms de version significatifs (ex: v1.0.0) en suivant un schéma de versionnement (ex: SemVer).
- Notes de Version Détaillées: Fournissez des informations claires et concises sur les changements inclus dans chaque Release, en utilisant un format standardisé (ex: Markdown).
- Automatisation: Intégrez la création de Releases dans votre pipeline CI/CD pour un processus de release plus efficace et fiable.

Gitlab Pages

- Permet de publier des sites web statiques
- Directement à partir d'un dépôt dans GitLab
- Un *Job* spécial nommé `pages` génère tous les `artifacts` d'un site web dans le dossier spécial `public/` .
- Example avec GitBook

Autres fonctionnalités CI/CD

Introduction à Auto DevOps

- **Qu'est-ce que Auto DevOps ?**
  - Automatisation de l'ensemble du flux de travail CI/CD
  - Utilisation des meilleures pratiques et des outils intégrés
- **Avantages d'Auto DevOps**
  - Simplification de la configuration CI/CD
  - Amélioration de la qualité du code et de la sécurité
  - Accélération du temps de mise sur le marché

📘 Gitlab Documentation: Auto DevOps

Configuration d'Auto DevOps dans GitLab

- **Activer Auto DevOps**
  - Niveau projet et groupe
  - Configuration globale dans l'administration de GitLab
- **Prérequis**
  - Variables d'environnement nécessaires
  - Configuration du cluster Kubernetes pour certains stages

Composants et flux de travail d'Auto DevOps

- **Étapes clés**
  - Analyse de code, Test, Sécurité, Déploiement
- **Composants intégrés**
  - Analyse statique du code, Tests de vulnérabilité, DAST, Monitoring
- **Flux de travail**
  - Déclenchement automatique à chaque commit
  - Étapes conditionnelles basées sur la configuration du projet

Personnalisation et optimisation du pipeline Auto DevOps

- **Personnalisation**
  - Exclusion de stages spécifiques
  - Ajout de scripts personnalisés dans `.gitlab-ci.yml`
- **Optimisation**
  - Utilisation du cache et des artefacts entre les stages
  - Configuration des règles pour exécuter certaines étapes uniquement

Cas d'utilisation et bénéfices de l'Auto DevOps

- **Cas d'utilisation**
  - Projets nécessitant une mise en œuvre rapide du CI/CD
  - Équipes cherchant à adopter les meilleures pratiques sans configuration complexe
- **Bénéfices**
  - Accélération du cycle de développement
  - Amélioration continue de la qualité et de la sécurité des applications
  - Réduction des coûts d'infrastructure grâce à l'efficacité opérationnelle

Références

Architecture

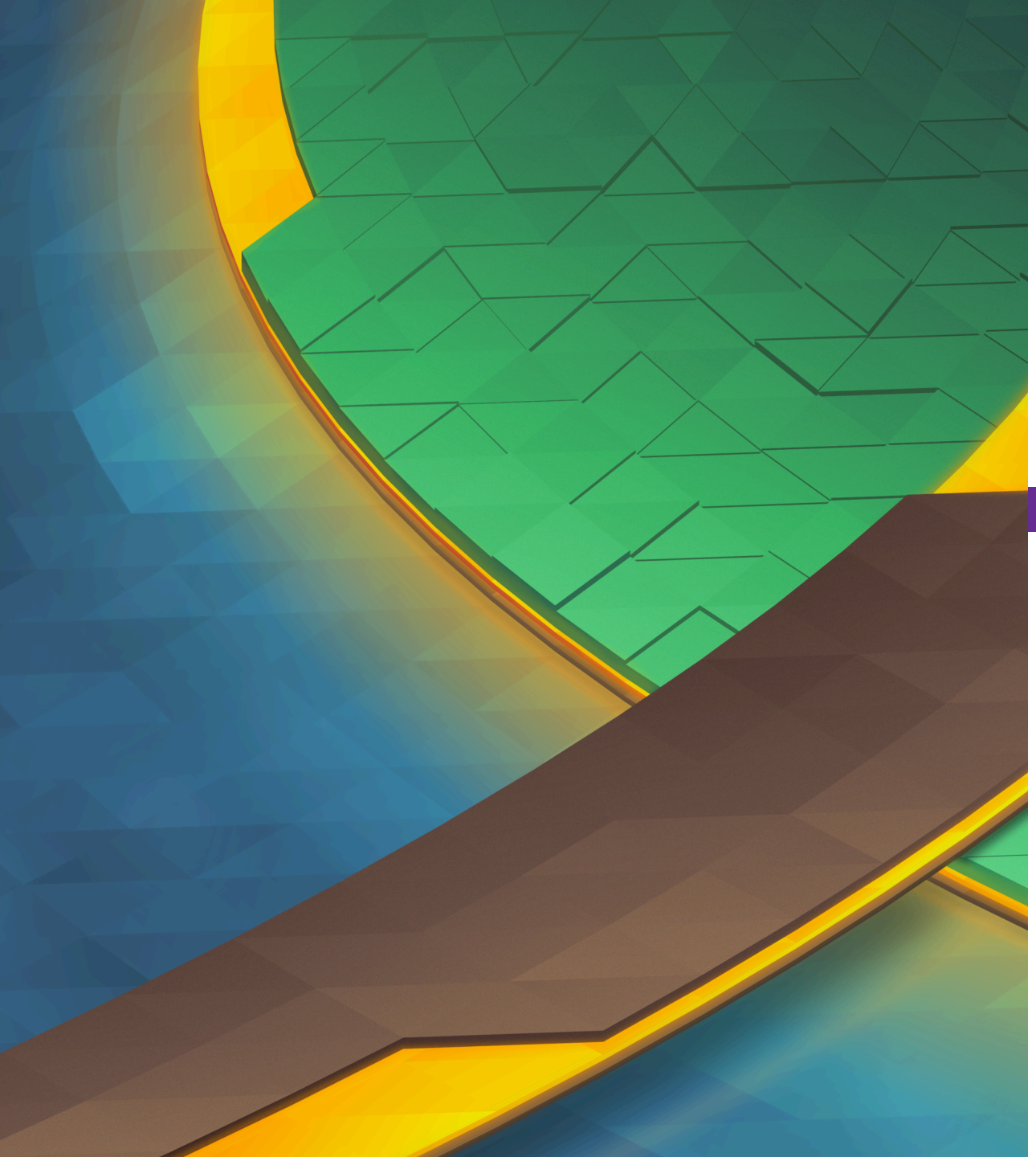
- Gitlab Docs: Runners
- GitLab Docs: Executors
- Valentin Despa: A Brief Guide to GitLab-CI Runners and Executors

Pipelines

- Syntax of .gitlab-ci.yaml
- .gitlab-ci.yml keyword reference
- François-Emmanel GOFFINET: Intégration continue avec GitLab-CI
- Gitlab for Dummies

Livraison et déploiement

- Gitlab Docs: Releases



Merci pour votre attention !