Git/Gitlab

Christophe Dufour

Git

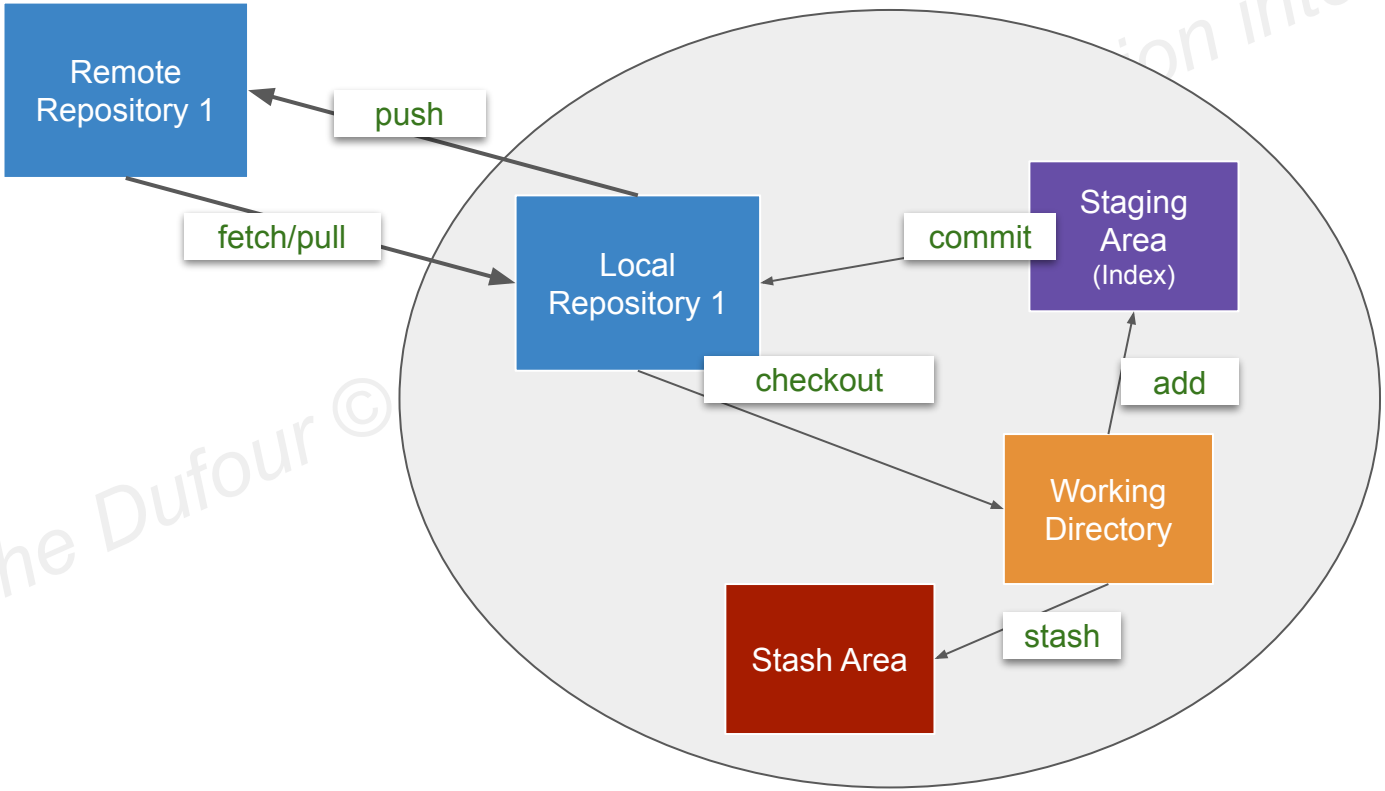
Qu’est-ce que Git ?

- Système de contrôle de version (Version Control System) distribué
  - Permet de mémoriser et retrouver différentes versions d’un projet/fichier
  - Facilite le travail collaboratif
- Créé par T. Linus pour le développement de Linux
- Première version en 2005
- Open Source
- Disponible sur toutes les plateformes

Version Control System (VCS)

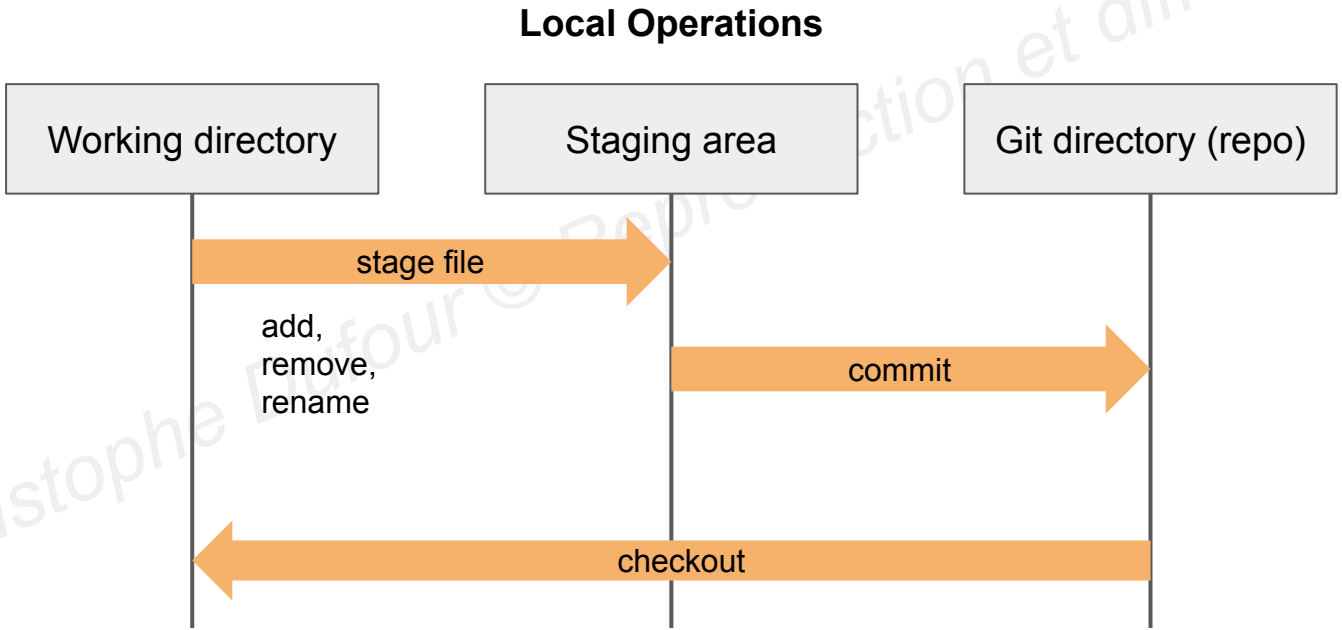
- Définitions simples
  - **Version**: contenu du projet à un moment de son cycle de vie
  - **Dépôt (repository)**: historique du projet, contient toutes ses versions
  - **Branche (branch)**: variante d'un projet
- Pendant la phase de développement d'un projet, il faut:
  - Enregistrer ses versions dans un dépôt afin de pouvoir retrouver une version spécifique (non "buggée" par exemple)
  - Travailler sur différentes variantes (branches) en parallèle, de fusionner ces variantes
  - Partager les modifications faites par différentes développeurs

Schéma général



3 zones locales

GIT dispose de 3 zones: le répertoire de travail, la zone d'index, le répertoire git



Configuration

```
# Paramètre le nom qui sera attaché aux commits et aux tags
$ git config --global user.name “Your Name”

# Paramètre l’adresse email qui sera attachée aux commits et aux tags
$ git config --global user.email “you@example.com”
```

Démarrer un projet

```
# Génère un nouveau dépôt local. Si [project name] est fourni, Git crée un nouveau dossier et
# initialise un dépôt dedans ; sinon, un nouveau dépôt est initialisé dans le dossier courant
$ git init [projet name]

# Télécharge un projet avec son historique complet depuis un dépôt distant
$ git clone [project url]
```

Tâches courantes 1/2

```
# Affiche l'état du working directory
$ git status

# Ajoute un fichier au staging area
$ git add [file]

# Affiche les changements entre le working directory et staging area
$ git diff [file]

# Affiche les changements entre le staging area et le repository
$ git diff --staged [file]

# Supprime les changements dans le working directory. Opération irréversible
$ git checkout [file]

# Rétablit le repository à un état précédent
$ git reset [file]
```

Tâches courantes 2/2

```
# Retire le fichier du working directory et du staging area
$ git rm [file]

# Crée un nouveau commit à partir des changements ajoutés au staging area. Le commit doit être accompagné d'un message
$ git commit -m "descriptive message"

# Stocke les changements du working directory dans le stash pour un usage futur
$ git stash

# Applique dans le working directory les changements stockés puis efface le stash
$ git stash pop

# Supprime le stash
$ git stash drop [stash]
```

Travail avec les branches

```
# Liste toutes les branches du repository. L'option -a inclut également les branches distantes
$ git branch [-a]

# Crée une nouvelle branche à partir de la position courante de HEAD
$ git branch [branch_name]

# Fait basculer le working directory vers le branche spécifiée. L'option -b crée la nouvelle
# branche spécifiée si elle n'existe pas
$ git checkout [-b] [branch_name]

# Fusionne la branche spécifiée dans la branche courante
$ git merge [from_branch_name]

# Supprime la branche spécifiée
$ git branch -d [branch_name]
```

Voir son travail

```
# Liste l'historique des commits de la branche courante. -n count limite la liste aux count derniers commits
$ git log [-n count]

# Vue d'ensemble des commits locaux et distants avec labels et historique graphe. Un commit per ligne
$ git log --oneline --graph --decorate --all

# Liste les commits présents dans la branche courante et non fusionnés dans ref (ref peut être un nom de
# branche ou de tag)
$ git log ref..

# Liste les commits présents dans ref et non fusionnés dans la branche courante
$ git log ..ref

# Liste les opérations (checkouts, commits, etc.) effectuées dans le repository local
$ git reflog
```

Tags

```
# Liste tous les tags
$ git tag

# Crée un tag référencé [name] au niveau du commit courant. [commit sha] permet de spécifier le commit
# servant de base au tag
$ git tag [name] [commit sha]

# Supprime le tag du repository local
$ git tag -d [name]
```

Revenir en arrière (annuler des changements)

```
# Fait basculer la branche courante vers la [target_reference]. Tout changement devient “uncommitted change”. L’option --hard supprime tous les changements
$ git reset [--hard] [target_reference]

# Supprime uniquement les changements apportés par le commit spécifié [commit sha] (sans toucher aux
# commits suivants dans l’historique) et crée un nouveau commit
$ git revert [commit sha]
```

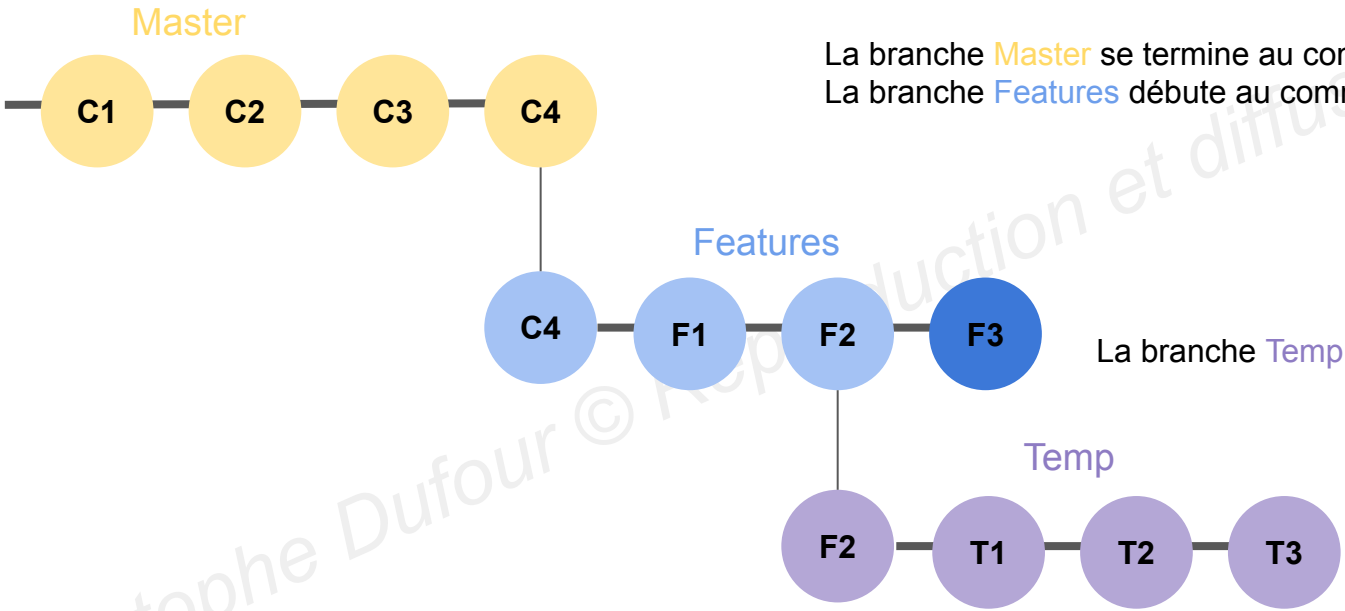
Travailler avec un repo distant

```
# Télécharge les changements depuis le remote mais ne les applique pas localement
$ git fetch [remote]

# Télécharge (fetch) les changements depuis le remote et les fusionne dans la branche courante
$ git pull [remote]

# Envoie les changements locaux sur le remote. --tags pour envoyer les tags également
$ git push [--tags] [remote]

# Envoie le branche locale sur le remote
$ git push -u [remote] [branch]
```

Étapes habituelles pour l'exécution d’un job (tâche)

- Délégation du job à un Gitlab runner
- Téléchargement et démarrage d’une image Docker
- Clonage du dépôt
- Installation des dépendances
- Execution des étapes (.gitlab-ci.yml)
- Enregistrement du résultat (si nécessaire)

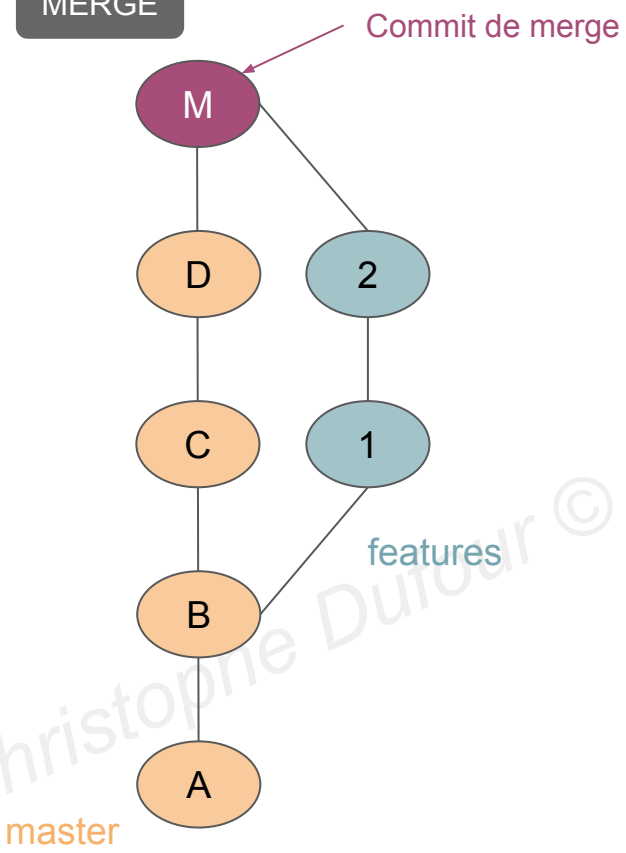
Branches



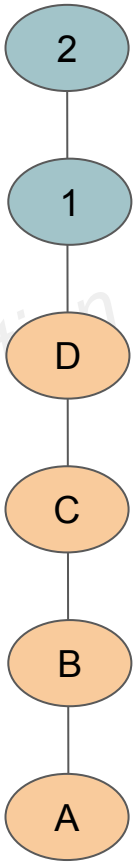
La branche Master se termine au commit C4
La branche Features débute au commit C4

La branche Temp débute au commit F2

MERGE



REBASE



git checkout master
git rebase features
git rebase --continue

Le merge produit un commit de merge après la résolution des conflits

Le rebase produit un historique linéaire après résolution des conflits

Git stash

- Cette commande permet de mettre de côté des changements qu'on n'est pas sûr de vouloir appliquer immédiatement
- Permet de travailler sur une autre branche, faire des commit...
- … et d'appliquer les changements mis en attente ("stashed") ultérieurement
- Commandes:
  - **git stash**: met dans la zone de stash les chagenement
  - **git stash list**: affiche la liste des changements "stashed"
  - **git stash apply**: applique les changements "stashed" dans la branche courante
  - **git stash drop**: supprimer les changements "stashed"

Liens utiles

- https://learngitbranching.js.org/
- https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf
- https://books.goalkicker.com/GitBook/
- https://www.grafikart.fr/formations/git

Gitlab

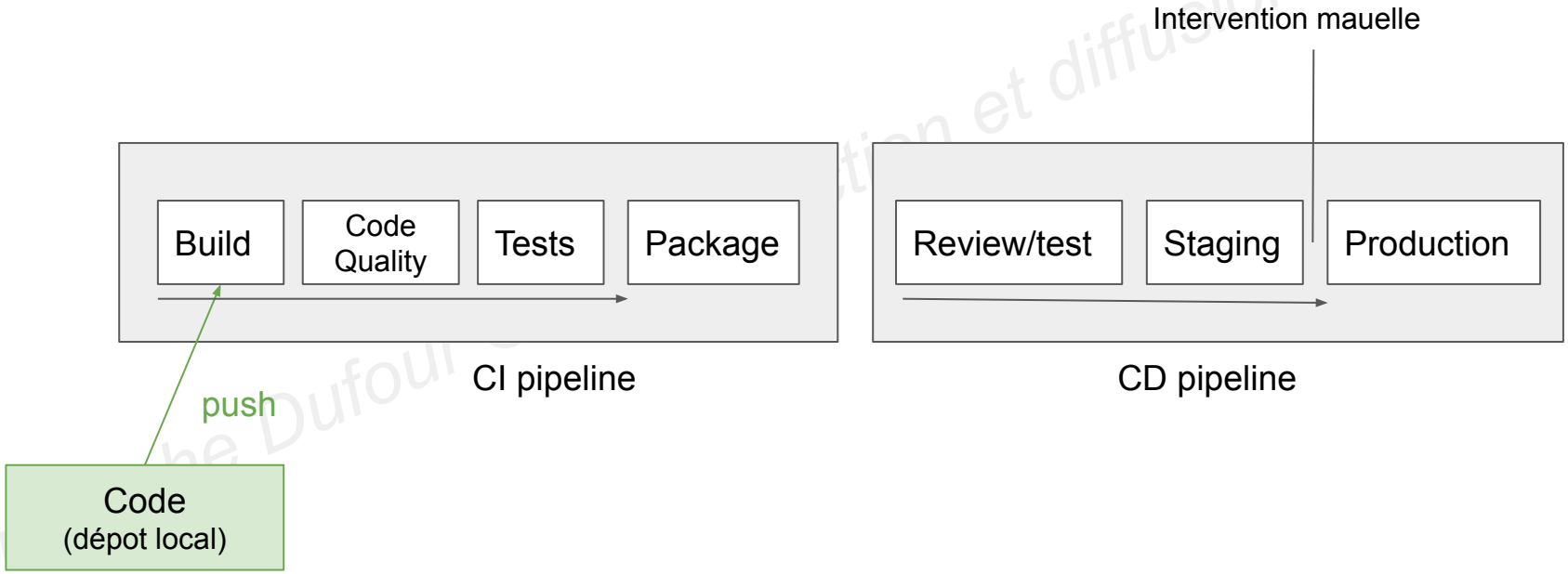
Qu’est-ce que Gitlab ?

- Hôte distant pour dépôts git (comme github)
- Nombreuses fonctionnalités en plus du stockage
  - Wiki
  - Monitoring
  - **CI/CD (fichier .gitlab-ci.yml)**
  - Pages statiques
  - Suvi de bugs
- Première version en 2011
- Open Source

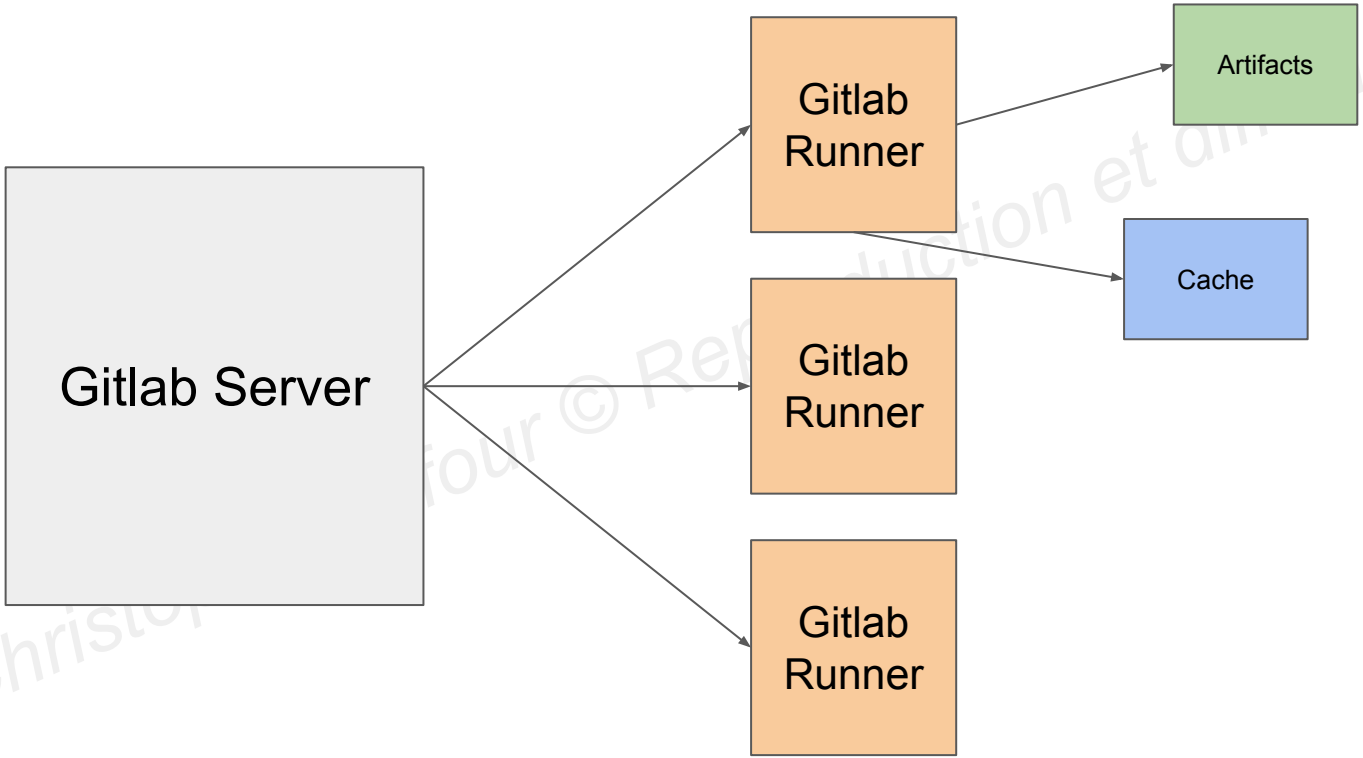
Avantages du CI

- Erreurs détectées tôt dans le processus de développement
- Réduit les problèmes d'intégration
- Permet aux développeurs de travailler plus rapidement
- Assure que chaque changement est "releasable" (peut être déployé)
- Réduit le risque d'un nouveau déploiement
- Fournit de la valeur plus rapidement

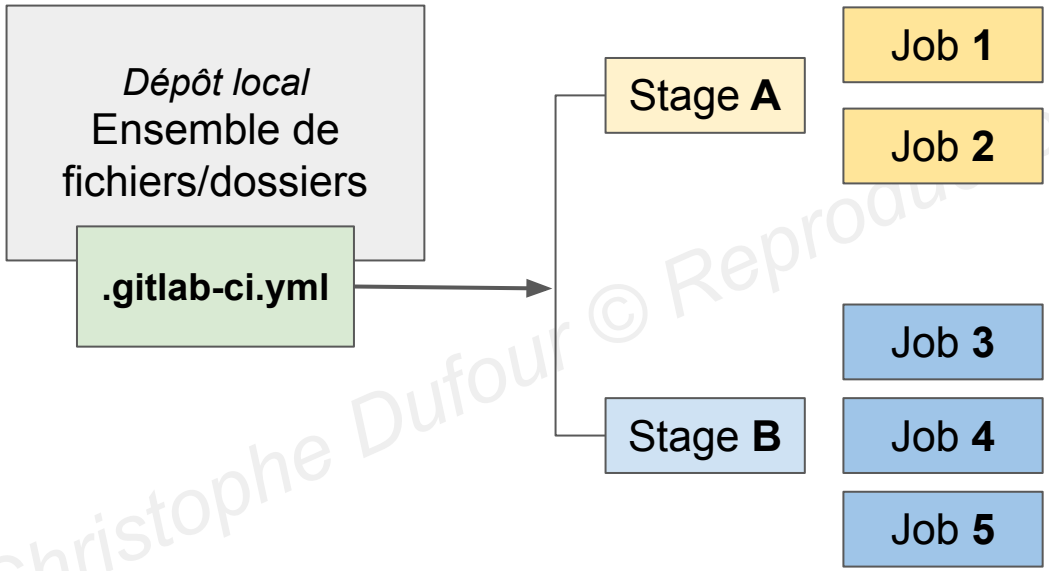
Pipeline classique

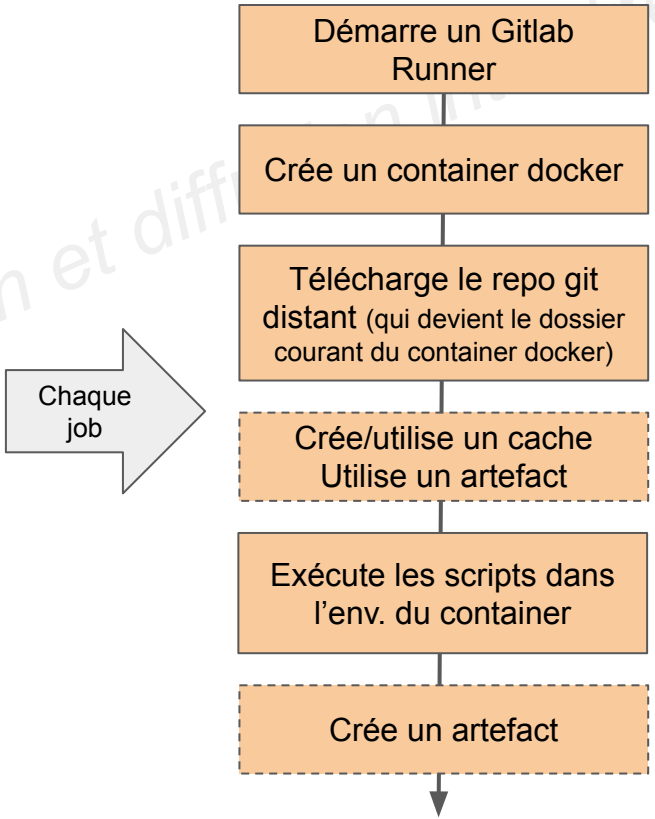


Architecture Gitlab



Fichier .gitlab-ci.yml et workflow





Exemple

.gitlab-ci.yml

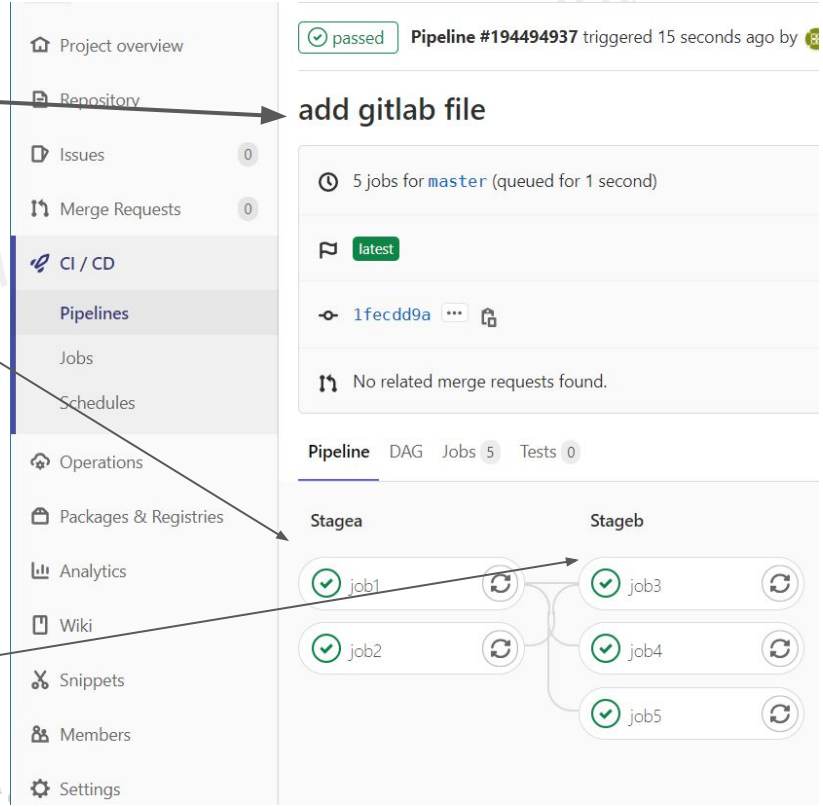
gitlab-presentation > .gitlab-ci.yml

```yaml
stages:
    - stageA
    - stageB

job1:
    stage: stageA
    script:
        - echo "StageA -> job1"
job2:
    stage: stageA
    script:
        - echo "StageA -> job2"
job3:
    stage: stageB
    script:
        - echo "StageB -> job3"
job4:
    stage: stageB
    script:
        - echo "StageB -> job4"
job5:
    stage: stageB
    script:
        - echo "StageB -> job5"
```

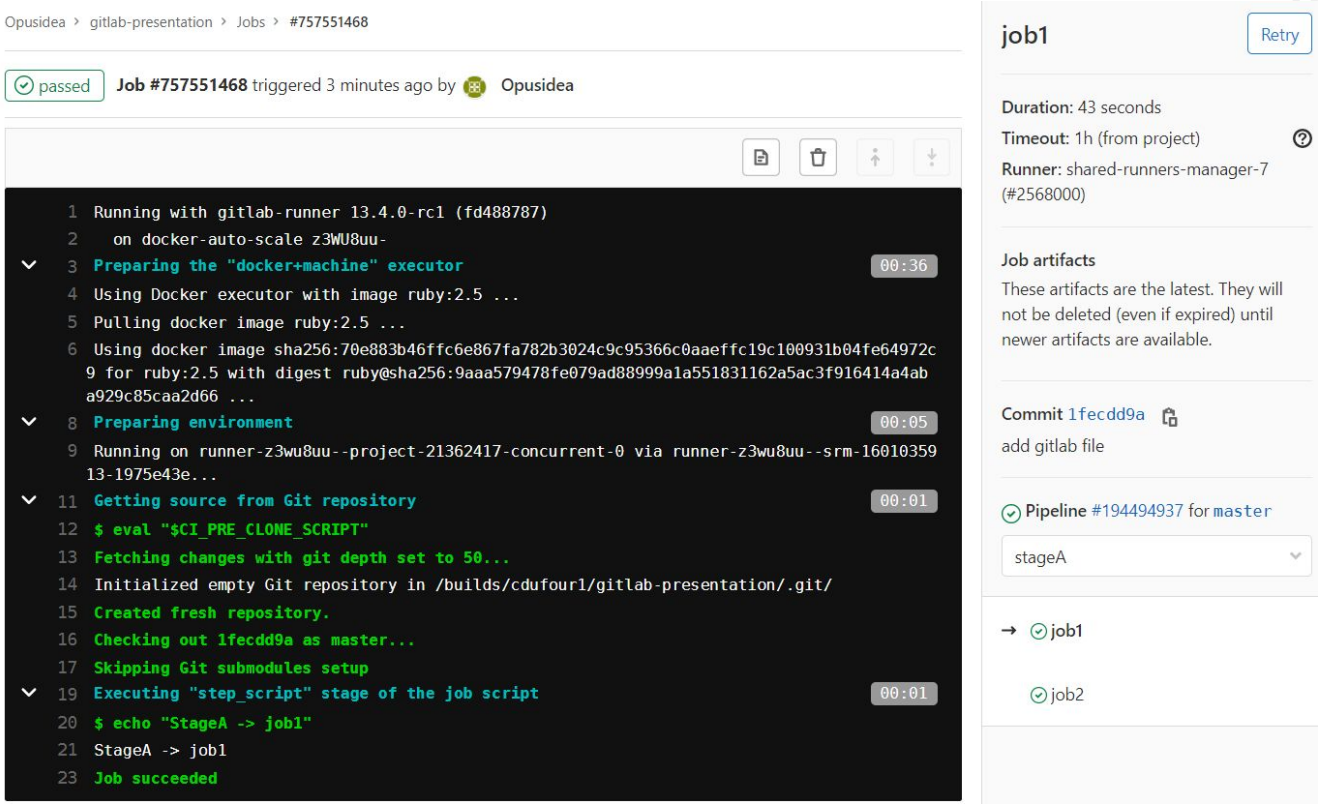
commit/push ayant déclencé la pipeline

Première étape de 2 jobs en parallèle

Deuxième étape de 3 jobs en parallèle

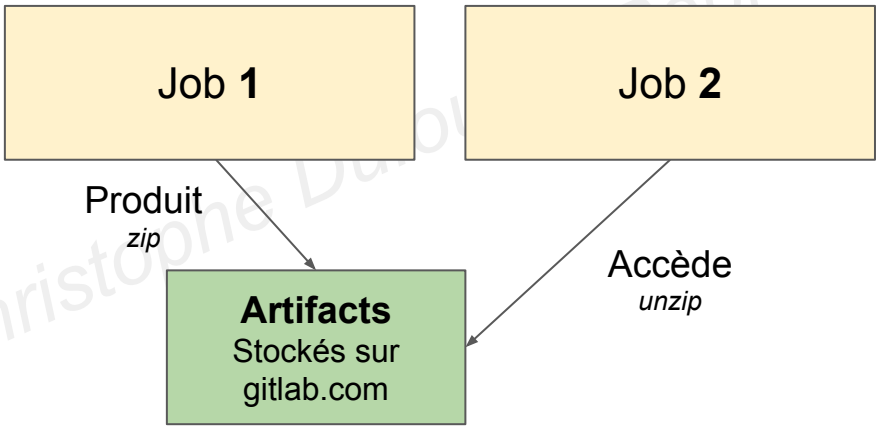


Détail d’un job



Les artefacts (artifacts)

- Fichier ou ensemble de fichiers produits par un job (Gitlab runner)
  - Clé **artifacts.paths:** [“path1”, “path2”, ...”] dans le fichier .gitlab-ci.yml
- Consultables et téléchargeables sur gitlab.com
- Accessibles par les jobs suivants (exemple: tester, déployer le build)
- Moyen de passer le produit (build) d’un job au job suivant



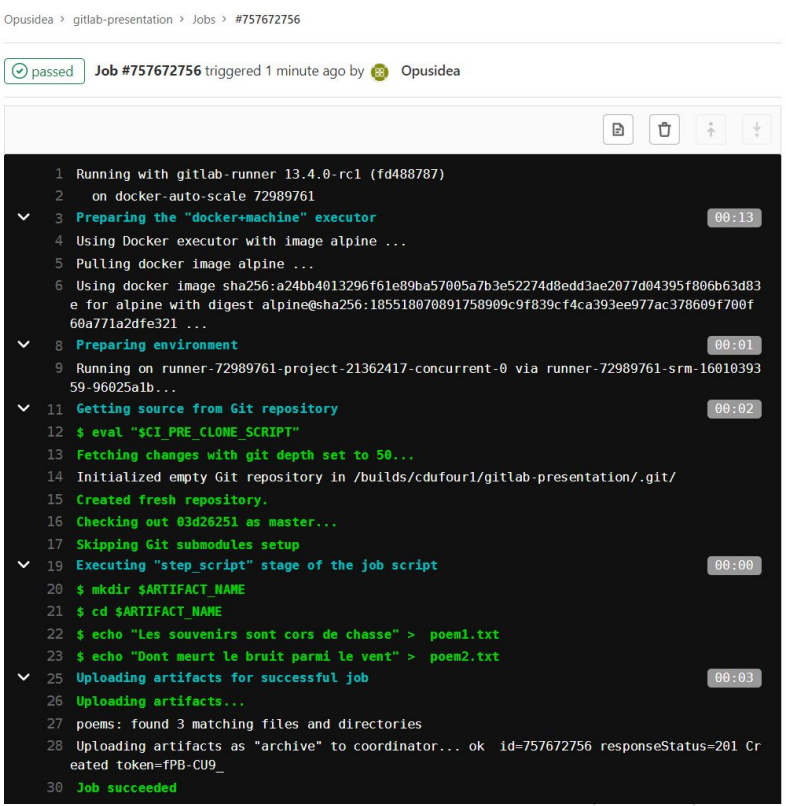
Job produisant un artefact 1/2

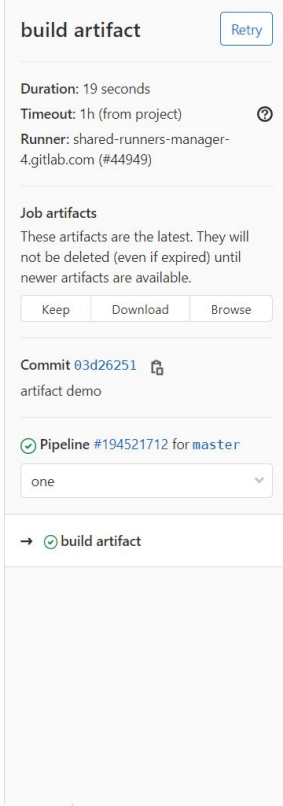
.gitlab-ci.yml

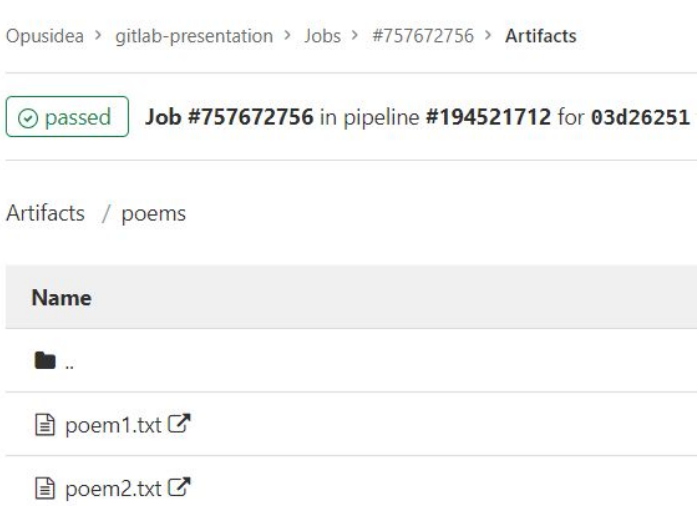
gitlab-presentation > .gitlab-ci.yml

```yaml
stages:
    - one

# variables accessibles à tous les jobs (référencées par un "$"" initial)
variables:
    ARTIFACT_NAME: poems

build artifact:
    image: alpine # image plus lègère que l'image par défaut
    stage: one
    script:
        - mkdir $ARTIFACT_NAME # crée un dossier
        - cd $ARTIFACT_NAME # entre dans le dossier
        - echo "Les souvenirs sont cors de chasse" > poem1.txt # file 1
        - echo "Dont meurt le bruit parmi le vent" > poem2.txt # file 2
    artifacts:
        paths:
            - $ARTIFACT_NAME # artifact = dossier contenant poem1.txt et poem2.txt
```

Job produisant un artefact 2/2







Artefact et jobs 1/2

.gitlab-ci.yml

gitlab-presentation > .gitlab-ci.yml

```yaml
stages:
  - one
  - two

image: alpine # remplace l'image par défault pour tous les jobs

# variables accessibles à tous les jobs (référencées par un "$"" initial)
variables:
  ARTIFACT_NAME: poems

build artifact:
  stage: one
  script:
    - mkdir $ARTIFACT_NAME # crée un dossier
    - cd $ARTIFACT_NAME # entre dans le dossier
    - echo "Les souvenirs sont cors de chasse" > poem1.txt # file 1
    - echo "Dont meurt le bruit parmi le vent" > poem2.txt # file 2
  artifacts:
    paths:
      - $ARTIFACT_NAME # artifact = dossier contenant poem1.txt et poem2.txt

test artifact:
  stage: two
  script:
    # exécution d'un script shell placé à la racine le repo git
    # ce script prend le nom de l'artefact en entrée
    - ./test.sh $ARTIFACT_NAME
```

test.sh

gitlab-presentation > test.sh

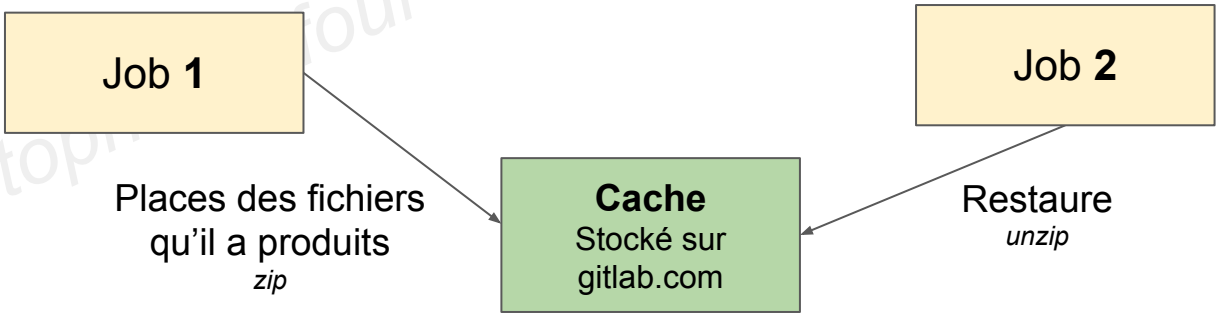
```sh
#!/bin/sh
echo "*** TEST SCRIPT BY CHRIS (OPUSIDEA) ***"

DIR="$1"
FILES="$DIR/*"

if [ -d "$DIR" ]; then
    echo "[+] $DIR folder exists"
    num_files=$(ls $DIR | wc -l)
    echo "[+] Contains $num_files files"

    for f in $FILES
    do
        echo "\t => $f"
    done

else
    echo "[-] $DIR folder do not exist"
fi
```

Artefact et jobs 2/2


```
Running with gitlab-runner 13.4.0-rc1 (fd488787)
  on docker-auto-scale 72989761
Preparing the "docker+machine" executor
Preparing environment                                                    00:01
Getting source from Git repository                                       00:01
Downloading artifacts                                                    00:01
Downloading artifacts for build artifact (757796741)...
Downloading artifacts from coordinator... ok        id=757796741 responseStatus=200 OK to
ken=LX4KEeTE
Executing "step_script" stage of the job script                          00:01
$ ./test.sh $ARTIFACT_NAME
*** TEST SCRIPT BY CHRIS (OPUSIDEA) ***
[+] poems folder exists
[+] Contains 2 files
\t => poems/poem1.txt
\t => poems/poem2.txt
Job succeeded
```

Le cache

- Dans certains cas, optimise le temps d'exécution des runners
- Peut être global ou local (job)
- Si défini globalement, tous les jobs tenteront de le récupérer
- Un job n'ayant pas besoin de récupérer le cache, peut le désactiver (cache: {})
- Ne pas confondre avec les artifacts
- Use case: gestion des dépendances
- Peut être effacé (pipelines/clear cache runners)



Le cache, exemple 1/3

.gitlab-ci.yml

gitlab-presentation > .gitlab-ci.yml

```yaml
stages:
    - step1
    - step2

image: node:alpine # image incluant les programmes node et npm

cache:
    key: myKey # nom de la clé au choix
    paths:
        # mise en cache du dossier node_modules créé par "npm install" (step1)
        - node_modules

install deps:
    stage: step1
    script:
        - npm install # installe les dépendances (express) listés dans package.json

# le job "run server" n'a pas besoin d'installer la dépendance express pour
# démarrer le serveur web, il la trouvera dans le cache
run server:
    stage: step2
    script:
        - npm start & # run server in background mode
```

Le cache, exemple 2/3

```
1  Running with gitlab-runner 13.4.0-rc1 (fd488787)
2    on docker-auto-scale 72989761
3  Preparing the "docker+machine" executor                    00:09
8  Preparing environment                                      00:03
11 Getting source from Git repository                         00:01
19 Restoring cache                                            00:01
20 Checking cache for myKey...
21 FATAL: file does not exist
22 Failed to extract cache
24 Executing "step_script" stage of the job script            00:03
25 $ npm install
26 npm WARN gitlab-presentation@1.0.0 No description
27 added 50 packages from 37 contributors and audited 50 packages in 1.531s
28 found 0 vulnerabilities
30 Saving cache                                               00:01
31 Creating cache myKey...
32 node_modules: found 398 matching files and directories
33 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/21
   362417/myKey
34 Created cache
36 Job succeeded
```

4.gitlab.com (#44949)

Job artifacts

These artifacts are the latest. They will not be deleted (even if expired) until newer artifacts are available.

Commit e3934cae

add cache-demo branc

Pipeline #194583067 for cache-demo

step1

→ install deps

Le cache, exemple 3/3

```
1  Running with gitlab-runner 13.4.0-rc1 (fd488787)
2    on docker-auto-scale z3WU8uu-
3  Preparing the "docker+machine" executor                    00:15
8  Preparing environment                                      00:02
11 Getting source from Git repository                         00:02
19 Restoring cache                                            00:01
20 Checking cache for myKey...
21 Downloading cache.zip from https://storage.googleapis.com/gitlab-com-runners-cache/project/21362417/myKey
22 Successfully extracted cache
24 Executing "step_script" stage of the job script            00:00
25 $ npm start &
27 Saving cache                                               00:01
28 Creating cache myKey...
29 node_modules: found 398 matching files and directories
30 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/21362417/myKey
31 Created cache
33 Job succeeded
```

(#2568000)

Job artifacts

These artifacts are the latest. They will not be deleted (even if expired) until newer artifacts are available.

Commit 11d5de9e

add comments

Pipeline #194586409 for cache-demo

step2

→ run server

Le cache, erreur en case de désactivation

```
  on docker-auto-scale fa6cab46
Preparing the "docker+machine" executor                    00:16
Preparing environment                                      00:02
Getting source from Git repository                         00:02
Executing "step_script" stage of the job script            00:01
$ npm start
> gitlab-presentation@1.0.0 start /builds/cdufour1/gitlab-presentation
> node server.js
internal/modules/cjs/loader.js:896
  throw err;
  ^
Error: Cannot find module 'express'
Require stack:
- /builds/cdufour1/gitlab-presentation/server.js
    at Function.Module._resolveFilename (internal/modules/cjs/loader.js:893:15)
    at Function.Module._load (internal/modules/cjs/loader.js:743:27)
    at Module.require (internal/modules/cjs/loader.js:965:19)
    at require (internal/modules/cjs/helpers.js:88:18)
    at Object.<anonymous> (/builds/cdufour1/gitlab-presentation/server.js:1:17)
    at Module._compile (internal/modules/cjs/loader.js:1076:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1097:10)
    at Module.load (internal/modules/cjs/loader.js:941:32)
    at Function.Module._load (internal/modules/cjs/loader.js:782:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:72:12) {
  code: 'MODULE_NOT_FOUND',
  requireStack: [ '/builds/cdufour1/gitlab-presentation/server.js' ]
}
npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! gitlab-presentation@1.0.0 start: `node server.js`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the gitlab-presentation@1.0.0 start script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.
npm WARN Local package.json exists, but node_modules missing, did you mean to install?
npm ERR! A complete log of this run can be found in:
npm ERR!     /root/.npm/_logs/2020-09-25T16_48_39_112Z-debug.log
ERROR: Job failed: exit code 1
```

New issue

Duration: 21 seconds
Timeout: 1h (from project)
Runner: shared-runners-manager-3.gitlab.com (#44028)

Job artifacts
These artifacts are the latest. They will not be deleted (even if expired) until newer artifacts are available.

Commit d285b2af
disable cache for step2

Pipeline #194622440 for cache-demo

step2

→ run server

disable cache for step2

parent 2c883551 cache-demo

No related merge requests found

Pipeline #194622440 failed with stages in 1 minute and 26 seconds

Changes 1 Pipelines 1

Showing 1 changed file with 1 addition and 1 deletion

.gitlab-ci.yml

```
...  ...  @@ -20,5 +20,5 @@ install deps:
20   20   run server:
21   21       stage: step2
22   22       script:
23        -         - npm start & # run server in background mode
     23   +         - npm start  # run server
24   24       cache: {}
```