

Formation GITLAB/GITLAB-CI

Présentation

- Qui suis-je ?



Présentation

- Qui êtes-vous ?



Logistique



Pause en milieu de session



Vos questions sont les bienvenues.
N'hésitez pas !



Feuille d'évaluation à remettre remplie en fin de session



Merci d'éteindre vos téléphones

Sommaire

- Introduction
- Gestion du dépôt avec Gitlab
- Gitlab CI/CD
- La Gestion des environnements



Introduction

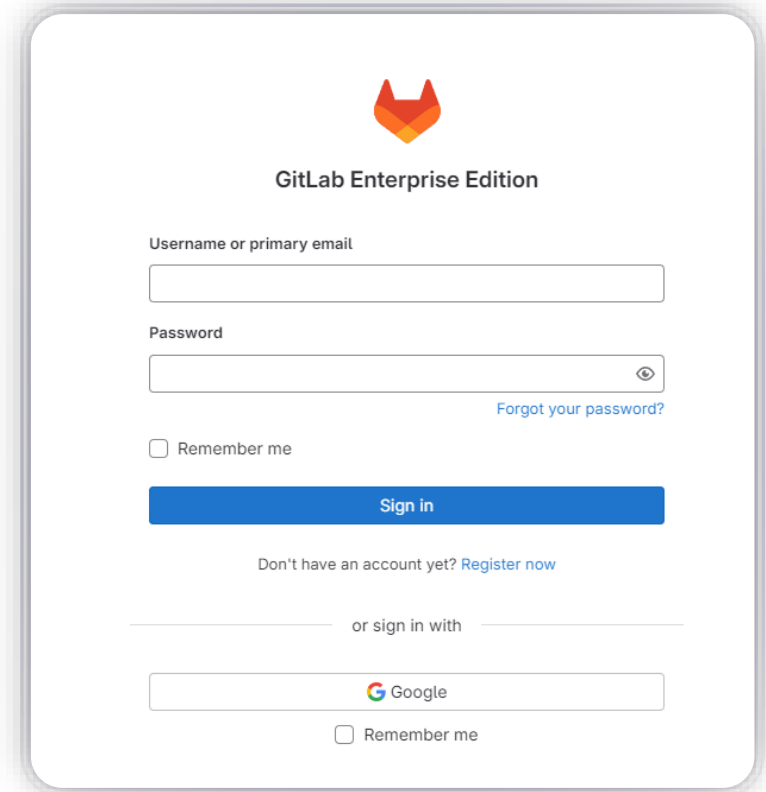
- Création d'un compte GitLab sur gitlab.com. Parcours de l'interface.
- GitLab Basics
- Les différents types d'utilisation de GitLab
- Les concepts Git
- Gestion locale des fichiers.
- Gestion des branches. Fusions des branches et gestion des conflits.

Introduction: GitLab CE & GitLab EE

- Création d'un compte GitLab sur gitlab.com. Parcours de l'interface.
- GitLab Basics
- Les différents types d'utilisation de GitLab : gitlab.com, on premise,
- Les concepts Git : blob, tree, commit, revision, branche, tag...
- Gestion locale des fichiers. Consultation et modification de l'historique de travail.
- Gestion des branches. Fusions des branches et gestion des conflits.
- Travaux pratiques
 - Mise en place d'un dépôt distant et simulation d'un travail collaboratif.

Création d'un compte GitLab

- Logging/mot de passe
- SSO



The image shows a screenshot of the GitLab Enterprise Edition login interface. At the top is the GitLab logo (an orange fox head) and the text "GitLab Enterprise Edition". Below this are two input fields: "Username or primary email" and "Password". The password field has a toggle icon (an eye) to its right. To the right of the password field is a link that says "Forgot your password?". Below the password field is a checkbox labeled "Remember me". A blue "Sign in" button is positioned below the "Remember me" checkbox. Below the button is a link that says "Don't have an account yet? Register now". Below this is a horizontal line with the text "or sign in with" in the center. Below the line is a Google sign-in button, which consists of the Google logo and the word "Google". Below the Google button is another checkbox labeled "Remember me".

GitLab Enterprise Edition

Username or primary email

Password


[Forgot your password?](#)

☐ Remember me

Sign in

Don't have an account yet? [Register now](#)

or sign in with


 Google

☐ Remember me

GitLab Basics

1. **Création d'un compte :** Comme décrit précédemment, vous devez créer un compte sur GitLab en fournissant un nom d'utilisateur, une adresse e-mail et un mot de passe.
2. **Création d'un projet :** Une fois connecté à GitLab, vous pouvez créer un nouveau projet. Un projet peut contenir le code source de votre application, des problèmes, des wikis, etc. Pour créer un projet, cliquez sur le bouton "New Project" sur le tableau de bord de GitLab.
3. **Clonage d'un projet :** Pour travailler sur un projet, vous devez le cloner sur votre machine locale. Utilisez la commande `git clone` en fournissant l'URL du projet GitLab.

```
bash
```


 Copy code

```
git clone <URL_du_projet>
```

GitLab Basics

4. **Ajout et commit de changements :** Effectuez des modifications dans les fichiers de votre projet et utilisez les commandes Git suivantes pour ajouter les modifications et créer un commit.


bash

 Copy code

```
git add .  
git commit -m "Description du commit"
```

5. **Pousser les changements vers GitLab :** Une fois que vous avez effectué vos commits locaux, poussez-les vers GitLab avec la commande `git push`.

bash

 Copy code


```
git push origin <nom_de_la_branche>
```

6. **Gestion des branches :** Les branches sont utilisées pour développer des fonctionnalités de manière isolée. Vous pouvez créer une nouvelle branche avec `git checkout -b <nom_de_la_branche>` et basculer entre les branches avec `git checkout <nom_de_la_branche>`.

GitLab Basics

7. **Merge (fusion) des branches** : Une fois que vous avez terminé le développement d'une fonctionnalité dans une branche, vous pouvez la fusionner dans la branche principale (généralement `master` ou `main`). Utilisez la commande `git merge`.

bash

 Copy code

```
git checkout master  
git merge <nom_de_la_branche>
```

8. **Gestion des problèmes (issues)** : Utilisez les problèmes GitLab pour suivre les tâches, les bogues, et d'autres améliorations. Vous pouvez attribuer des problèmes à des utilisateurs spécifiques, les étiqueter et les organiser dans des milestones.
9. **Intégration continue (CI/CD)** : GitLab propose des fonctionnalités d'intégration continue et de déploiement continu. Vous pouvez configurer des pipelines CI/CD pour automatiser les tests et le déploiement de votre application.
10. **Collaboration** : Invitez d'autres personnes à contribuer à votre projet en les ajoutant comme membres. Vous pouvez définir des niveaux d'accès différents pour chaque membre.

Les différents types d'utilisation de GitLab

1. Gestion de code source (Version Control System - VCS) :

- **Répertoires de code :** GitLab permet aux équipes de stocker leur code source dans des référentiels (repositories) Git.
- **Branches :** Les développeurs peuvent créer des branches pour travailler sur des fonctionnalités isolées ou des correctifs de bugs sans affecter la branche principale (habituellement `master`).
- **Fusion (Merge) :** Une fois les modifications terminées dans une branche, elles peuvent être fusionnées avec la branche principale.

2. Collaboration :

- **Issues :** Les équipes peuvent suivre les tâches, les fonctionnalités ou les bogues à l'aide des issues.
- **Merge Requests (MR) :** Les développeurs peuvent soumettre des demandes de fusion pour que leurs modifications soient examinées avant d'être fusionnées dans la branche principale.

3. Intégration Continue (CI) / Déploiement Continu (CD) :

- **Pipelines :** GitLab offre des fonctionnalités d'intégration continue où des pipelines automatisés peuvent être configurés pour tester et déployer automatiquement le code.
- **Runners :** Des agents d'exécution (runners) peuvent être configurés pour exécuter des tâches spécifiques dans le pipeline.

Les différents types d'utilisation de GitLab

4. Gestion de projet :

- **Tableau de bord du projet :** Les équipes peuvent organiser leur travail à l'aide de tableaux de bord, de listes de tâches, etc.
- **Wiki :** Les projets GitLab peuvent inclure des wikis pour la documentation du projet.

5. Gestion des droits d'accès :

- **Contrôle d'accès :** GitLab offre des fonctionnalités de gestion des droits d'accès pour contrôler qui peut accéder, lire, écrire ou fusionner du code dans un projet.

6. Gestion de la CI/CD en tant que code :

- **Fichiers de configuration CI/CD :** La configuration de l'intégration continue et du déploiement continu peut être définie en tant que code dans le référentiel.

7. Registre de conteneurs :

- **Registre GitLab :** Les équipes peuvent stocker et gérer des images de conteneurs Docker dans le registre intégré de GitLab.

8. Monitoring et Suivi :

- **Intégration de métriques et de suivi :** GitLab propose des fonctionnalités intégrées pour suivre les métriques et surveiller les performances du code.

Les concepts Git

1. Repository (Dépôt) :

- Un dépôt Git est un espace de stockage qui contient l'historique des fichiers et répertoires, ainsi que les métadonnées associées.
- Il peut être local (sur votre machine) ou distant (sur un serveur).

2. Commit :

- Un commit représente un ensemble de changements dans le code source.
- Chaque commit est associé à un message décrivant les modifications apportées.

3. Branch (Branche) :

- Une branche est une version parallèle du code source, créée à partir d'une branche existante (généralement la branche principale, appelée "master" ou "main").
- Les branches permettent le développement simultané de fonctionnalités sans perturber la branche principale.

4. Merge (Fusion) :

- La fusion est le processus de combinaison des modifications d'une branche vers une autre, généralement pour intégrer les modifications d'une fonctionnalité dans la branche principale.

5. Pull Request (Demande de tirage) :

- Une demande de tirage est une fonctionnalité des plates-formes d'hébergement de code telles que GitHub ou GitLab. Elle permet à un développeur de demander à intégrer ses modifications d'une branche vers une autre.

Les concepts Git

6. Clone :

- Cloner un dépôt signifie créer une copie locale d'un dépôt distant sur votre machine.

7. Remote (Dépôt distant) :

- Un dépôt distant est une copie du dépôt Git hébergée sur un serveur. Il peut être utilisé pour la collaboration entre plusieurs développeurs.

8. Fetch :

- La récupération est le processus de mise à jour de votre dépôt local avec les dernières modifications du dépôt distant, sans fusionner ces modifications avec votre code actuel.

9. Pull :

- Un pull est similaire à une récupération, mais il récupère également les modifications du dépôt distant et les fusionne automatiquement avec votre branche locale.

10. Push :

- Pousser consiste à envoyer vos modifications locales vers un dépôt distant.

11. Conflict (Conflit) :

- Un conflit survient lorsqu'il y a des modifications concurrentes dans des parties du code, et Git ne peut pas fusionner automatiquement ces changements. Il nécessite une intervention manuelle pour résoudre le conflit.

12. Tag :

- Un tag est une référence immuable pointant vers un commit spécifique. Il est souvent utilisé pour marquer des versions stables du logiciel.

Gestion locale des fichiers

1. Initialisation d'un référentiel Git :

- Pour commencer à utiliser Git localement, vous devez d'abord initialiser un référentiel Git dans votre répertoire de projet en utilisant la commande `git init`. Cela crée un sous-dossier `.git` qui contient toutes les informations nécessaires à Git pour suivre les modifications de votre projet.

```
bash
```

[Copy code](#)

```
git init
```

2. Ajout de fichiers au suivi de Git :

- Utilisez la commande `git add` pour ajouter des fichiers à l'index (zone de suivi) de Git. Cela signifie que les modifications de ces fichiers seront incluses dans le prochain commit.

```
bash
```

[Copy code](#)

```
git add nom_fichier
```

- Pour ajouter tous les fichiers modifiés, vous pouvez utiliser :

```
bash
```

[Copy code](#)

```
git add .
```


Gestion locale des fichiers

3. Création de commits :

- Une fois que vous avez ajouté des fichiers à l'index, vous pouvez créer un commit avec la commande `git commit`. Cela enregistre les modifications dans le référentiel Git.

```
bash
```

[Copy code](#)

```
git commit -m "Message de commit descriptif"
```

4. Visualisation de l'état du référentiel :

- Utilisez la commande `git status` pour voir l'état actuel de votre répertoire de travail par rapport au référentiel Git.

```
bash
```

[Copy code](#)

```
git status
```

5. Historique des commits :

- Pour voir l'historique des commits, utilisez la commande `git log`.

```
bash
```

[Copy code](#)


```
git log
```

Gestion locale des fichiers

6. Gestion des branches :

- Les branches sont un moyen puissant de travailler sur des fonctionnalités isolées. Utilisez ``git branch`` pour voir les branches existantes et ``git checkout`` pour basculer entre les branches.

bash


 Copy code

```
git branch  
git checkout nom_de_branche
```

7. Récupération des modifications à distance :

- Utilisez ``git pull`` pour récupérer les modifications depuis la branche distante actuelle, et ``git push`` pour pousser vos modifications locales.

bash

 Copy code

```
git pull origin nom_de_branche  
git push origin nom_de_branche
```

Travaux pratiques



➤ Création d'un référentiel (repository) local :

- Initialisez un nouveau référentiel Git dans un dossier local.
- Ajoutez un fichier et effectuez un commit initial.

➤ Travail avec des branches :

- Créez une nouvelle branche pour développer une nouvelle fonctionnalité.
- Effectuez des modifications dans cette nouvelle branche.
- Fusionnez la nouvelle branche dans la branche principale.

➤ Gestion des conflits :

- Intentionnellement, créez un conflit en modifiant la même ligne dans deux branches différentes.
- Résolvez le conflit manuellement.
- Poussez vos modifications sur le référentiel distant.

• Collaboration avec des référentiels distants :

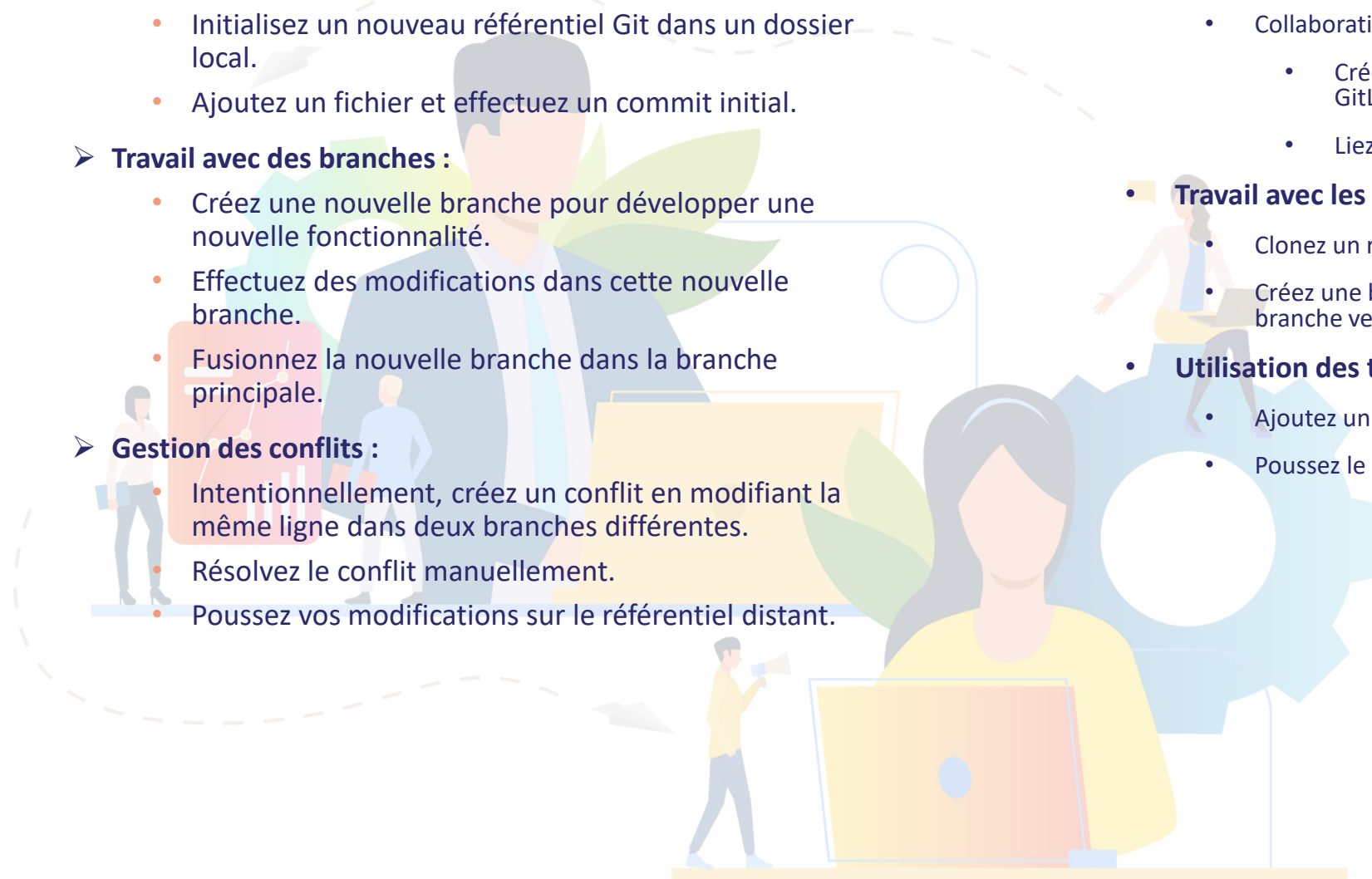
- Créez un référentiel sur une plateforme comme GitHub ou GitLab.
- Liez votre référentiel local au référentiel distant.

• Travail avec les remotes :

- Clonez un référentiel distant.
- Créez une branche locale, effectuez des modifications et poussez la branche vers le référentiel distant.

• Utilisation des tags :

- Ajoutez un tag pour marquer une version spécifique de votre projet.
- Poussez le tag vers le référentiel distant.



Solutions



```
bash

git init
touch README.md
git add README.md
git commit -m "Premier commit"
```

```
bash

# Sur la branche principale
git branch branche-1
git checkout branche-1
# Modifiez le même fichier sur la branche-1
git add .
git commit -m "Modification sur branche-1"
git checkout master
# Modifiez le même fichier sur la branche principale
git add .
git commit -m "Modification sur la branche principale"
# Merge avec conflit
git merge branche-1
```

```
bash

git branch nouvelle-fonctionnalite
git checkout nouvelle-fonctionnalite
# Effectuez des modifications
git add .
git commit -m "Ajout de la nouvelle fonctionnalité"
git checkout master
git merge nouvelle-fonctionnalite
```

```
bash

# Sur GitHub (ou GitLab)
# Créez un nouveau référentiel vide
# Copiez le lien du référentiel

# Sur le terminal local
git remote add origin <lien-du-referentiel>
git branch -M main
git push -u origin main
```

```
bash

git clone <lien-du-referentiel>
cd nom-du-referentiel
git checkout -b nouvelle-branche
# Effectuez des modifications
git add .
git commit -m "Nouvelle modification"
git push origin nouvelle-branche
```

```
bash

git tag -a v1.0 -m "Version 1.0"
git push --tags
```

Sommaire

- Introduction
- **Gestion du dépôt avec Gitlab**
- Gitlab CI/CD
- La Gestion des environnements



La gestion du dépôt avec GitLab

- La gestion des collaborateurs d'un projet et leurs droits.
- Le système d'issues et le lien avec les commits.
- Présentation du wiki et des snippets.
- Travaux pratiques

La gestion des collaborateurs d'un projet et leurs droits

1. Rôles :

- **Mainteneur (Maintainer) :** Les mainteneurs ont un accès complet au projet. Ils peuvent ajouter, modifier et supprimer des fichiers, ainsi que gérer les membres du projet.
- **Développeur (Developer) :** Les développeurs peuvent contribuer au code source en soumettant des modifications, créer des branches, etc.
- **Reporter :** Les reporters ont un accès en lecture seule au projet et peuvent signaler des problèmes, commenter, mais ne peuvent pas apporter de modifications directes au code.

2. Membres du projet :

- Les utilisateurs individuels ou les groupes peuvent être ajoutés en tant que membres d'un projet. La gestion des membres est effectuée dans les paramètres du projet.

3. Invitations :

- Les mainteneurs peuvent envoyer des invitations aux utilisateurs pour les inviter à rejoindre un projet. Les invitations peuvent être envoyées à des utilisateurs spécifiques ou à des groupes entiers.

La gestion des collaborateurs d'un projet et leurs droits

4. Groupes :

- GitLab permet de créer des groupes qui regroupent plusieurs projets. Les membres d'un groupe ont généralement des droits similaires sur tous les projets inclus dans le groupe.

5. Branches protégées :

- Les mainteneurs peuvent protéger certaines branches du projet pour éviter des modifications non autorisées. Les règles de protection de branche peuvent spécifier qui peut effectuer des opérations telles que la fusion (merge) sur une branche.

6. Contrôle d'accès avancé :

- GitLab propose un système de contrôle d'accès basé sur des règles plus avancé, appelé "Access Control Lists" (ACL), qui permet de définir des autorisations très spécifiques pour différents utilisateurs ou groupes.

7. Web IDE et édition en ligne :

- GitLab propose un éditeur en ligne et un environnement de développement intégré (Web IDE) qui permettent aux utilisateurs autorisés de modifier des fichiers directement depuis l'interface web.

Le système d'issues et le lien avec les commits

1. Commits :

- Un commit est une unité de base dans un système de contrôle de versions comme Git. Il représente un ensemble de modifications apportées à un ensemble de fichiers.
- Lorsque vous effectuez des changements dans votre code source, vous créez un commit pour enregistrer ces modifications. Chaque commit a un message qui décrit les changements effectués.
- Les commits sont utilisés pour suivre l'évolution du code au fil du temps. Ils facilitent la collaboration, permettent de revenir à des versions antérieures du code et aident à comprendre l'historique des modifications.

2. Système d'issues :

- Un système d'issues est un outil utilisé pour suivre, gérer et discuter des tâches, des bogues, des améliorations et d'autres types de travaux à réaliser dans un projet logiciel.
- Les issues sont souvent utilisées pour signaler des problèmes, suggérer des fonctionnalités, attribuer des tâches aux membres de l'équipe, etc.
- Chaque issue a généralement un titre, une description détaillée et peut être associée à des étiquettes, des responsables, des dates d'échéance, etc.

Le système d'issues et le lien avec les commits

- Les commits et les issues sont liés dans le sens où les commits sont souvent associés à des problèmes spécifiques.
- Lorsqu'un développeur travaille sur une issue particulière, il peut créer une branche de code dédiée à cette issue. Les commits effectués dans cette branche sont ensuite regroupés (via une fusion ou pull request) pour résoudre l'issue.
- Lorsqu'un commit est effectué pour résoudre un problème spécifique, le message de commit peut faire référence à l'issue associée en utilisant un numéro d'issue. Par exemple, un message de commit pourrait contenir "Fix #123" pour indiquer que le commit résout l'issue numéro 123.

En résumé, les commits sont des enregistrements des modifications du code, tandis que les issues sont des éléments de suivi pour le travail à accomplir. Les commits peuvent être liés aux issues, créant ainsi une traçabilité entre les changements dans le code et les tâches spécifiques ou les problèmes qu'ils résolvent.

Présentation du wiki et des snippets

- Le wiki GitLab est un espace collaboratif où les membres du projet peuvent créer, éditer et partager des documents directement dans le dépôt GitLab.
- Il est souvent utilisé pour stocker des informations telles que la documentation du projet, des guides, des notes, des procédures, etc.
- Le wiki peut être créé pour chaque projet GitLab, offrant ainsi un moyen pratique d'organiser et de partager des informations liées au projet.
- Les utilisateurs peuvent éditer les pages du wiki via l'interface web GitLab, et les changements sont enregistrés dans le dépôt Git, ce qui signifie qu'ils sont versionnés et peuvent être suivis au fil du temps.

Présentation du wiki et des snippets

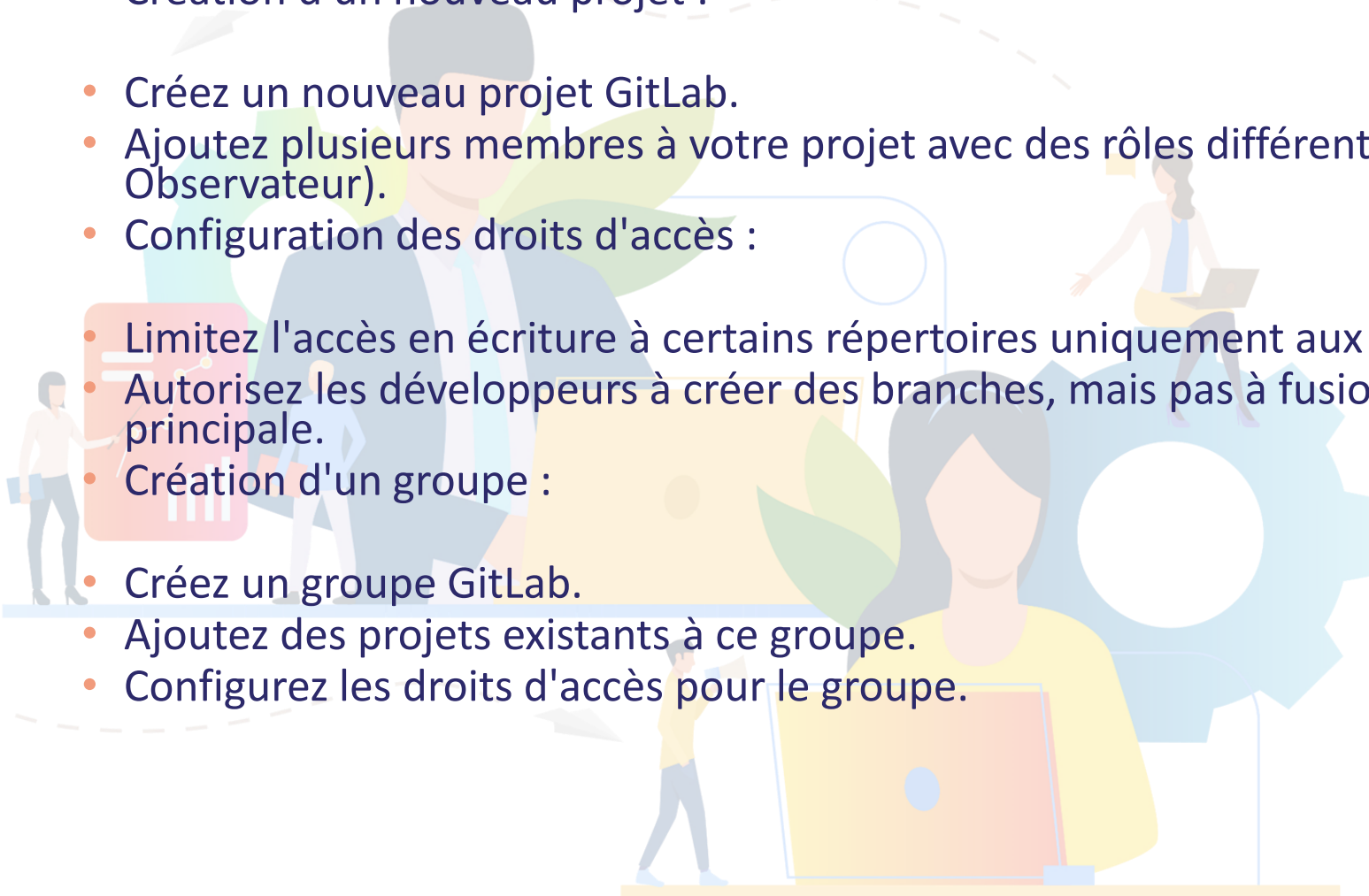
- Les snippets GitLab sont des fragments de code ou des portions de texte que vous pouvez créer, stocker et partager de manière indépendante du code source principal du projet.
- Ils peuvent être publics, accessibles à tous, ou privés, accessibles uniquement aux membres autorisés du projet.
- Les snippets peuvent être utilisés pour partager des exemples de code, des configurations, des commandes, ou tout autre morceau d'information texte.
- Les snippets peuvent également inclure des fichiers, et ils sont utiles pour partager du code entre des projets différents ou avec d'autres utilisateurs sans avoir à créer un dépôt complet.

Travaux pratiques



➤ Exercice 1 : Paramétrage des droits

- Création d'un nouveau projet :
- Créez un nouveau projet GitLab.
- Ajoutez plusieurs membres à votre projet avec des rôles différents (Mainteneur, Développeur, Observateur).
- Configuration des droits d'accès :
 - Limitez l'accès en écriture à certains répertoires uniquement aux mainteneurs.
 - Autorisez les développeurs à créer des branches, mais pas à fusionner directement dans la branche principale.
- Création d'un groupe :
 - Créez un groupe GitLab.
 - Ajoutez des projets existants à ce groupe.
 - Configurez les droits d'accès pour le groupe.

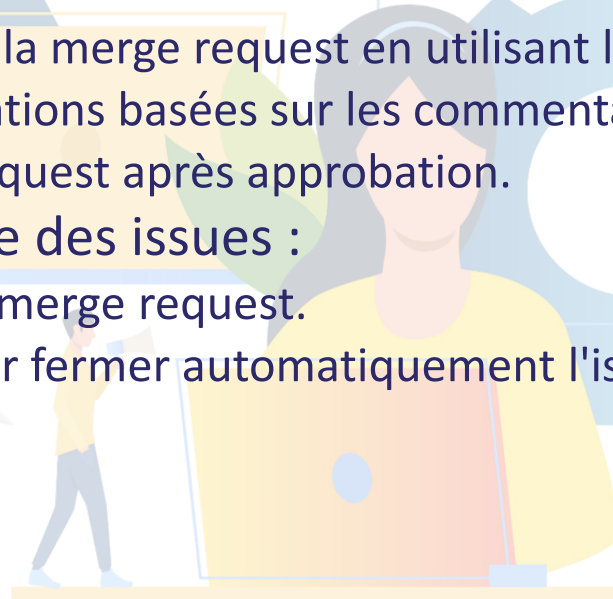


Travaux pratiques



➤ Exercice 2 : Création de Merge Requests

- Branches de fonctionnalité :
 - Créez une nouvelle branche pour une fonctionnalité dans votre projet.
 - Ajoutez quelques modifications à cette branche.
- Création de Merge Request :
 - Créez une merge request à partir de la branche de fonctionnalité vers la branche principale.
 - Ajoutez des détails sur les changements effectués et demandez à un collègue de la réviser.
- Révision et Fusion :
 - Faites une révision de la merge request en utilisant les commentaires sur le code.
 - Effectuez des modifications basées sur les commentaires.
 - Fusionnez la merge request après approbation.
- Fermeture automatique des issues :
 - Liez une issue à votre merge request.
 - Configurez GitLab pour fermer automatiquement l'issue lors de la fusion de la merge request.

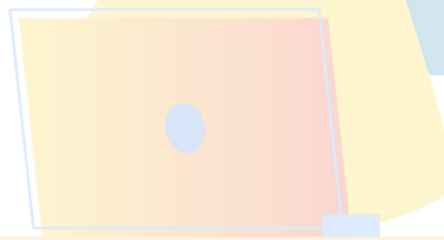


Travaux pratiques



➤ Exercice 3 : Intégration continue

- Configuration d'un pipeline CI/CD :
 - Ajoutez un fichier `.gitlab-ci.yml` à votre projet avec un pipeline CI/CD simple.
 - Utilisez une image Docker pour exécuter vos étapes de build et de test.
- Ajout de badges au README :
 - Configurez GitLab CI pour générer des badges.
 - Ajoutez ces badges à votre fichier `README.md` pour afficher l'état du dernier pipeline.
- Utilisation des variables d'environnement :
 - Utilisez des variables d'environnement sécurisées dans votre fichier `.gitlab-ci.yml`.
 - Exécutez des commandes basées sur ces variables.



Sommaire

- Introduction
- Gestion du dépôt avec Gitlab
- **Gitlab CI/CD**
- La Gestion des environnements



GitLab CI/CD

- Présentation de GitLab CI/CD et des GitLab runners.
- Présentation de Docker.
- Auto DevOps
- Features flags et Progressive Delivery
- Le fichier manifeste gitlab-ci.yml, présentation du langage YAML.
- Les balises essentielles pour décrire des étapes, des jobs, des traitements (stages, images, script...).
- Le suivi d'exécution du pipeline. Jobs automatiques, manuels et planifiés.
- Les artifacts et l'amélioration des performances avec le cache.
- La documentation officielle relative à la syntaxe du fichier manifeste.

Présentation de GitLab CI/CD et des GitLab runners

- **GitLab CI/CD** (Continuous Integration/Continuous Deployment) est une composante intégrée à la plateforme GitLab, offrant des fonctionnalités complètes pour automatiser le processus de développement logiciel, du code source à la livraison.

1. **Pipeline CI/CD** : Un pipeline est une suite d'étapes automatisées qui permettent de construire, tester et déployer des applications de manière continue. Chaque pipeline est défini dans un fichier `.gitlab-ci.yml` au sein du dépôt GitLab.
2. **Jobs** : Les jobs sont les tâches individuelles à l'intérieur d'un pipeline. Chaque job est responsable d'une étape spécifique du processus, comme la compilation du code, l'exécution de tests, la construction de conteneurs, etc.
3. **Runners** : Les runners sont des agents qui exécutent les jobs définis dans le pipeline. Ils peuvent être hébergés sur des machines dédiées ou partagées, y compris sur les infrastructures cloud.
4. **Intégration avec les environnements cloud** : GitLab CI/CD offre une intégration native avec des services cloud populaires, ce qui facilite le déploiement sur des plateformes telles que AWS, Google Cloud, Azure, etc.
5. **Déploiement continu** : GitLab CI/CD prend en charge le déploiement continu, permettant d'automatiser la mise en production des applications après la réussite des tests.
6. **Sécurité intégrée** : GitLab CI/CD intègre des fonctionnalités de sécurité telles que l'analyse statique du code, la vérification des dépendances, et la détection des vulnérabilités, contribuant ainsi à améliorer la sécurité du code produit.

Présentation de GitLab CI/CD et des GitLab runners

- **GitLab Runners** sont des agents d'exécution qui permettent l'exécution des jobs définis dans les pipelines de GitLab CI/CD. Ils jouent un rôle crucial dans le processus d'automatisation en exécutant les tâches définies dans le fichier `.gitlab-ci.yml`. Voici quelques points clés sur les GitLab Runners :

1. **Shared Runners** : Ces runners sont partagés entre plusieurs projets. Ils sont gérés au niveau de l'instance GitLab et peuvent être utilisés par tous les projets de cette instance.
2. **Specific Runners** : Ces runners sont spécifiques à un projet particulier. Ils sont configurés et utilisés uniquement pour ce projet spécifique.

Caractéristiques et fonctionnalités :

1. **Compatibilité multi-plateforme** : Les GitLab Runners peuvent être installés sur différentes plateformes, y compris Linux, Windows et macOS, offrant une flexibilité dans le choix de l'infrastructure.
2. **Extensions** : Les GitLab Runners peuvent être étendus avec des tags et des exécuteurs spécifiques pour s'adapter à des scénarios d'exécution plus complexes.
3. **Exécution de conteneurs** : Les GitLab Runners peuvent exécuter des jobs dans des conteneurs Docker, permettant une isolation et une reproductibilité des environnements d'exécution.
4. **Évolutivité** : Les GitLab Runners peuvent être déployés en mode scalable, permettant d'ajuster la capacité d'exécution en fonction des besoins de charge.

Présentation de Docker

- **Docker** est une plateforme open-source qui automatise le déploiement d'applications à l'intérieur de conteneurs logiciels. Ces conteneurs sont légers, portables et autonomes, ce qui facilite la création, la distribution et l'exécution d'applications dans des environnements isolés. Docker a révolutionné la manière dont les développeurs, les opérationnels et les équipes de DevOps gèrent et déploient des applications.

Avantages de Docker :

- **Consistance de l'environnement** : Élimination des problèmes liés aux différences d'environnement entre les étapes de développement, de test et de production.
- **Efficacité des ressources** : Utilisation optimale des ressources système grâce à la virtualisation légère des conteneurs.
- **Facilité de déploiement** : Rapidité de déploiement des applications sur n'importe quelle infrastructure prenant en charge Docker.
- **Scalabilité** : Possibilité de scaler les applications horizontalement en ajoutant des conteneurs.
- **Gestion des dépendances** : Les dépendances logicielles sont encapsulées dans les conteneurs, évitant les conflits et les problèmes de compatibilité.

Auto DevOps

- Auto DevOps est une fonctionnalité intégrée à GitLab qui vise à automatiser une grande partie du processus de développement, de la construction à la mise en production. Il s'agit d'un ensemble de meilleures pratiques et de pipelines CI/CD prédéfinis, conçus pour simplifier le déploiement continu et la livraison continue dans les projets GitLab.

Utilisation d'Auto DevOps dans GitLab :

1. Activation :

- Auto DevOps peut être activé pour un projet GitLab en quelques clics dans les paramètres du projet.

2. Personnalisation :

- Bien qu'Auto DevOps fournisse un pipeline par défaut, il peut être personnalisé en ajoutant ou en modifiant des étapes pour répondre aux besoins spécifiques du projet.

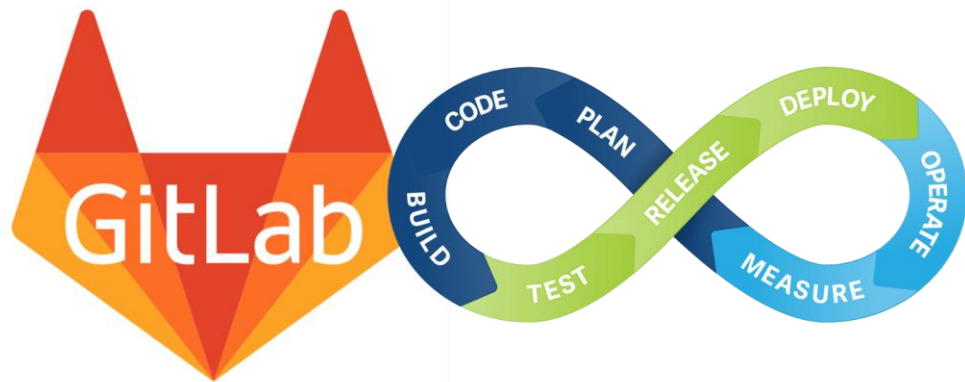
3. Variables d'environnement :

- Les variables d'environnement peuvent être configurées pour ajuster le comportement d'Auto DevOps en fonction des exigences spécifiques du projet.

4. Intégration avec d'autres fonctionnalités GitLab :

- Auto DevOps s'intègre parfaitement avec d'autres fonctionnalités GitLab telles que le registre de conteneurs, les déploiements sur Kubernetes, les rapports de sécurité, etc.

Auto DevOps



1. Pipeline CI/CD prédéfini :

- Auto DevOps définit un pipeline CI/CD par défaut pour les projets GitLab. Ce pipeline inclut des étapes telles que la construction, les tests, la sécurité, la vérification de la qualité du code, le déploiement et la surveillance.

2. Détection automatique des applications :

- Auto DevOps essaie de détecter automatiquement le type d'application que vous développez (par exemple, Ruby, Node.js, Java) et ajuste le pipeline en conséquence, en utilisant des conventions et des modèles prédéfinis.

3. Déploiement automatique :

- Le pipeline Auto DevOps est configuré pour déployer automatiquement les applications sur différents environnements, tels que le développement, la production, etc.
- Les déploiements peuvent se faire sur des clusters Kubernetes, des serveurs virtuels, ou d'autres plateformes prises en charge par GitLab.

4. Intégration de la sécurité :

- Auto DevOps intègre des analyses de sécurité dans le pipeline, y compris la détection des vulnérabilités dans les dépendances et la vérification de la conformité avec les meilleures pratiques de sécurité.

5. Suivi des performances :

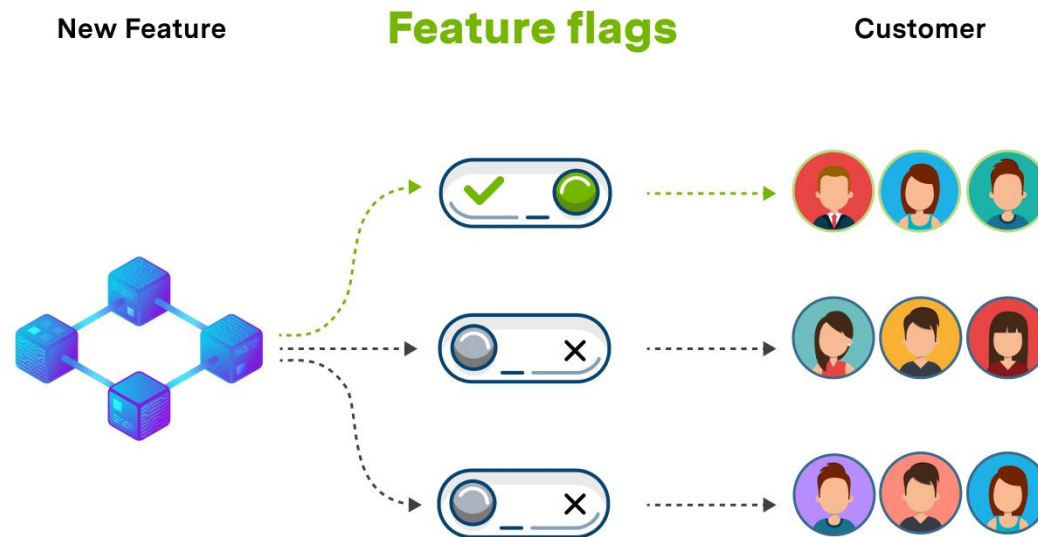
- La fonctionnalité inclut également des outils de suivi des performances pour aider à identifier les goulots d'étranglement et à améliorer les performances de l'application.

6. Gestion automatique des environnements :

- Auto DevOps gère automatiquement la création et la gestion des environnements pour chaque branche, simplifiant ainsi les tests et les validations parallèles.

Features flags et Progressive Delivery

- Dans GitLab, les fonctionnalités associées aux Feature Flags et au Progressive Delivery font partie intégrante des capacités de gestion du cycle de vie des applications. GitLab propose plusieurs fonctionnalités et outils pour mettre en œuvre ces concepts



Features flags et Progressive Delivery

Progressive Delivery dans GitLab :

1. Review Apps :

- Les Review Apps dans GitLab facilitent le déploiement d'applications pour chaque fusion (merge request) afin de permettre aux développeurs de tester des modifications de code dans un environnement isolé avant la fusion.
- Les Review Apps peuvent être utilisées pour tester des fonctionnalités spécifiques avant qu'elles ne soient fusionnées dans la branche principale.

2. Canary Deployments :

- GitLab Auto DevOps propose la possibilité de réaliser des déploiements canari, qui consistent à déployer une nouvelle version d'une application pour un sous-ensemble de l'infrastructure ou des utilisateurs afin de minimiser les risques en cas de problème.

3. Feature Flags avec Review Apps :

- GitLab permet d'intégrer les Feature Flags avec les Review Apps pour une approche plus avancée du Progressive Delivery.
- Cela signifie que vous pouvez activer des Feature Flags spécifiques pour les Review Apps, permettant de tester des fonctionnalités spécifiques dans des environnements isolés avant de les déployer plus largement.

Feature Flags dans GitLab :

1. Feature Flags :

- GitLab propose une gestion native des Feature Flags dans le cadre du développement de fonctionnalités.
- Les Feature Flags dans GitLab permettent de déployer du code avec des fonctionnalités cachées derrière une bascule. Cela permet de contrôler l'activation ou la désactivation de fonctionnalités en temps réel, sans nécessiter de déploiement supplémentaire.
- Vous pouvez définir des conditions pour activer automatiquement ou désactiver une feature flag en fonction de critères spécifiques, tels que le pourcentage d'utilisateurs, les utilisateurs spécifiques, etc.
- GitLab intègre également les Feature Flags avec le suivi des performances, facilitant l'observation des performances des fonctionnalités activées.


Le fichier manifeste gitlab-ci.yml, présentation du langage YAML

- Le fichier gitlab-ci.yml est un fichier de configuration essentiel dans GitLab qui définit les étapes et les paramètres du pipeline CI/CD (Intégration Continue / Déploiement Continu). Il s'agit d'un fichier YAML (YAML Ain't Markup Language) qui est utilisé pour décrire comment les différentes étapes du pipeline doivent être exécutées.

1. Définition des Stages :

- Les étapes du pipeline sont définies sous la clé `stages`. Chaque étape représente une phase spécifique du processus d'intégration et de déploiement. Par exemple : `build`, `test`, `deploy`, etc.

yaml

 Copy code


```
stages:  
  - build  
  - test  
  - deploy
```

Le fichier manifeste gitlab-ci.yml, présentation du langage YAML

2. Jobs :

- Les jobs représentent des tâches individuelles à l'intérieur de chaque étape. Chaque job peut avoir ses propres scripts, dépendances et actions. Ils sont définis sous la clé `jobs`.

yaml

 Copy code

```
jobs:
  - build_job:
      script:
        - echo "Building the application..."
  - test_job:
      script:
        - echo "Running tests..."
  - deploy_job:
      script:
        - echo "Deploying the application..."
```

Le fichier manifeste gitlab-ci.yml, présentation du langage YAML

3. Variables d'environnement :

- Les variables d'environnement peuvent être définies au niveau du fichier `gitlab-ci.yml` pour être utilisées dans les scripts des jobs.

yaml

 Copy code


```
variables:  
  DATABASE_URL: "postgres://user:password@host:5432/db"
```

Le fichier manifeste gitlab-ci.yml, présentation du langage YAML

4. Scripts d'exécution des Jobs :

- Chaque job peut avoir un script spécifique à exécuter. Il peut s'agir de commandes de compilation, de tests, de déploiement, etc.

yaml

 Copy code


```
jobs:
  - build_job:
    script:
      - npm install
      - npm run build
  - test_job:
    script:
      - npm test
  - deploy_job:
    script:
      - docker-compose up -d
```

Le fichier manifeste gitlab-ci.yml, présentation du langage YAML

5. Dépendances entre les Jobs :

- Vous pouvez spécifier des dépendances entre les jobs. Par exemple, le job `test` pourrait dépendre du job `build`.

yaml

 Copy code

```
jobs:
  - build_job:
      script:
        - echo "Building the application..."
  - test_job:
      script:
        - echo "Running tests..."
      dependencies:
        - build_job
```

Le fichier manifeste gitlab-ci.yml, présentation du langage YAML

6. Déclencheurs (Triggers) :

- Vous pouvez spécifier des déclencheurs pour déclencher le pipeline, par exemple lors d'un push sur une branche spécifique.

yaml

 Copy code

```
trigger:
  include:
    - local: '/path/to/another/repository/.gitlab-ci.yml'
```

Le fichier manifeste gitlab-ci.yml, présentation du langage YAML

7. Notifications :

- Vous pouvez définir des notifications pour être informé des résultats du pipeline.

yaml

 Copy code


```
notify:
  email:
    recipients:
      - developer@example.com
```

Le fichier manifeste gitlab-ci.yml, présentation du langage YAML

Balises pour décrire des étapes (`stages`):

1. **`stages`** : Définit la liste des étapes du pipeline CI/CD.

yaml

 Copy code

```
stages:  
  - build  
  - test  
  - deploy
```


Le fichier manifeste gitlab-ci.yml, présentation du langage YAML

Balises pour décrire des jobs :

1. **`jobs`** : Définit la liste des jobs à exécuter. Chaque job est une unité d'exécution dans une étape.

```
yaml Copy code
jobs:
  - build_job:
    script:
      - echo "Building the application..."
  - test_job:
    script:
      - echo "Running tests..."
```

2. **`script`** : Contient les commandes à exécuter pour le job. Les commandes sont écrites en shell.

```
yaml Copy code
jobs:
  - build_job:
    script:
      - npm install
      - npm run build
```

Les balises essentielles pour décrire des étapes, des jobs, des traitements (stages, images, Script,,

3. **`stage`** : Spécifie l'étape à laquelle le job appartient.

```
yaml Copy code

jobs:
  - build_job:
      stage: build
      script:
        - echo "Building the application..."
```

4. **`artifacts`** : Permet de spécifier des fichiers ou des répertoires à conserver en tant qu'artefacts du job, qui peuvent être utilisés par d'autres jobs.

```
yaml Copy code

jobs:
  - build_job:
      artifacts:
        paths:
          - build/
      script:
        - echo "Building the application..."
```

5. **`only` et `except`** : Permettent de spécifier les conditions pour exécuter ou exclure un job en fonction de certains critères, tels que des branches spécifiques.

```
yaml Copy code

jobs:
  - build_job:
      script:
        - echo "Building the application..."
      only:
        - master
```

Les balises essentielles pour décrire des étapes, des jobs, des traitements (stages, images, script...)

Balises pour décrire des traitements spécifiques :

1. `image` : Spécifie l'image Docker à utiliser pour le job. L'image contient l'environnement d'exécution du job.

```
yaml
jobs:
  - build_job:
      script:
        - echo "Building the application..."
      image: node:14
```

2. `services` : Spécifie les services Docker à démarrer et à lier au job.

```
yaml
jobs:
  - test_job:
      script:
        - echo "Running tests..."
      services:
        - postgres:latest
```

3. `variables` : Définit des variables d'environnement spécifiques au job.

```
yaml
jobs:
  - build_job:
      script:
        - echo "Building the application..."
      variables:
        DATABASE_URL: "postgres://user:password@host:5432/db"
```

Le suivi d'exécution du pipeline. Jobs automatiques, manuels et planifiés

- Le suivi de l'exécution du pipeline dans GitLab est une partie importante du processus d'intégration continue. GitLab offre plusieurs mécanismes pour suivre l'exécution des jobs, que ce soit automatique, manuel ou planifié.

Suivi d'exécution automatique :

1. Tableau de bord du pipeline :

- Sur l'interface web de GitLab, le tableau de bord du pipeline fournit une vue d'ensemble de l'état actuel du pipeline.
- Vous pouvez visualiser les étapes du pipeline, les jobs en cours d'exécution, les jobs terminés et les résultats.

2. Notifications par e-mail et webhooks :

- GitLab peut être configuré pour envoyer des notifications par e-mail ou déclencher des webhooks à chaque fois qu'un job ou un pipeline est terminé.

3. Rapports de pipeline :

- GitLab génère des rapports détaillés pour chaque pipeline, montrant les résultats de chaque job, les durées d'exécution, les erreurs éventuelles, etc.


Le suivi d'exécution du pipeline. Jobs automatiques, manuels et planifiés

Suivi d'exécution manuelle :

1. Déclenchement manuel des jobs :

- Certains jobs peuvent être configurés pour ne pas être déclenchés automatiquement à chaque commit, mais plutôt manuellement. Cela peut être utile pour des tâches spécifiques ou des déploiements.

yaml

 Copy code

```
jobs:
  - deploy_job:
      script:
        - echo "Deploying the application..."
      when: manual
```

- Le job ci-dessus doit être déclenché manuellement à partir de l'interface web GitLab.

2. Approbation manuelle des merge requests :

- Dans le contexte d'une merge request, vous pouvez configurer GitLab pour exiger une approbation manuelle avant de fusionner la branche.


Le suivi d'exécution du pipeline. Jobs automatiques, manuels et planifiés

Suivi d'exécution planifiée :

1. Jobs planifiés (Scheduled Jobs) :

- Vous pouvez configurer des jobs pour être exécutés à des intervalles réguliers en utilisant les jobs planifiés.

yaml

 Copy code

```
jobs:
  - scheduled_job:
      script:
        - echo "This job is scheduled to run at specific intervals"
      only:
        - schedules
```

- Ces jobs sont généralement utilisés pour des tâches telles que des sauvegardes, des mises à jour périodiques, etc.

2. Calendrier d'exécution :

- GitLab propose un calendrier d'exécution intégré où vous pouvez planifier l'exécution des pipelines à des moments spécifiques.

Le suivi d'exécution du pipeline. Jobs automatiques, manuels et planifiés

Suivi des résultats :

1. Visualisation des logs et artefacts :

- Les logs des jobs sont accessibles depuis l'interface web GitLab, permettant de visualiser en détail chaque étape de l'exécution.
- Les artefacts générés par les jobs peuvent également être téléchargés pour une analyse plus approfondie.

2. Rapports de tests :

- Si vos jobs incluent des tests, GitLab génère des rapports de tests qui fournissent des informations sur la réussite ou l'échec des tests individuels.

3. Tableau de bord de la performance :

- GitLab propose des tableaux de bord de performance qui permettent de suivre les métriques importantes telles que le temps d'exécution des jobs, le taux de réussite, etc.

Les artifacts et l'amélioration des performances avec le cache

- Les artifacts dans GitLab sont des fichiers et des répertoires générés par les jobs d'un pipeline et qui peuvent être conservés pour une utilisation ultérieure. Les principaux points à noter sur les artifacts sont les suivants :

1. Définition des Artifacts :

- Dans la configuration d'un job, vous pouvez spécifier quels fichiers ou répertoires doivent être conservés comme artifacts.

yaml

Copy code

```
jobs:
  - build:
      script:
        - make
      artifacts:
        paths:
          - binaries/
          - reports/
```


Les artifacts et l'amélioration des performances avec le cache

2. Conservation et Téléchargement :

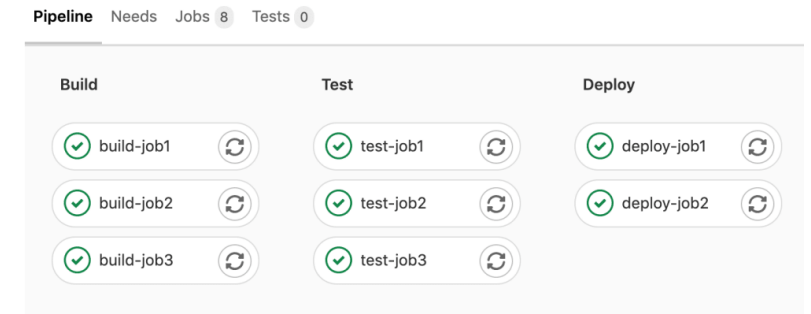
- Les artifacts sont conservés après l'exécution du job et peuvent être téléchargés pour inspection ou utilisation dans d'autres jobs.

```
yaml Copy code

jobs:
  - deploy:
      script:
        - deploy_script.sh
      dependencies:
        - build
      artifacts:
        paths:
          - binaries/
          - reports/
```

3. Utilisation entre Jobs :

- Les artifacts peuvent être utilisés entre différents jobs d'un même pipeline. Par exemple, les fichiers binaires générés lors de la phase de construction peuvent être utilisés dans la phase de déploiement.



Les artifacts et l'amélioration des performances avec le cache

- Le **cache** dans GitLab permet de stocker des fichiers ou des répertoires spécifiques entre différentes exécutions de jobs. Cela peut considérablement réduire le temps d'exécution des jobs en évitant de re-télécharger ou reconstruire des dépendances. Voici comment cela fonctionne :

1. Définition du Cache :

- Vous pouvez spécifier quels fichiers ou répertoires doivent être mis en cache entre les exécutions des jobs.

```
yaml
jobs:
  - build:
      script:
        - make
      cache:
        paths:
          - node_modules/
          - vendor/
```

Copy code

Les artifacts et l'amélioration des performances avec le cache

2. Restauration et Stockage :

- Le cache est automatiquement restauré au début de l'exécution d'un job, et les fichiers modifiés ou ajoutés sont stockés dans le cache à la fin de l'exécution du job.

```
yaml Copy code

jobs:
  - test:
      script:
        - npm install
        - npm test
      cache:
        paths:
          - node_modules/
```

3. Utilisation entre Jobs :

- Comme avec les artifacts, le cache peut être utilisé entre différents jobs d'un même pipeline pour partager des données entre eux.

```
yaml Copy code

jobs:
  - deploy:
      script:
        - deploy_script.sh
      dependencies:
        - build
      cache:
        paths:
          - node_modules/
```

Les artifacts et l'amélioration des performances avec le cache

- Avantages et Considérations :

- **Performance** : Les artifacts et le cache permettent de réduire le temps d'exécution des jobs en évitant de re-générer ou re-télécharger des dépendances.
- **Conservation des résultats** : Les artifacts permettent de conserver les résultats des jobs, tels que les fichiers de construction, les rapports de tests, etc.
- **Réutilisation** : Les artifacts et le cache favorisent la réutilisation des données entre différents jobs, améliorant l'efficacité du processus CI/CD.
- **Considérations sur l'Espace Disque** : Il est important de surveiller l'utilisation de l'espace disque, car une utilisation excessive des artifacts et du cache peut entraîner des problèmes d'espace de stockage.

La documentation officielle relative à la syntaxe du fichier manifeste

La documentation officielle de GitLab relative à la syntaxe du fichier de configuration du pipeline CI/CD, communément appelé fichier **gitlab-ci.yml**, est disponible sur le site web de GitLab. Voici où vous pouvez trouver cette documentation :

1. [Documentation GitLab CI/CD](#):

- La documentation principale pour GitLab CI/CD est accessible à cette URL. Vous y trouverez des informations détaillées sur la configuration du pipeline, les jobs, les étapes, les balises, et bien plus encore.

2. [Page sur la Configuration avec .gitlab-ci.yml](#):

- Cette page spécifique couvre la syntaxe du fichier **gitlab-ci.yml**. Elle explique en détail comment créer et configurer votre fichier de pipeline pour vos projets.

3. [Référence de la configuration .gitlab-ci.yml](#):

- Cette section fournit une référence détaillée de toutes les balises, options et configurations possibles dans le fichier **gitlab-ci.yml**. C'est une ressource utile pour comprendre chaque élément de la syntaxe.

4. [Exemples de Configuration .gitlab-ci.yml](#):

- Sur cette page, vous trouverez des exemples concrets de configurations **gitlab-ci.yml** pour différents types de projets et de scénarios. Cela peut être utile pour avoir des modèles à suivre.

5. [Guide de configuration de base](#):

- Si vous démarrez avec GitLab CI/CD, ce guide de configuration de base fournit des étapes simples pour créer un fichier **gitlab-ci.yml** et configurer un pipeline.

Sommaire

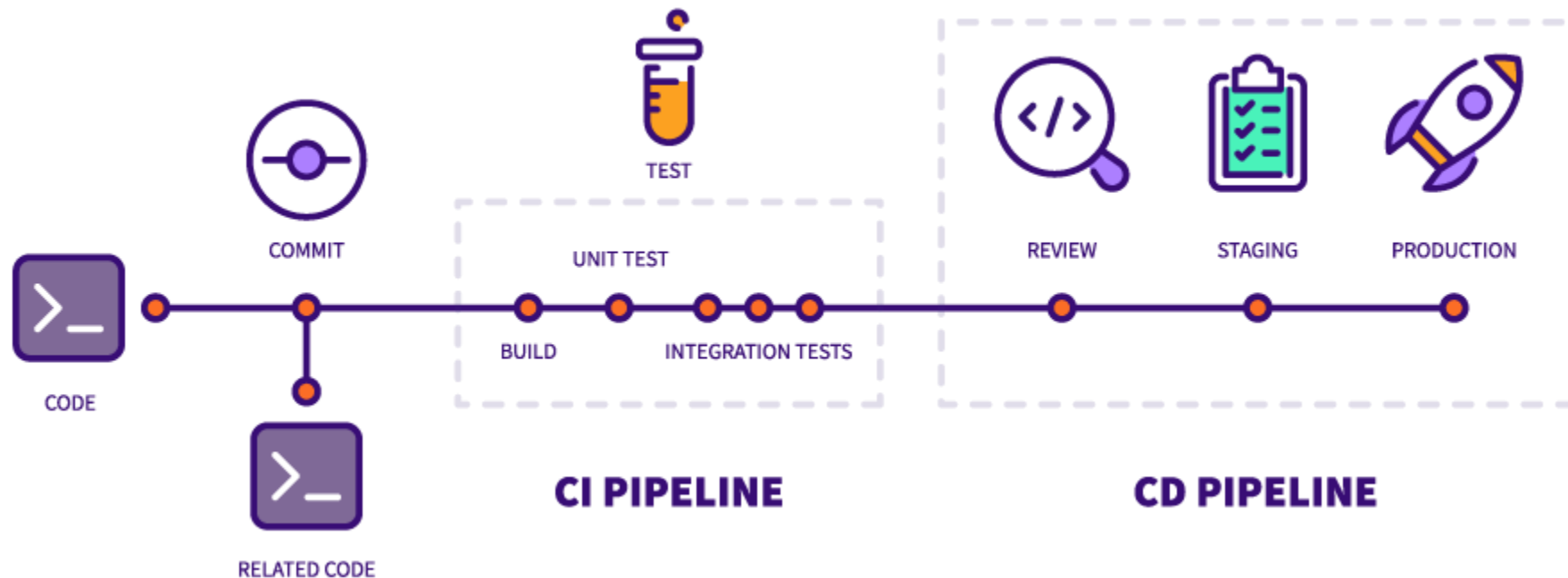
- Introduction
- Gestion du dépôt avec Gitlab
- Gitlab CI/CD
- **La Gestion des environnements**



La gestion des environnements

La gestion des environnements.

- La gestion des environnements dans GitLab est une fonctionnalité qui permet de déployer des applications sur différents environnements (comme le développement, la production, la pré-production (Staging), etc.) et de suivre ces déploiements.



La gestion des environnements.

- Voici comment la gestion des environnements est généralement réalisée dans GitLab :

Configuration du Fichier `.gitlab-ci.yml` :

1. Définition des Environnements :

- Dans le fichier `.gitlab-ci.yml`, vous pouvez définir des jobs qui déploient votre application sur des environnements spécifiques.

yaml

Copy code

```
stages:
  - deploy


deploy_production:
  stage: deploy
  script:
    - deploy_script.sh
  environment:
    name: production
    url: https://production.example.com
```

La gestion des environnements.

2. Utilisation de Balises `environment` :

- L'utilisation de la balise `environment` dans la configuration du job permet de lier le déploiement à un environnement spécifique.

yaml

 Copy code

```
environment:  
  name: production  
  url: https://production.example.com
```

La gestion des environnements.

Suivi des Environnements dans l'Interface Web :

1. Tableau de Bord des Environnements :

- GitLab offre un tableau de bord dédié pour les environnements. Vous pouvez y accéder depuis l'interface web pour voir tous les environnements et leurs états.

2. Suivi des Déploiements :

- Pour chaque environnement, vous pouvez suivre les déploiements associés, voir les commits liés et accéder aux informations détaillées sur l'état du déploiement.

Variables d'Environnement :

1. Variables d'Environnement Définies :

- Vous pouvez définir des variables d'environnement spécifiques à chaque environnement dans GitLab, ce qui permet de paramétrer le comportement de l'application dans différents contextes.

yaml

Copy code

```
deploy_production:
  stage: deploy
  script:
    - deploy_script.sh
  environment:
    name: production
    url: https://production.example.com
  variables:
    DB_PASSWORD: "$PRODUCTION_DB_PASSWORD"
```

La gestion des environnements

Intégration avec les Fonctionnalités GitLab :

1. Review Apps :

- GitLab propose la fonctionnalité des Review Apps qui crée automatiquement des environnements pour chaque merge request, facilitant le test des changements avant la fusion.

2. Suivi des Performances :

- Les performances des environnements peuvent être suivies dans GitLab, y compris des métriques telles que la durée des déploiements et d'autres statistiques.

3. Déploiements Progressifs (Canary Deployments) :

- GitLab prend également en charge les déploiements canari, permettant de déployer progressivement une nouvelle version de l'application sur un sous-ensemble de l'environnement.

Notifications :

1. Notifications d'Environnement :

- GitLab peut être configuré pour envoyer des notifications par e-mail ou via d'autres canaux lorsque des changements sont déployés dans un environnement.

2. Approbations d'Environnement :

- GitLab offre la possibilité de définir des approbations manuelles pour les environnements, ce qui signifie que certaines personnes doivent approuver le déploiement avant qu'il ne soit effectif.

En complément

Fonctionnalités complémentaires de GitLab CE & GITLAB EE

GitLab EE (Enterprise Edition) propose des fonctionnalités supplémentaires par rapport à GitLab CE (Community Edition). GitLab EE est une version payante destinée aux entreprises et aux organisations qui ont des besoins plus avancés en matière de gestion du cycle de vie du développement logiciel. Voici quelques-unes des fonctionnalités que GitLab EE offre en plus par rapport à GitLab CE :

Fonctionnalités complémentaires de GitLab CE & GITLAB EE

- 1. Gestion des autorisations améliorée :** GitLab EE propose des fonctionnalités de gestion des autorisations plus avancées, ce qui est crucial dans les environnements d'entreprise où différents utilisateurs ont des niveaux d'accès différents aux projets et aux dépôts.
- 2. Audit des activités :** GitLab EE inclut des outils d'audit qui permettent de suivre et d'enregistrer les activités des utilisateurs, ce qui est essentiel pour la conformité aux réglementations et pour la sécurité.
- 3. Gestion avancée des déploiements :** Les fonctionnalités de déploiement continu (CI/CD) de GitLab EE sont plus avancées, offrant des options supplémentaires pour l'automatisation des tests, le déploiement sur des environnements complexes, etc.

Fonctionnalités complémentaires de GitLab CE & GITLAB EE

- 1.Assistance technique** : Les utilisateurs de GitLab EE bénéficient d'un support technique plus étendu par rapport à la communauté GitLab CE. Cela peut inclure un support prioritaire, des sessions de formation et d'autres services.
- 2.Intégrations et extensions** : GitLab EE propose parfois des intégrations et des extensions exclusives qui ne sont pas disponibles dans la version CE. Cela peut inclure des intégrations avec des outils tiers, des services cloud, etc.
- 3.Sécurité avancée** : GitLab EE propose des fonctionnalités de sécurité avancées pour aider à protéger les dépôts et les pipelines CI/CD contre les menaces potentielles. Cela peut inclure des outils de détection des vulnérabilités, des fonctionnalités de sécurité avancées pour les déploiements, etc.

Merci pour votre attention

➤ Des questions?

