

GITLAB



- I. Découverte de Gitlab
- II. Gitlab runner
- III. CICD dans Gitlab
- IV. SCRUM dans Gitlab

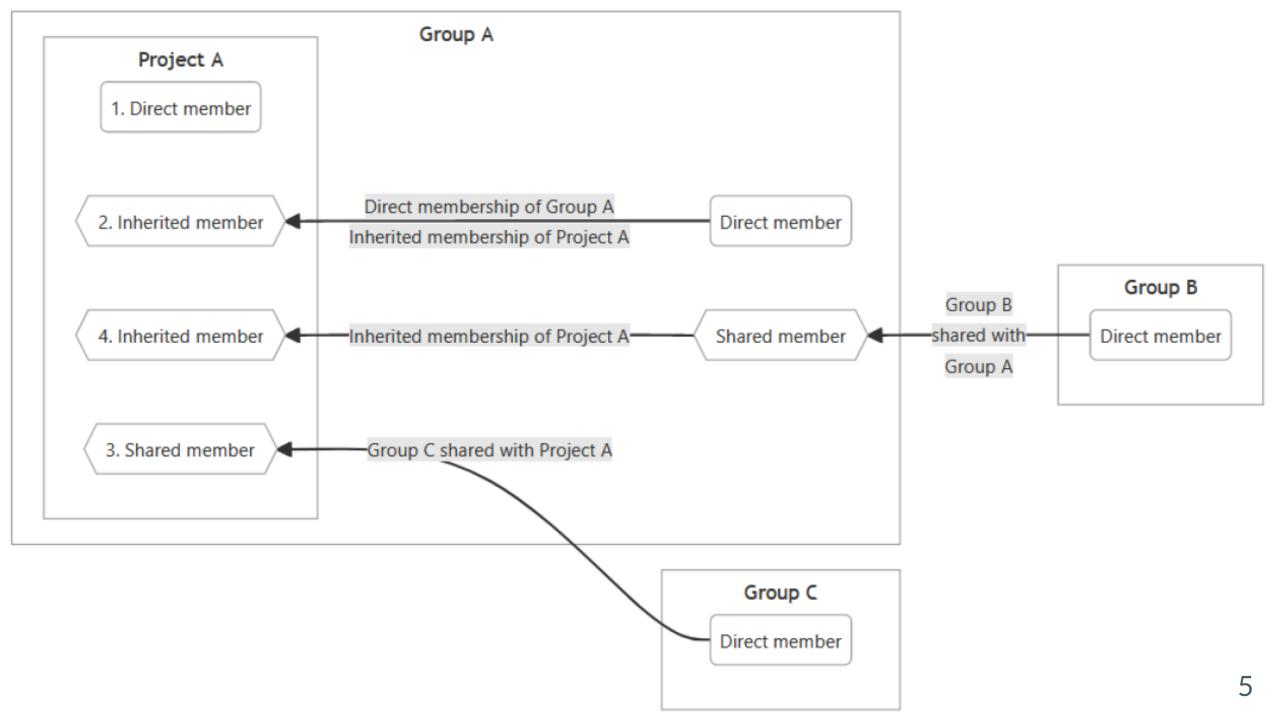


I. Découverte de Gitlab



Concepts organisationnels dans gitlab

- Utilisateurs: peuvent appartenir à un projet ou un groupe
- Projets: gèrent un dépôt git peuvent appartenir à un groupe
- **Groupes:** groupes de *projets*.
- Espaces de noms: Les utilisateurs et les groupes définissent des namespaces pour dédoublonner les noms de projets
 - un projet dans le namespace utilisateur n'est connu que de cet utilisateur : projet personnel
 - deux groupes peuvent définir un projet de même nom : backend/monitoring et frontend/monitoring





droits d'accès

- Owner: créateur d'un projet / groupe tous les droits
- Maintainer: Admin tout sauf suppression + « Settings » projet / groupe
- **Developper:** écriture sur le *code, merge requests, issues, CI / CD*
- Reporter: droits de *commenter* écriture sur *issues, reviews*
- Guest: droit de lecture

ici



accès (à la VM formation)

Login / mot de passe
 Url: https://gitlab.myusine.fr
 Login: root / Mdp: R00tt00R

- Mailhog (compte fake centralisant tous les appels SMTP)
 - Url: http://gitlab.myusine.fr:8025



accès API

- UI: Jeton d'accès à l'API
 - Avatar utilisateur > Preferences > « Acess Token » > api

```
## IDEM AVEC GITLAB-RAILS
sudo gitlab-rails console
> user = User.find_by_username('automation-bot')
> token = user.personal_access_tokens.create(scopes: [:api], name: 'Automation token')
> token.set_token('name-your-own-token')
> token.save!
gitlab-rails runner "user = User.find_by_username;..." ## IDEM en ligne !
## utilisation dans un CLIENT d'API: ex. python
# pip install --upgrade python-gitlab
import gitlab
gl = Gitlab(url="https://<host>", token="name-your-own-token")
```



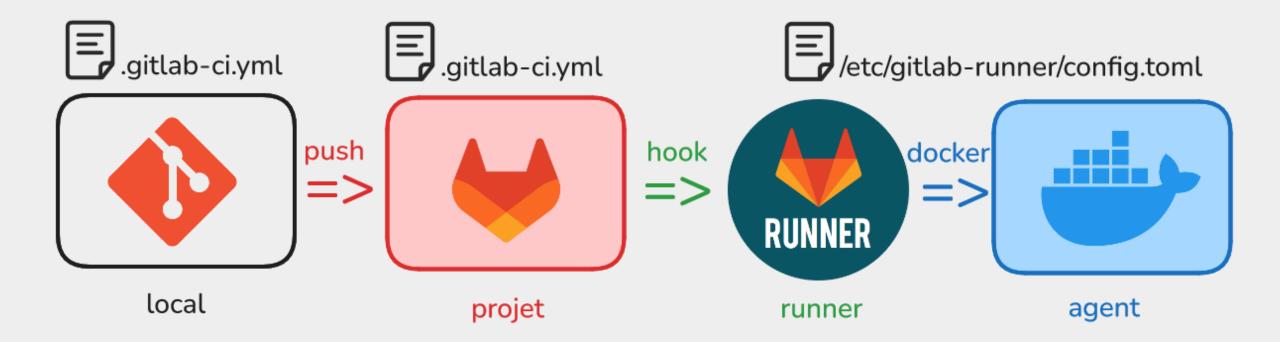
II. Gitlab Runner



présentation

- gitlab-runner composant indépendant de Gitlab chargé du CI/CD
- configure des *processus appelés runners*
- un runner exécute des jobs écrits dans le fichier .gitlab-ci.yml
- Un job dans un agent pouvant être :
 - une machine physisque
 - o une VM Virtualbox, VMWare, Hyper-V...
 - o un conteneur *Docker, LXC, Kubernetes, ...*







configuration

- Un runner est associé à un *projet/groupe* d'instances de gitlab
- projet/groupe => « Settings » => « CICD » => « Runners »
- fichier de configuration: /etc/gitlab-runner/config.toml

```
concurrent = 4 # nb de jobs exécutables en parallèle

[runners.docker]
volumes = [
  "/cache",
  "/etc/gitlab/trusted-certs/<host>.crt:/etc/gitlab-runner/certs/ca.crt:ro",
  "/var/run/docker.sock:/var/run/docker.sock"
]# dns interne au réseau docker pour un gitlab en local
extra_hosts = ["gitlab.myusine.fr:172.17.0.1"]
```



III. CICD dans Gitlab



YAML: introduction

- format de représentation de données par *sérialisation*, conçu pour être *aisément modifiable et lisible*
- Dérivé de la représentation d'un objet JSON déplié



YAML: imbrications

- objets JSON { ... } => indentation de 2 / 4 chars.
- listes JSON [...] => indentation de 2 / 4 chars. + « [espace] »



YAML: alias et ancres

```
struct: &ref ## "&": nomme un objet réutilisable
 k1: v1
 k2: v2
copy:
 my_alias: *ref ## "*": copie exacte du contenu de "struct:"
extends:
 my_anchor:
            ## "<<: *" : copie + ajout ou modifications
   <<: *ref
   k2: v2-bis
   k3: nex-value
```



écrire un job

- Un job est un *objet yaml* dans le fichier **.gitlab-ci.yml**
 - dont le nom est arbitraire
 - contient une liste script[] contenant des lignes de commandes
 - on ajoute une liste de tags[] correspondants aux runners susceptibles d'exécuter le job
 - before_script[]: exécute des commandes préparatoires
 - after_script[]: exécute des commandes de nettoyage

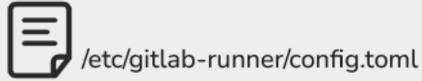


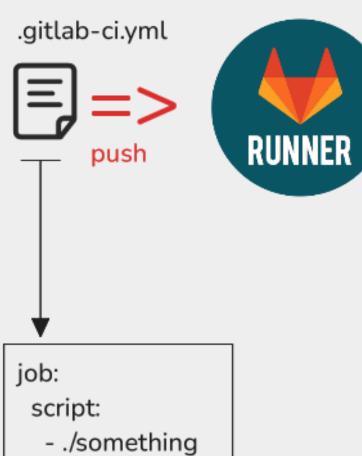
```
unit tests:
  before_script: [ ... ]
  script:
    - echo "launch tests here !"
    - ...
  after_script: [ ... ]
  tags:
    - myusine
    - ...
```

- préfixer le job avec "." => .unit tests: désactive le job
- "Chaque fois que le fichier est poussé sur gitlab, on peut observer l'exécution du job dans la section à gauche CICD => pipelines

99

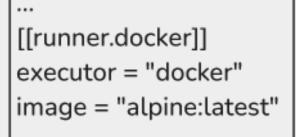
















3° script:

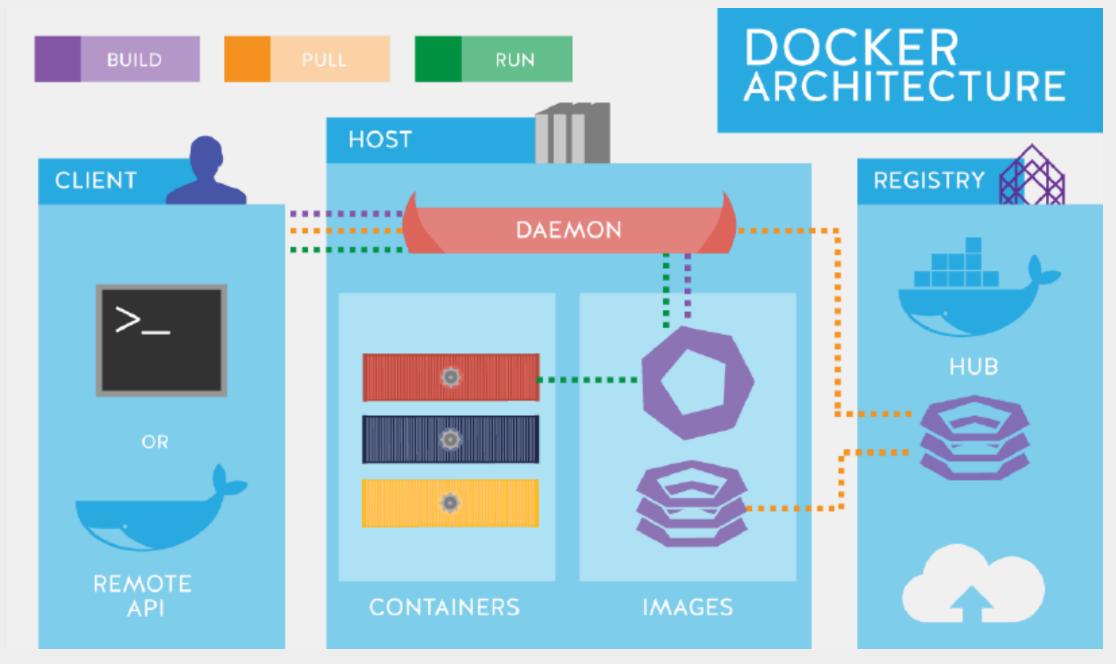




choisir une image (agent docker)

- A l'instar d'une *image disque*, une **image docker** contient l'état d'un système de fichier *instantané / snapshot* i.e librairies et exécutables installés
- c'est un environnement isolé du reste de la machine
- les images viennent a priori d'un registre d'images public sur https://hub.docker.com
- Deploy > Container Registry: registre d'images privé
 - +rapide & images custom.







registre d'images privé

 Ajout d'une image au registre accessible à l'adresse gitlab.myusine.fr:5050

```
# télécharger depuis hub.docker.com
docker pull <image>:<tag>
# copier l'image en préfixant un nouveau nom avec l'URL du projet
docker tag <image>:<tag> gitlab.myusine.fr:5050/myusine/dev/<image>:<tag>
# s'authentifier sur le registre avec les identifiants
docker login gitlab.myusine.fr:5050 -u root -p R00tt00R
# pousser l'image sur le registre
docker push gitlab.myusine.fr:5050/myusine/dev/<image>:<tag>
```



image - mise en oeuvre

- On utilise la clé image: nom_image:nom_tag
 - o nom: techno / tag: version ou environnement
- périmètre (subsidiarité)
 - o runner => /etc/gitlab-runner/config.toml
 - o pipeline / job => dans le fichier .gitlab-ci.yml

```
image: pipeline_image_nom:pipeline_image_tag

job:
   image: job_image_nom:job_image_tag
```



référence et variables prédéfinies

- clés utiles dans gitlab CI : https://docs.gitlab.com/ee/ci/yaml/
- variables d'environnement prédéfinies: ici
- utiliser la valeur des variables, en les préfixant par « \$ »

```
## variables arbitraires créées par le développeur
variables:
   MY_VAR: "here !"

unit tests:
   ## variable prédéfinie créée par Gitlab CI
   image: $CI_REGISTRY_IMAGE/python:rc-slim-buster
   script:
   - echo "launch tests $MY_VAR"
```



variables cachées

- pour les informations sensibles:
 - o secrets: mots de passe, certificats, clés privées
 - adresses IP
 - o ports réseau
- projet/groupe > Settings > CICD > Variables > Add Variable

```
# PKEY, SSH_USER, SSH_HOST dans gitlab cI
script:
- ssh -i $PKEY SSH_USER@SSH_HOST
```



séquencement des jobs

- Par défaut, les jobs sont exécutés en parallèle
 - fonction du nb cpus + config dans /etc/gitlab-runner/config.toml
- un ordre d'exécution des jobs par étapes est créé avec stages[]:
 - o un élément de cette liste est un « stage »
 - les job sont associés aux stages par la clé stage:
- " deux jobs dans un même stage sont exécutés en parallèle



```
stages:
   - testing
   - building

unit tests:
   stage: testing
   image: $CI_REGISTRY_IMAGE/python:rc-slim-buster
   script:
    - echo "launch tests here !"
```



conditionnalité des jobs

- rules[]: liste des règles d'exécution d'un job au moyen de clauses
 - o chaque règle vraie exécute le job OU logique
 - o une règle est vraie si toutes ces clauses sont vraies ET logique
- if: \$VAR [== | != | =~ | !~] "value" [&& | ||]
 - o tester la valeur d'une variable exacte ou liée à une regex
- changes[]: vraie si au un des chemins listés a été modifié exists[]: IDEM pour l'existence
- when: contexte de déclenchement on_success -> par défaut, on_failure, always, never, manual



```
## déclenchement du pipeline
workflow:
  rules:
   - if: $CI_PIPELINE_SOURCE == "push"
    - if: $CI_PIPELINE_SOURCE == "api"
## déclenchement du job
unit tests:
  ## variables d'environnement arbitraires
 variables:
    MANUAL_TRIGGER: "off"
  rules:
    - if: $CI_COMMIT_BRANCH ~= /^feature-[0-9]+/
      changes:
       - README.md
    - if: $MANUAL_TRIGGER == "on"
```



prélever des artefacts

- artifacts: fait remonter toute ressource crée à l'exécution d'un job
 - dans les jobs suivants, pour réutilisation
 - o dans Gitlab UI pour téléchargement
- artefacts:paths[]: chemins à faire remonter
- artefacts:untracked: fait remonter tous les fichiers «U» dans git
- artefacts:expire_in: temps de disponibilité avant suppression par défaut "30 days"



```
unit tests:
    script:
    - echo "content" > artifact.txt
    artifacts:
    paths:
        - ./artifact.txt
    expire_in: "1 hour"
```



faire monter les rapports de jobs

- artifacts:reports: fait remonter des *rapports* dans Gitlab UI en *datavisualisation*
- rapports compatibles avec un format compris par Gitlab
 - o ex: junit: pour les tests unitaires
- Exception: la couverture de code globale peut être prélevée depuis la sortie d'un job avec la clé

```
coverage: "/<regex>/"
```



```
unit tests:
    script:
    - mvn test
    coverage: "/Total.*?([0-9]{1,3})%/"
    artifacts:
        expire_in: "1 hour"
        reports:
            junit: "target/surefire-reports/TEST-*.xml"
```



créer / utiliser le cache

- cache: fait remonter toute ressource crée à l'exécution d'un job
 - o utilisé pour *mettre en cache les dépendances du projet*
- cache: key: nomme les objets mis en cache
- cache:paths[]: désigne les chemins à mettre en cache
- cache:untracked: ajoute au cache les fichiers à l'état «U» dans git
- cache:policy: indique si l'on veut
 - pull => installer le cache existant dans le job
 - push => faire remonter le cache généré dans le job
 - o pull-push => le 2



```
unit tests:
    script:
    - mvn compile
    cache:
    key: cache
    untracked: true
    paths:
     - ./.m2/repository
```



utiliser les services

- **services[]:** configure un *agent auxiliaire* adossé à un job utile quand le job nécessite une *techno client / serveur*
- services[0].name: nom de l'image
- services[0].alias: alias/nom d'hôte réseau pour la cnx client

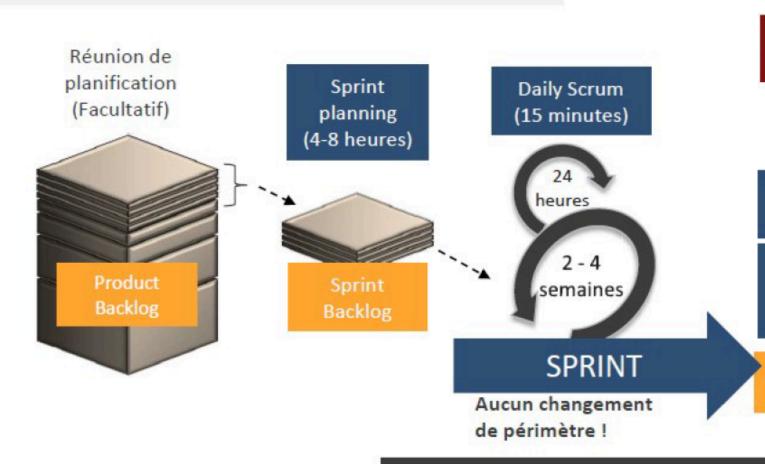
```
job:
  image: client:latest
  script:
    - ./client.sh --host=server-domain-host
  services:
    - name: server:latest
     alias: server-domain-host
```



IV. SCRUM dans Gitlab



Scrum = 3 Rôles + 3 Artefacts + 5 Evénements





Sprint review (2-4 heures)

Sprint retrospective (1,5-3 heures)

Incrément

Scrum est basé sur des itérations de blocs de temps.

Où interviennent les opérations IT ?



rôles dans Scrum

- Product Owner: possède la vision du produit, exprime le besoin
 - observe régulièrement l'avancée du projet
 - au rythme de la diffusion de livrables
 - o il est soit client, soit en interface avec le client
- Agile Process Owner: utilise les principes Agile et Scrum
 - o conçoit, gère et mesurer les processus individuels
- Scrum Master (Dev) et Agile Service Manager (Ops)
 - o garants du cadre méthodologique de l'équipe.
 - veillent à l'ordre du jour et rythment les communications



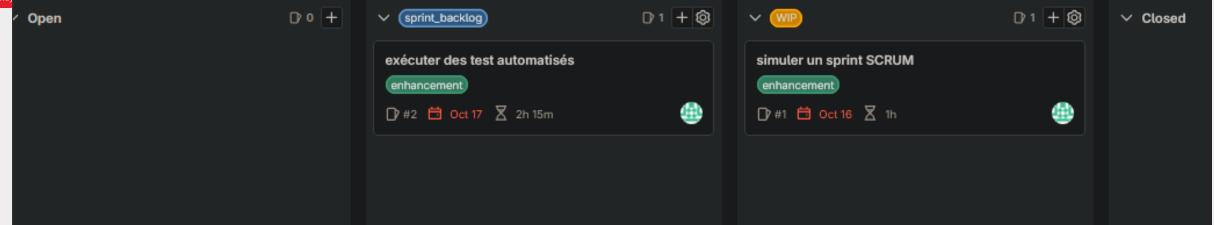
plannification

- 1. Créer des labels

 Projet/Groupe > Manage > Labels
- 2. catégoriser Boards, issues

 Projet/Groupe > Plan > Boards | Issues
- 3. Créer au 2 labels pour les colones sprint backlog et WIP
- 4. Créer des labels classiques Bug, Enhancement, Hot Fix, ...







créer des issues sous forme de « users stories »

- « user story »: description rédigée du point de vue de l'utilisateur
- « epic »: grande user story à décomposer
- critères d'une user story à respacter INVEST
 - Indépendante: sur le sprint en cours
 - Négociable: fait l'objet d'une concertation
 - Valeur: doit apporter de la valeur ajoutée aux clients
 - Estimable: en temps «due date» ou complexité «points de sprint»
 - Suffisamment petite: livraison au sein d'un seul Sprint.
 - Testable: permet d'en déduire un test TDD ou BDD



templates de user story « US »

• templates de description dans .gitlab/issue_templates/xxxx.md OU Projet/groupe > Settings > General > default.md

```
### Given: Context ...
<!-- WHEN? WHERE? WHO? -->
### When: Action To Do ...
* XXXXXX
### Then: Expected Results
<!-- WHAT? Metrics ? -->
<!-- Quick Actions -->
/label ~enhancement # catégorisation depuis la desc.
/estimate ??? # estimation en temps Ex. 1h 30m
```



workflow SCRUM / GITLAB / GIT

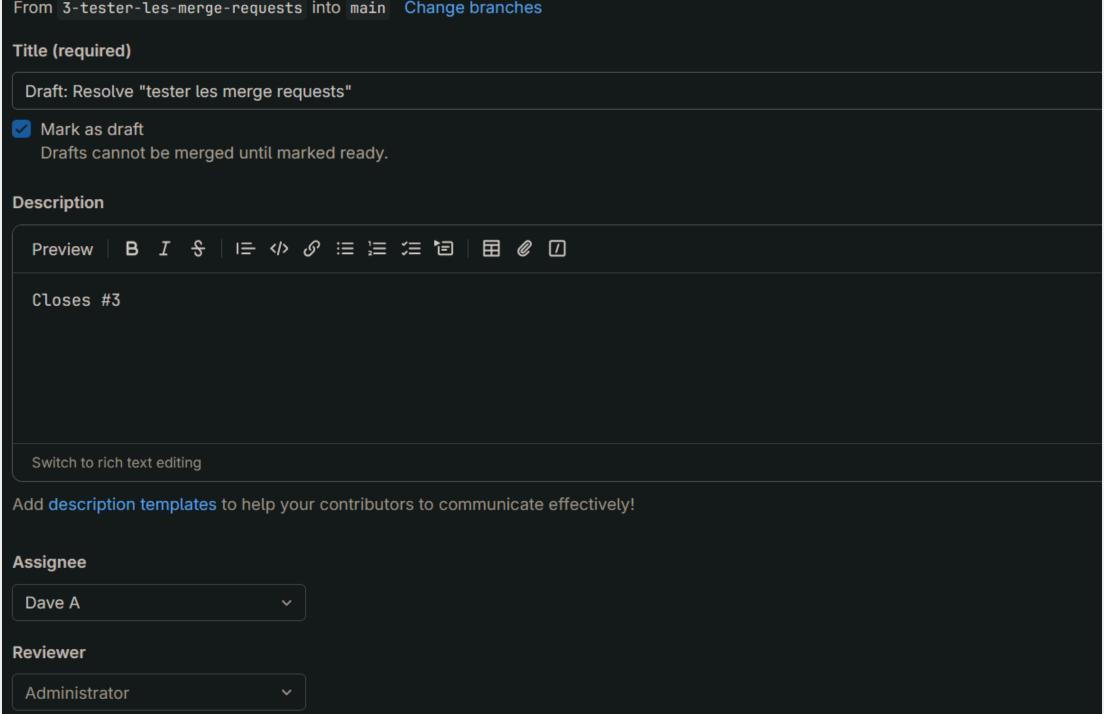
- 1. **Design** créer une **issue** > *liste "Open" de Boards*
 - éditer l'issue pour décrire l'US avec le template
- 2. **Sprint planning** assigner + estimer l'US > *liste "Sprint backlog"*
- 3. **Go** créer une **Merge Request** *MR > liste "WIP"*
 - o bouton "Merge Request" dans l'édition de l'issue
 - o configurer la *MR* et "Créer" > donne une **branche de feature**
 - o rapatrier la branche de feature pour travailler en local



Merge Request MR

- espcace de travail collectif dans gitlab
 - lié à une issue et/ou une branche de fonctionnalité
 - rassemble les informations de travail sur la fonctionnalité modifs, commits, commentaires, pipelines sur la branche
- prépare la fusion de la branche de fonctionnalité dans la branche commune
 - o commence dans l'état « Draft »: non fusionnable
 - on assigne un responsable de la MR
 - on assigne un responsable de la **revue de code**



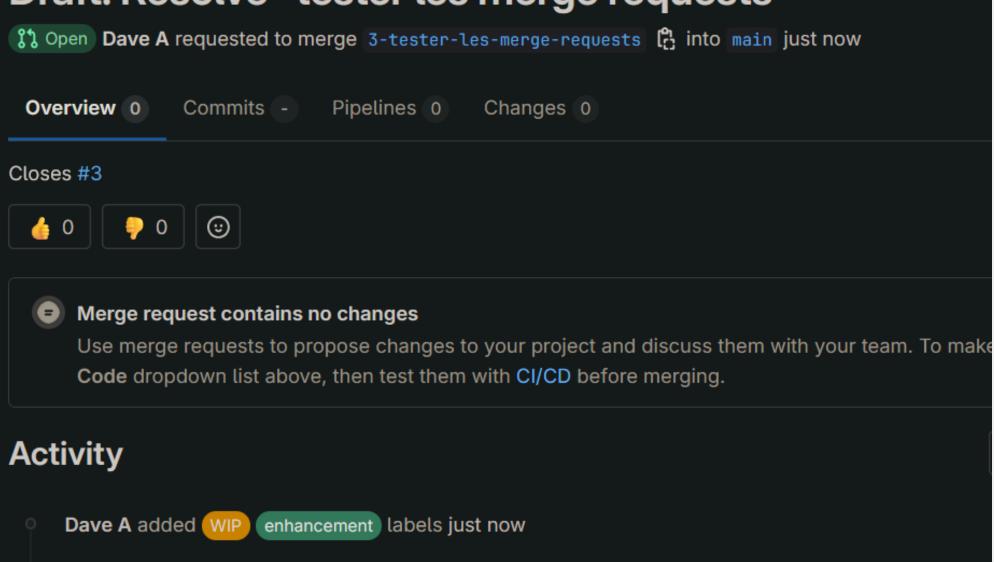




Draft: Resolve "tester les merge requests"

Dave A requested review from @root just now

Dave A assigned to @dev_a just now



47



revue de code par les pairs

- obervations, débat sur le code d'une fonctionnalité ante fusion
- 1. onglet "changes": étude des diffs des divers commits de la branche
- 2. télécharger la branche à fusionner en local et considérer le Δ avec une autre branche de fonctionnalité

```
git fetch && git checkout <distant-hash> ## Mode détaché !!
git checkout -b <tmp-branch> && git merge <local-branch>
## OU appliquer un patch
git checkout <local-hash>
curl -k "https://../merge_requests/xx.patch" | git am
```

3. utiliser un job «Review App», i.e déploiement dynamique de test

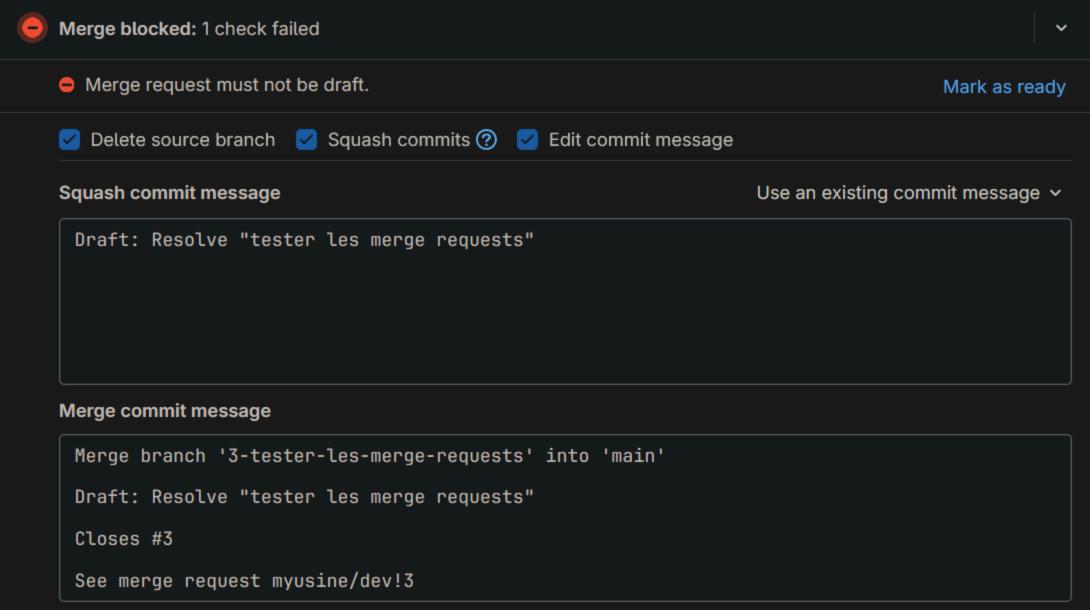


finaliser une MR

- si la revue de code est probante on configure la fusion
- 1. bouton « Mark as Ready » si la MR est à l'état « Draft »
- 2. le pipeline devrait ou doit **être en succès** si Settings > Merge Requests > Merge checks > case cochée
- 3. suppression auto de la branche de fonctionnalité
- 4. **squash** tous les commits de la branche *en un seul commit condensé* + un message de commit condensé
- 5. Si Erreur inverser les modifs du commit de fusion --no-ff

 Revert: git revert -m 1 <merge-commit>







nettoyer le travail en local

```
## télécharger la fusion faite sur gitlab
git checkout main && git pull

## supprimer les branches de fonctionnalité et de suivi inutiles
git branch -d <local-feature-branch>
git fetch --prune
```



