Environnement de test

Florent Pruvost (modifié par Pierre Ramet)

September 20, 2026

Contents

- Bien contrôler son environnement de test :
  - le construire et maintenir la recette,
  - déployer l'image servant aux tests,
  - automatiser la construction et le déploiement via gitlab-ci.
- Réfléchir à la notion d'images vs. machine bare-metal qu'il faut réinstaller à chaque fois :
  - tester le déploiement du même environnement sur plusieurs runners,

  - discuter du choix de réinstaller toutes les dépendances ou non selon le contexte (si trop lourd/long à redéployer, il vaut mieux maintenir une image préconstruite).

- Quand et comment (différentes stratégies) mettre à jour l'environnement de test ?

1 Introduction à GitLab CI et aux runners

- Voir le cours sur l'intégration continue avec GitLab (`https://ramet.gitlab.io/r4.02-qualite-dev/gitlab-ci.html`)
- Voir également le cours sur l'orchestration de conteneurs avec docker/podman (`https://moodle.u-bordeaux.fr/course/view.php?id=5358`)
  - on a vu comment construire une image podman contenant un runner GitLab configuré en mode shell
  - on va voir comment configurer un runner GitLab en mode docker, mais configuré nativement pour utiliser podman
- Mettre en place des runners de type shell et docker
- Tester avec un job dans les 2 configurations

1.1 Forker le projet Heat, accès à l'API GitLab avec un jeton personnel

1. Forker le projet Heat sur le GitLab de l'IUT. Il devrait être visible à cette URL : `https://gitlab-ce.iut.u-bordeaux.fr/\protect\T1\textdollarUSER/heat`. Ajouter *fpruvost* et *ramet* comme "Maintainer" à votre projet via Manage -> Members -> Invite members. Si vous aviez déjà un fork de heat, le supprimer via Settings -> General -> Advanced -> Delete project.
2. Activer les pipelines (CI/CD) dans les paramètres (Settings -> General -> Visibility, project features, permissions) de votre projet Heat.
3. Désactiver les "Instance runners" : dans Settings -> CI/CD -> Runners -> Instance runners, désactiver l'option *Enable instance runners for this project.*
4. Créer un jeton d'accès personnel (**Personal Access Token**) (Preferences -> Access Tokens -> Add new token) sur le GitLab de l'IUT avec accès **api**, puis le sauvegarder localement :

```
echo "le token avec acces complet api" > ~/.gitlabtoken
export TOKEN=‘cat ~/.gitlabtoken‘
# vérifier que vous pouvez cloner votre dépôt heat
git clone https://$USER:$TOKEN@gitlab-ce.iut.u-bordeaux.fr/$USER/heat.git
cd heat
```

   Note : Ce jeton est personnel et doit être gardé secret, il permet de vous authentifier sur GitLab. Il inclut les droits de lecture/écriture sur le registre GitLab. Il n'est pas nécessaire pour cloner votre projet si vous avez déjà configuré une clé SSH sur votre compte GitLab.

   Bonnes pratiques professionnelles : gestion des secrets et des jetons :

   - **Jamais de secret dans Git** : Ne commitez **jamais** de fichier contenant un jeton d'accès ou un mot de passe (pensez à ajouter `/.gitlabtoken` ou les fichiers d'environnement dans votre `.gitignore`).
   - **Variables CI/CD sécurisées** : Dans un pipeline GitLab CI, on ne stocke pas de jeton dans les fichiers sources. On utilise les variables de projet (**Settings -> CI/CD -> Variables**) en cochant impérativement les options :
     - **Masked** : empêche la valeur du secret d'apparaître en clair dans les logs d'exécution des jobs.
     - **Protected** : restreint l'exposition de la variable aux seules branches ou tags protégés (ex. `master` ou tags de release).
   - **Jeton temporaire de job** : Pour les opérations au sein du pipeline (téléversement de paquet, pull d'image de registre), privilégiez la variable prédéfinie `$CI_JOB_TOKEN` qui est générée à la volée par GitLab, valide uniquement pendant la durée du job, et avec des privilèges restreints.
   - **Détection automatique de fuites** : GitLab propose le template officiel `Security/Secret-Detection.gitl` qui analyse automatiquement chaque commit pour détecter si des clés privées, mots de passe ou jetons d'API ont été accidentellement ajoutés au code source.

1.2 Installation de gitlab-runner

Installer une version récente de gitlab-runner (>=16.0) :

```
cd $HOME
curl -L --output ./gitlab-runner "https://s3.dualstack.us-east-1.amazonaws.com/gitlab-runner-downloads/
chmod +x ./gitlab-runner
export PATH=$PATH:$HOME
# puis démarrer le programme gitlab-runner
~/gitlab-runner run
# laissez-le tourner, ouvrez un nouvel onglet dans le terminal pour la suite
```

Pour le moment, cela renvoie une erreur car le fichier de configuration `config.toml` n'existe pas encore. Il faut faire un premier enregistrement de runner pour le créer automatiquement.

1.3 Runner de type shell

Enregistrer un runner de type executor *shell* via : Settings -> CI/CD -> Runners -> Create project runner. Remplir les champs **Tags** "shell" et **Runner description** "IUT shell mode", puis valider **Create runner**. Ensuite, sélectionner **Operating system** : Linux. Puis copier/coller la commande `gitlab-runner register` proposée dans votre terminal. Exemple :

```
~/gitlab-runner register  --url https://gitlab-ce.iut.u-bordeaux.fr  --token glrt-iMzkBpbwCRpJ-vysaXA4
```

Suivre les étapes en tapant sur la touche Entrée deux fois, puis à la troisième étape choisir l'executor : **shell**.

> **Note** : Remarquez que le token du runner est spécifique à chaque projet et doit être gardé secret (il est également différent de celui utilisé pour l'API GitLab).

Puis redémarrer gitlab-runner :

```
~/gitlab-runner run
```

Il doit bien s'exécuter sans erreurs cette fois.

> **Note** : Si `~/gitlab-runner run` renvoie des erreurs, vérifiez votre fichier `~/.gitlab-runner/config.toml` car il pourrait comporter d'anciennes configurations de runners (années précédentes). Dans ce cas, supprimez les lignes correspondant aux anciens runners et redémarrez `~/gitlab-runner run`.

À cette étape, vous devriez voir votre runner associé à votre projet GitLab dans la section Settings -> CI/CD -> Runners -> Project runners :

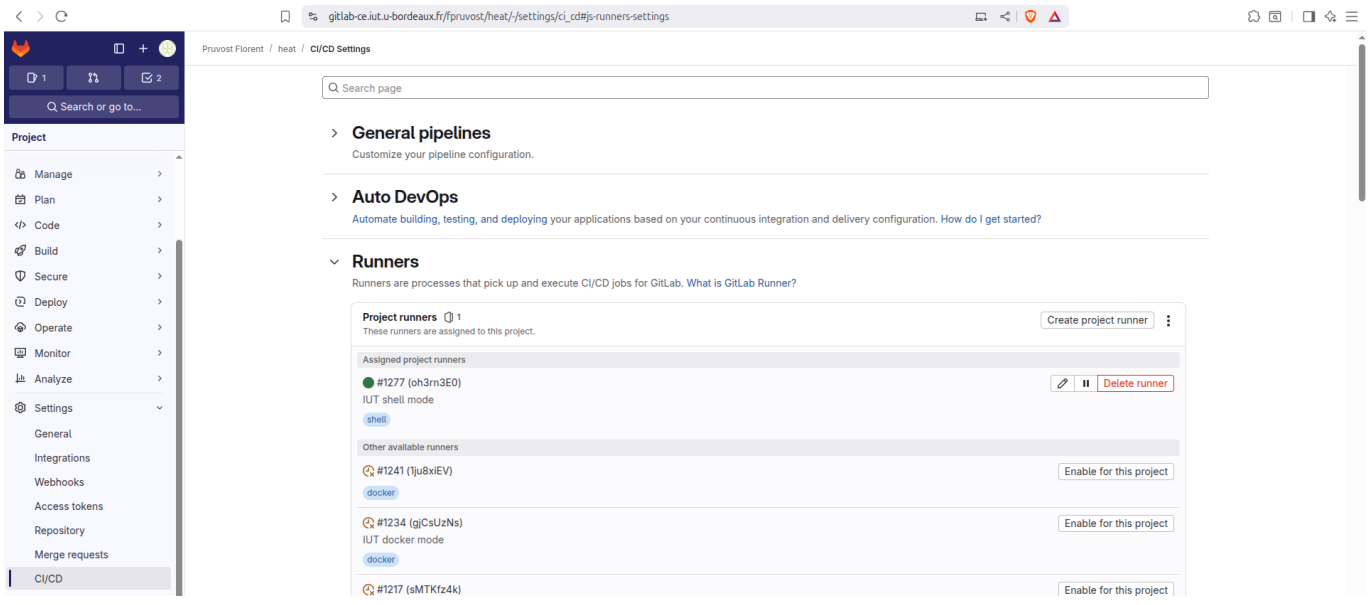

Figure 1: gitlab runner shell

Dans un terminal shell, naviguer dans votre fork de Heat, `cd heat`, créer une branche git **'auto'**, se positionner dessus, puis ajouter le fichier `.gitlab-ci.yml` permettant de lancer un job sur chaque runner, par exemple pour juste vérifier que le job se lance bien :

```
job-shell:
  tags: ['shell']
  script: ls -la
```

Pour voir les logs des jobs gitlab-ci, il faut se rendre sur la page Build -> Pipelines puis cliquer sur la pastille du pipeline (dans la colonne Status) ou sur la pastille d'un job en particulier (colonne Stages).

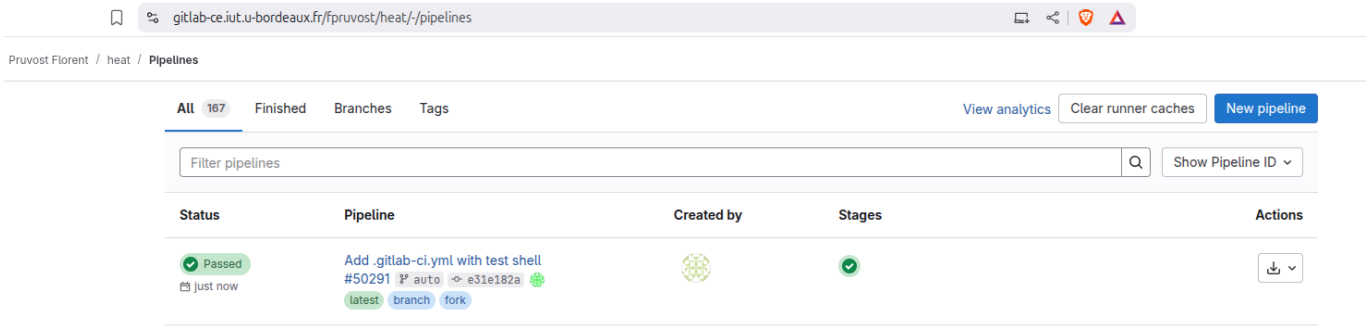

Figure 2: 1er pipeline

L'erreur suivante peut apparaître :

```
ERROR: Job failed: prepare environment: exit status 1. Check https://docs.gitlab.com/runner/shells/inde
```

Pour régler le problème, ouvrir le fichier `.bash_logout` à la racine de votre home, puis commenter la section suivante :

```
#if [ "$SHLVL" = 1 ]; then
#    [ -x /usr/bin/clear_console ] && /usr/bin/clear_console -q
#fi
```

Vous pouvez relancer le job en cliquant sur le bouton "Retry all failed or cancelled jobs" sur la page des pipelines sur la droite ou en cliquant directement sur le job en échec puis sur le bouton "Run again".

Le job doit passer au vert et les logs doivent afficher le résultat des commandes inscrites dans la section script du job :

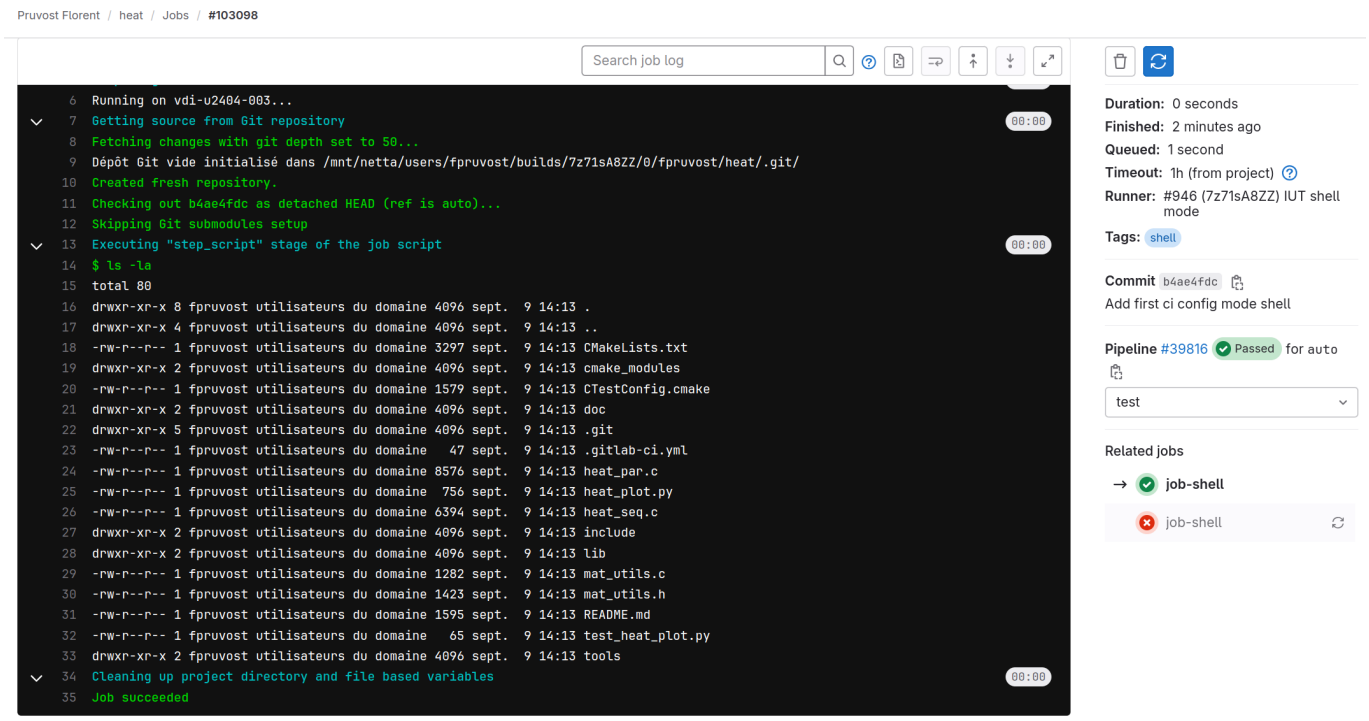

Figure 3: job shell

Note : Remarquez que GitLab clone automatiquement le projet sur la bonne branche 'auto' et vous positionne directement dans le dossier *heat*. Remarquez aussi que dans la colonne de droite, le champ **Runner** donne le nom du runner sur lequel le job s'est exécuté.

En mode *shell*, les jobs sont directement lancés dans votre environnement utilisateur tandis qu'en mode *docker*, les jobs sont lancés dans un conteneur d'une image docker (ou podman). Comme l'environnement *shell* n'est pas facilement configurable, car vous n'êtes pas administrateur sur la session Ubuntu, **on travaillera exclusivement à partir d'une image docker/podman en mode conteneur par la suite**.

1.4 Création du service gitlab-runner

À la place de `gitlab-runner run`, gitlab-runner peut aussi être démarré comme un service. L'avantage sera d'avoir les runners GitLab automatiquement disponibles dans vos prochaines sessions Ubuntu.

Arrêtez votre `gitlab-runner run` en cours avec Ctrl+C.

Voici les commandes pour créer un service gitlab-runner utilisateur :

```
mkdir -p ~/.config/systemd/user/

# copier le template vers ~/.config/systemd/user
curl https://automatisation-ramet-09f19b71cd035def286edcb4a35616cad4167379e2.gitlab.io/gitlab-runner.se

sed -i -e "s#\$HOME#$HOME#g" ~/.config/systemd/user/gitlab-runner.service

systemctl --user daemon-reload
systemctl --user enable --now gitlab-runner.service

# vérifier que le service gitlab-runner est bien actif avec
systemctl --user status gitlab-runner.service
```

> Pour exécuter toutes ces commandes d'un coup vous pouvez utiliser le script config-gitlab.sh de la manière suivante :
>
> ```
> source config-gitlab.sh
> ```

Cela va créer le fichier `~/.config/systemd/user/gitlab-runner.service` à partir de gitlab-runner.service, remplacer `$HOME` par votre répertoire home (cf. `echo $HOME`) puis activer le service gitlab-runner.

1.5 Runner de type docker

Enregistrer un nouveau runner de type executor *docker* via : Settings -> CI/CD -> Runners -> Create project runner. Remplir les champs **Tags** "docker" et **Runner description** "IUT docker mode", puis valider **Create runner**. Ensuite, sélectionner **Operating system** : Linux. Puis copier/coller la commande `gitlab-runner register` proposée dans votre terminal. Exemple :

```
gitlab-runner register  --url https://gitlab-ce.iut.u-bordeaux.fr  --token glrt-iMzkBpbwCRpJ-vysaXA4
```

Suivre les étapes en tapant sur la touche Entrée et indiquer executor **docker** et image par défaut : `registry.u-bordeaux.fr/`
Un nouveau runner est créé et visible sur la page Settings -> CI/CD -> Runners -> Project runners.

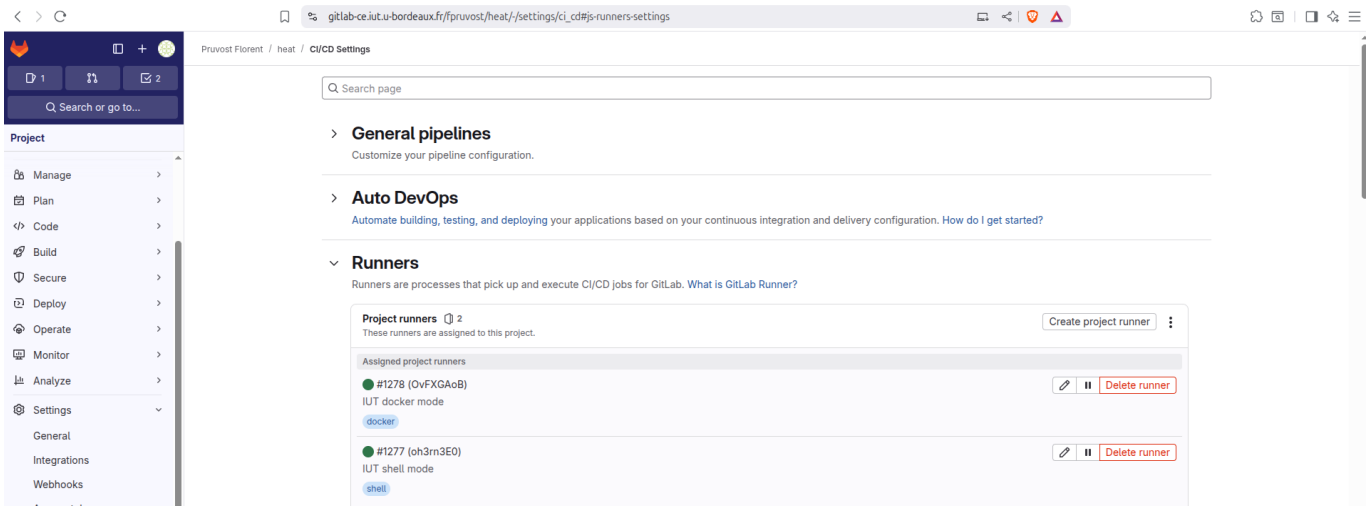

Figure 4: gitlab runner docker

Modifier le script `.gitlab-ci.yml` afin d'ajouter un nouveau job qui devra s'exécuter sur le runner docker, on utilise le système de tags pour cela. Le job shell précédent peut être commenté et ainsi non exécuté juste en ajoutant un point (.) devant le nom du job. Commiter et pousser puis observer les logs du job (Build -> Pipelines).

```
.job-shell:
  tags: ['shell']
  script: ls -la

job-docker:
  tags: ['docker']
  image: registry.u-bordeaux.fr/vhub/alpine:3.18
  script: ls -la
```

> **Note** : Pour s'assurer que les jobs seront bien lancés avec les runners enregistrés lors de ce TP, il est nécessaire de désactiver les **instance runners** déjà installés au département. Pour cela, aller sur : Settings -> CI/CD -> Runners et **désactiver** l'option "Enable instance runners for this project" !

Le job doit être en erreur car docker n'est pas installé.

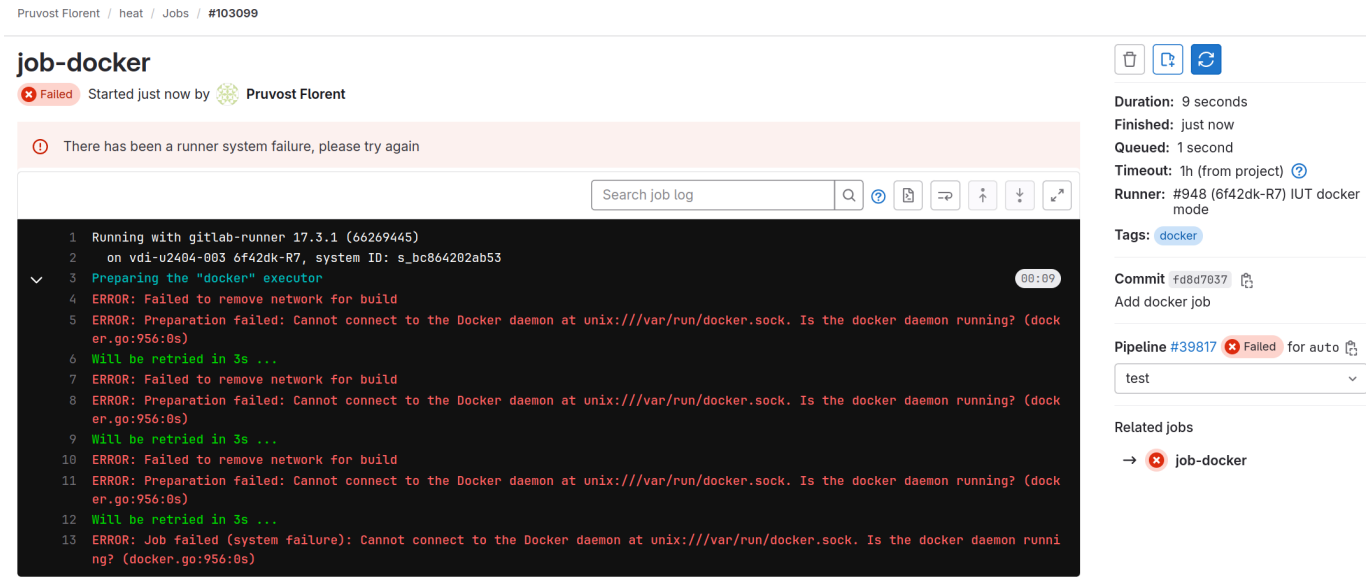

Figure 5: erreur job docker

En effet, sur les machines de l'IUT, docker n'est pas installé car podman est considéré comme une alternative plus sûre. Une configuration additionnelle de gitlab-runner est requise pour fonctionner avec podman (cf. `https://docs.gitlab.com/runner/executors/docker.html#use-podman-to-run-docker-commands`).

Voici les commandes pour créer un service podman utilisateur :

```
/mnt/netta/apps/vnet/bin/podman-init-storage

mkdir -p ~/.config/systemd
mkdir -p ~/.config/systemd/user

# copier les fichiers de /usr/lib/systemd/user/ vers ~/.config/systemd/user
cp /usr/lib/systemd/user/podman.service ~/.config/systemd/user/
cp /usr/lib/systemd/user/podman.socket ~/.config/systemd/user/

systemctl --user enable podman.socket
systemctl --user daemon-reload
systemctl --user start podman.socket

# vérifier que le service podman est bien actif avec
systemctl --user status podman.socket
```

Vous pouvez utiliser le script config-podman.sh de la manière suivante :

```
source config-podman.sh
```

Puis modifier le fichier `~/.gitlab-runner/config.toml` dans la section `[[runners]]` correspondante au runner docker :

1. Ajouter `environment = ["FF_NETWORK_PER_BUILD=1"]` dans la section `[[runners]]`
2. Modifier le champ `privileged = false` en `true` dans la section `[runners.docker]`
3. Ajouter `host = "unix:///run/user/?????/podman/podman.sock"` dans la section `[runners.docker]`, les points d'interrogation `?????` doivent être remplacés par le nombre donné par `systemctl --user status podman.socket` dans le champ Listen.

   Un exemple de configuration :

```
[[runners]]
  name = "runner-docker"
  url = "https://gitlab-ce.iut.u-bordeaux.fr"
  id = 482
  token = "glrt-blablabla-1234567890"
  token_obtained_at = 2023-08-10T15:16:12Z
  token_expires_at = 0001-01-01T00:00:00Z
  executor = "docker"
  environment = ["FF_NETWORK_PER_BUILD=1"]
  [runners.cache]
    MaxUploadedArchiveSize = 0
  [runners.docker]
    host = "unix:///run/user/76955/podman/podman.sock"
    tls_verify = false
    image = "registry.u-bordeaux.fr/vhub/alpine:3.18"
    privileged = true
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/cache"]
    shm_size = 0
    network_mtu = 0
```

Puis redémarrer le service :

```
systemctl --user restart gitlab-runner
```

Relancer le job-docker en erreur. Il doit maintenant passer au vert.

Notez que le job est lancé dans un conteneur podman issu de l'image "registry.u-bordeaux.fr/vhub/alpine:3.18".

Vous pouvez supprimer votre runner de type shell car on ne va plus s'en servir par la suite, cf. Settings -> CI/CD -> Runners -> Delete runner sur le runner "IUT shell mode".

1.6 Mise en place d'un pipeline

Le programme Heat est développé en langage C (avec des éléments C++ et scripts Python) et nécessite une phase de compilation. On va donc mettre en place un pipeline de type phase 1 *build*, puis phase 2 *test*.

Commiter et pousser ces modifications dans le fichier `.gitlab-ci.yml`.

```
stages:
  - build
  - test

default:
  image: registry.u-bordeaux.fr/vhub/alpine:3.18
  tags: ['docker']
```

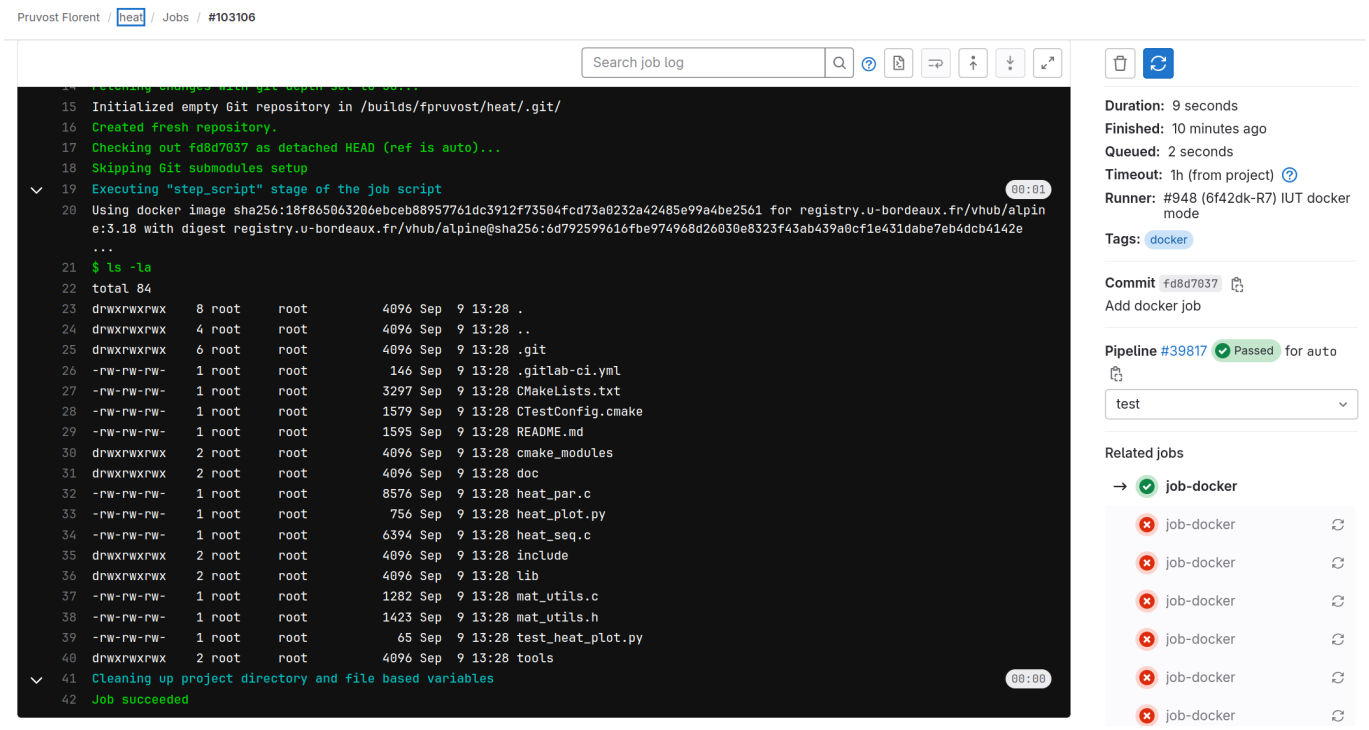

Figure 6: job docker

```
build:
  stage: build
  script:
    - cmake -B build
    - cmake --build build
  cache:
    key: "$CI_COMMIT_REF_SLUG"
    untracked: true
    policy: push

test:
  stage: test
  needs: ["build"]
  script:
    - ctest --test-dir build --verbose
  cache:
    key: "$CI_COMMIT_REF_SLUG"
    untracked: true
    policy: pull
```

Notes sur le fichier .gitlab-ci.yml :

1. **stages** permet de définir les étapes intermédiaires du pipeline et d'y associer un ensemble de jobs. Les jobs d'une étape suivante ne pourront pas démarrer tant que tous les jobs de l'étape précédente n'auront pas terminé.
2. **default** permet de définir certaines propriétés qui seront utilisées dans tous les jobs.
3. **cache** permet de sauvegarder des fichiers entre jobs ou pipelines successifs (**untracked: true** pour sauvegarder tous les fichiers en dehors de ce qui est indexé par Git *i.e.* tout ce qui a été généré après le git clone par la portion script). Dans notre cas précis, cette sauvegarde est stockée dans `/tmp/containers.$USER.d/storage/volumes` dans un fichier cache.zip. Plus d'explications sont disponibles dans la documentation en relation avec le cache GitLab.

En l'état, le job **build** va échouer car les dépendances du projet ne sont pas installées (gcc, cmake, etc.). En effet, l'image docker/podman utilisée, `registry.u-bordeaux.fr/vhub/alpine:3.18` par défaut, est très minimaliste et ne contient qu'un Linux de base, sans outils particuliers pour le développement C/C++. On va justement installer les dépendances manquantes dans la section "2. Définir son environnement de test".

1.7 Guide de démarrage rapide pour CMake

- Le programme Heat est constitué de fichiers .h (en-têtes) et .c à compiler et installer avec CMake.
- Les propriétés du projet CMake sont définies dans les fichiers `CMakeLists.txt`.
- L'installation d'un projet se déroule généralement en 3 étapes : 1. Configuration, 2. Compilation, 3. Installation.

1. La configuration du projet et la détection des dépendances installées sur le système s'effectuent avec

   ```
   cmake -B build
   ```

   Lors de cette étape, des fichiers `Makefile` sont générés dans le dossier `build/` afin de pouvoir compiler par la suite avec des appels à `make` ou `cmake --build`.

   Des options peuvent être choisies à ce moment-là avec -D, ex. :

   - `-DHEAT_USE_MPI=ON` pour activer la dépendance optionnelle MPI,
   - `-DCMAKE_INSTALL_PREFIX=$PWD/install` pour modifier le répertoire d'installation par défaut,
   - `-DBUILD_SHARED_LIBS=ON` pour compiler les éventuelles bibliothèques en mode dynamique (.so et non .a).

   ```
   cmake -B build -DBUILD_SHARED_LIBS=ON -DCMAKE_INSTALL_PREFIX=$PWD/install -DHEAT_USE_MPI=ON
   ```

2. La compilation se fait via

   ```
   cmake --build build # ajouter --verbose pour avoir des détails
   ```

   Des fichiers .c sont convertis en .o, qui pour certains sont archivés en bibliothèque .a ou .so, et pour d'autres associés pour construire un binaire exécutable. Pour supprimer/nettoyer les binaires :

   ```
   cmake --build build/ --target clean
   ```

   Chaque cible existante peut être construite séparément :

   ```
   cmake --build build/ --target heat
   ```

   pour compiler la bibliothèque cf. `ll build/lib/libheat.a`, et

   ```
   cmake --build build/ --target heat_seq
   ```

   pour compiler un des exécutables cf. `ll build/heat_seq`.

3. L'installation peut se faire via

   ```
   cmake --install build
   ```

Si des tests sont définis pour ctest (outil en ligne de commande de CMake), ceux-ci peuvent être déclenchés avec

```
ctest --test-dir build # ajouter --verbose pour avoir des détails
```

2 Définir son environnement de test

2.1 Installation des dépendances dans les jobs gitlab-ci

Les images de base existantes sur internet sont rarement suffisantes pour satisfaire les besoins de notre logiciel. Il faut donc ajouter l'installation des dépendances spécifiques dans les scripts de test. Modifier la configuration pour ajouter l'installation des dépendances, ici `apk add --update git make gcc g++ cmake`, dans une section **before_script** mise dans la partie **default** (sera exécutée avant les instructions dans `script:`) puis commit+push.

```
default:
  image: registry.u-bordeaux.fr/vhub/alpine:3.18
  before_script: apk add --update git make gcc g++ cmake
```

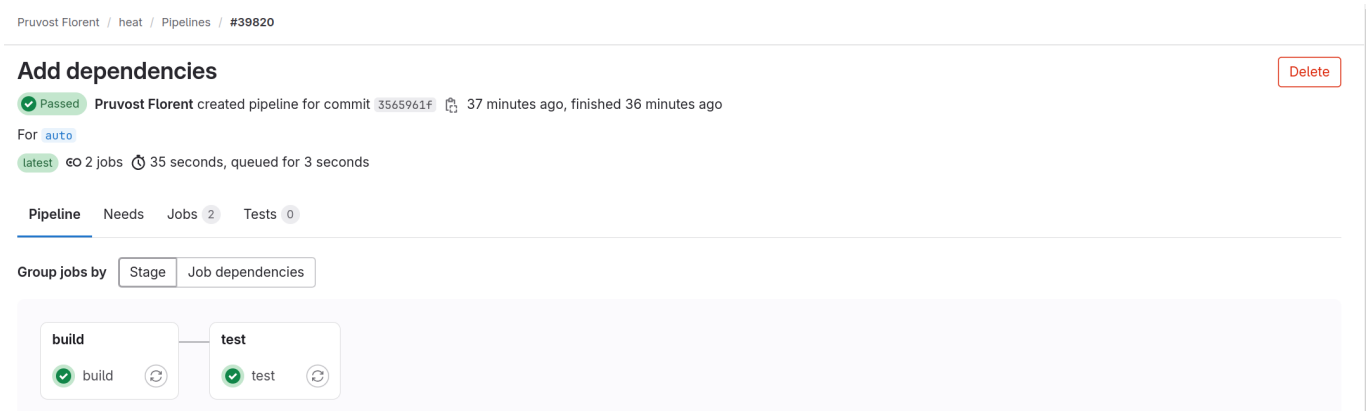

Figure 7: Pipeline build, test

2.2 Construction d'une image podman en local

En l'état, chaque nouveau push dans des branches déclenchera l'installation des dépendances. Cela peut être long et gourmand en ressources lorsque le poids des dépendances est élevé. On va donc construire une image une fois pour toutes contenant le bon environnement de test.

Construire une image docker en local (avec la commande podman) :

```
podman run -it registry.u-bordeaux.fr/vhub/alpine:3.18
# ajouter les paquets (apk add --update git make gcc g++ cmake), tester que heat compile, une fois term
```

À présent que l'environnement de test pour Heat est fonctionnel dans l'image, on peut sauvegarder l'ensemble des commandes dans un fichier `dockerfile-testing` (la recette) :

```
FROM registry.u-bordeaux.fr/vhub/alpine:3.18
RUN apk add --update git make gcc g++ cmake
```

puis taper :

```
podman build -t heat -f dockerfile-testing .
```

et vérifier que l'image se construit bien.

2.3 Stocker votre image sur le registre GitLab de heat

Pousser l'image sur le registre de conteneurs de votre projet GitLab, cf. Deploy -> Container Registry (l'option Container Registry doit être activée dans Settings -> General -> Visibility, project features, permissions).

- D'abord construire l'image avec la bonne convention de nommage pour pouvoir la pousser sur le registre. Le nom du registre est indiqué dans GitLab dans Deploy -> Container Registry, cf. Build image. En plus d'un nom, l'image est dotée d'une étiquette de version, par défaut *latest*.

  ```
  podman build -t gitlab-ce.iut.u-bordeaux.fr:5050/$USER/heat/testing -f dockerfile-testing .
  ```

- Vous pouvez associer à cette image un numéro de version, ex. *v0.1* en plus, à des fins d'archivage et de reproductibilité. En effet, lorsque vous mettrez à jour l'image par défaut (*latest*), la précédente version sera écrasée sans possibilité de vérifier plus tard quelles étaient les versions des dépendances.

  ```
  podman tag gitlab-ce.iut.u-bordeaux.fr:5050/$USER/heat/testing gitlab-ce.iut.u-bordeaux.fr:5050/$U
  ```

- Authentifiez-vous sur le registre GitLab puis poussez l'image :

  ```
  export TOKEN="votre jeton d'accès personnel, cf. https://gitlab-ce.iut.u-bordeaux.fr/-/profile/pers
  export TOKEN=‘cat ~/.gitlabtoken‘ # si vous l'avez stocké dans ce fichier
  podman login gitlab-ce.iut.u-bordeaux.fr:5050 -u $USER -p $TOKEN
  podman push gitlab-ce.iut.u-bordeaux.fr:5050/$USER/heat/testing
  podman push gitlab-ce.iut.u-bordeaux.fr:5050/$USER/heat/testing:v0.1
  ```

Vous retrouvez les images sur GitLab dans Deploy -> Container Registry, et elles seront réutilisables par les jobs des pipelines.

2.4 Utiliser votre image dans les jobs gitlab-ci

Maintenant, on voudrait pouvoir exploiter notre environnement défini par l'image heat/testing (cf. fichier `dockerfile-testi` dans notre pipeline de tests.

2.4.1 Utilisation de l'image heat/testing dans le .gitlab-ci.yml

Modifier son pipeline pour utiliser l'image docker qui vient d'être construite et stockée dans le registre GitLab :

- Il suffit de changer le nom de l'image dans le `.gitlab-ci.yml`, en remplaçant `image: registry.u-bordeaux.fr/vhub` par `image: gitlab-ce.iut.u-bordeaux.fr:5050/$USER/heat/testing` (remplacer $USER par votre login).
- Pour être plus générique, vous pouvez remplacer l'adresse de l'image inscrite en dur par `image: $CI_REGISTRY_IMAGE/` `CI_REGISTRY_IMAGE` étant une variable d'environnement créée par GitLab lors du job et qui contient le nom par défaut des images docker du registre associé au projet.
- Vu que votre image contient déjà les dépendances requises pour la compilation de Heat, vous pouvez ainsi simplifier le script `.gitlab-ci.yml` en **supprimant** l'installation des dépendances `before_script: apk add --update git make gcc g++ cmake` puis relancer le pipeline (commit+push) et vérifier que les tests passent toujours.

2.4.2 Ajout de la documentation

Ajouter le paquet **doxygen** dans la liste des paquets à installer via le `dockerfile-testing`. Reconstruire et pousser l'image sur le registre :

```
podman build -t gitlab-ce.iut.u-bordeaux.fr:5050/$USER/heat/testing -f dockerfile-testing .
podman push gitlab-ce.iut.u-bordeaux.fr:5050/$USER/heat/testing
```

Activer la construction de la documentation avec l'option de configuration CMake `-DHEAT_DOC=ON` cf. `cmake -B build -DHEAT_DOC=ON`, commit+push. Vérifier que dans le job **build**, Doxygen est bien trouvé par CMake (`-- Looking for doxygen - found`) et que la documentation est bien construite (`Generating html`).

2.4.3 Ajout d'une dépendance optionnelle

Faites de même pour la dépendance MPI : ajoutez les paquets *musl-dev openmpi openmpi-dev openssh* dans les dépendances et activez l'option CMake à la configuration `cmake -B build -DHEAT_DOC=ON -DHEAT_USE_MPI=ON`, commit+push. En cas d'échec du job de test, il faut ajouter des variables d'environnement au job, exemple :

```
test:
  stage: test
  needs: ["build"]
  variables:
    OMPI_ALLOW_RUN_AS_ROOT: "1"
    OMPI_ALLOW_RUN_AS_ROOT_CONFIRM: "1"
  script:
    - ctest --test-dir build --verbose
  cache:
    key: "$CI_COMMIT_REF_SLUG"
    untracked: true
    policy: pull
```

Vérifier que le job **test** s'exécute bien et contient le test parallèle MPI *heat_par_4* :

2.4.4 Conclusions

Remarquez qu'il est plus flexible de maintenir l'environnement de test via un fichier tel que le `dockerfile-testing`, car ainsi l'environnement est sauvegardé, partagé et peut être redéployé rapidement sur d'autres machines avec une simple commande docker. La maintenance au coup par coup d'une machine de test spécifique est pratique au départ, mais il est difficile de suivre ce qui a été finalement installé, et si la machine plante définitivement, il faut réinstaller une nouvelle machine spécifique, ce qui peut être fastidieux.

La reconstruction et le redéploiement de l'image docker à la main à chaque mise à jour ne sont pas très pratiques. On va donc automatiser cette action.

```
    Start 3: heat_par_4
3: Test command: /usr/bin/mpiexec "-n" "4" "./heat_par" "10" "10" "200" "2" "2" "0"
3: Working Directory: /builds/fpruvost/heat/build
3: Test timeout computed to be: 1500
3: heat: it = 0, t = 0.000e+00, err = 1.732e+00
3: heat: it = 10, t = 2.500e-02, err = 2.518e-01
3: heat: it = 20, t = 5.000e-02, err = 1.562e-01
3: heat: it = 30, t = 7.500e-02, err = 1.031e-01
3: heat: it = 40, t = 1.000e-01, err = 6.815e-02
3: heat: it = 50, t = 1.250e-01, err = 4.507e-02
3: heat: it = 60, t = 1.500e-01, err = 2.980e-02
3: heat: it = 70, t = 1.750e-01, err = 1.971e-02
3: heat: it = 80, t = 2.000e-01, err = 1.304e-02
3: heat: it = 90, t = 2.250e-01, err = 8.621e-03
3: heat: it = 100, t = 2.500e-01, err = 5.701e-03
3: heat: it = 110, t = 2.750e-01, err = 3.770e-03
3: heat: it = 120, t = 3.000e-01, err = 2.493e-03
3: heat: it = 130, t = 3.250e-01, err = 1.649e-03
3: heat: it = 140, t = 3.500e-01, err = 1.091e-03
3: heat: it = 150, t = 3.750e-01, err = 7.212e-04
3: heat: it = 160, t = 4.000e-01, err = 4.769e-04
3: heat: it = 170, t = 4.250e-01, err = 3.154e-04
3: heat: it = 180, t = 4.500e-01, err = 2.086e-04
3: heat: it = 190, t = 4.750e-01, err = 1.380e-04
3/3 Test #3: heat_par_4 .....................   Passed    0.13 sec
100% tests passed, 0 tests failed out of 3
Total Test time (real) =   0.13 sec
Saving cache for successful job
Not uploading cache auto-non_protected due to policy
Cleaning up project directory and file based variables
Job succeeded
```

Figure 8: Job de test MPI

3 Automatisation de la construction de l'image docker

On va ajouter un job GitLab pour construire l'image docker et la pousser sur le registre comme première étape.

Commencer par ajouter votre fichier `dockerfile-testing` dans l'index Git avec `git add` et `git commit`. Puis modifier le fichier `.gitlab-ci.yml` et y ajouter une première étape *buildenv* dans `stages` (avant *build* et *test*) ainsi qu'un nouveau job (cf. `https://docs.gitlab.com/ee/user/packages/container_registry/build_and_push_images.html#use-g` comme ci-dessous :

```
buildenv:
  stage: buildenv
  image: quay.io/podman/stable
  before_script:
    - podman login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" $CI_REGISTRY
  script:
    - podman build -t $CI_REGISTRY_IMAGE/testing -f dockerfile-testing .
    - podman push $CI_REGISTRY_IMAGE/testing
```

Note : L'authentification nécessaire pour accéder au registre est gérée automatiquement pour un pipeline.

Pour information, en `Docker` natif, cela donnerait :

```
buildenv:
  stage: buildenv
  image: docker
  services:
    - docker:dind
  before_script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
  script:
    - docker build -t $CI_REGISTRY_IMAGE/testing -f dockerfile-testing .
    - docker push $CI_REGISTRY_IMAGE/testing
```

Si le job échoue avec un message d'erreur du type "dial tcp: lookup docker on... no such host", il faudra ajouter une ligne à votre fichier de configuration du runner. Éditer le fichier `~/.gitlab-runner/config.toml` et modifier le champ volume tel que `volumes = ["/var/run/docker.sock:/var/run/docker.sock", "/cache"]`.

4 Maintenance de l'image

Réfléchir aux différentes conditions possibles pour le déclenchement de la reconstruction de l'image docker de test. Pourquoi reconstruire :

1. Selon une politique interne de rétrocompatibilité.
2. Si on a besoin d'une version plus récente des dépendances (changement d'API, nouvelle fonctionnalité).
3. Selon la cible client (contrainte sur l'environnement).
4. Valider pour les versions LTS des OS ciblés (ex. Windows 10, Ubuntu 22.04), ne pas rester sur un vieil OS car les dépendances vont devenir figées, nous empêchant de traiter des mises à jour nécessaires avec les versions plus récentes.

4.1 Toujours tout reconstruire

C'est le cas par défaut dans notre situation actuelle : chaque nouveau commit poussé sur le serveur déclenche un nouveau pipeline incluant le job **buildenv**. Pour changer cet état de fait, il faut ajouter des règles spécifiques au job GitLab en question afin de le déclencher uniquement sous certaines conditions (cf. sections suivantes).

Cette stratégie permet de démontrer que nos tests sont valides dans un environnement dont on a testé le déploiement juste avant. Cela renforce le sentiment de robustesse. En revanche, ce n'est envisageable que pour des cas où la réinstallation des dépendances à chaque pipeline est assez rapide (quelques minutes maximum). Dans le cas contraire, cela peut devenir contre-productif et on envisagera de réduire le temps dû au déploiement de l'environnement de test (cf. sections suivantes).

4.2 Ne reconstruire qu'une partie

Dans certaines situations, on peut choisir d'identifier une partie "stable" et une autre plus "en avance de phase" (avec des dépendances plus ou moins exotiques). Par exemple, sur certaines dépendances on peut avoir besoin d'une version non release (non numérotée, ex. branche master ou autre branche). Ainsi, on peut construire un environnement de base stable (template), qu'on ne construit qu'une fois de temps en temps, et redéployer la partie "en avance de phase" à chaque pipeline.

Dans cette optique, commentez le job de reconstruction de l'image (on va considérer que c'est la partie stable qui n'a pas besoin d'être réinstallée) et ajoutez un job **gtest** installant la toute dernière version de `googletest` à chaque nouveau pipeline, exemple :

```
stages:
  - buildenv
  - gtest
  - build
  - test

default:
  image: $CI_REGISTRY_IMAGE/testing
  tags: ['docker']

.buildenv:
   stage: buildenv
   image: quay.io/podman/stable
   before_script:
     - podman login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" $CI_REGISTRY
   script:
     - podman build -t $CI_REGISTRY_IMAGE/testing -f dockerfile-testing .
     - podman push $CI_REGISTRY_IMAGE/testing

gtest:
  stage: gtest
  script:
    - git clone https://github.com/google/googletest.git
    - cd googletest/
    - cmake -B build -S . -DCMAKE_INSTALL_PREFIX=$PWD/../gtest
    - cmake --build build
    - cmake --install build
  cache:
    key: "$CI_COMMIT_REF_SLUG"
    untracked: true
    policy: push
```

Modifier le fichier `CMakeLists.txt` afin de vérifier que `googletest` a bien été installé :

- ajouter `find_package(GTest REQUIRED)` à la ligne 12 de `CMakeLists.txt`

Modifier le job **build** afin de pouvoir utiliser l'installation de `googletest` :

- `cmake -B build -DHEAT_DOC=ON -DHEAT_USE_MPI=ON -DCMAKE_PREFIX_PATH=$PWD/gtest`
- `policy:  pull-push`

Commiter et pousser afin de valider ce nouveau pipeline. Dans le job **build**, vous devriez observer que CMake trouve bien GTest :

```
-- Found GTest: /builds/fpruvost/heat/gtest/lib/cmake/GTest/GTestConfig.cmake (found version "1.16.0")
```

> **Note** : Une explication détaillée de l'option `policy` est disponible dans la documentation en relation avec le cache GitLab.

Cette expérience étant terminée, et parce qu'on ne se servira pas de GTest par la suite, inversez le commit afin de revenir à l'état précédent :

```
git revert HEAD~1
git push
```

4.3 Mise à jour de l'image stable

Avec une stratégie où l'on ne reconstruit pas tout lors de chaque pipeline, il faut décider d'une politique de mise à jour pour la construction de l'image de base "stable". Plusieurs possibilités s'offrent à nous pour déclencher la mise à jour.

4.3.1 Si le Dockerfile change

Une première méthode consiste à mettre à jour l'image dès que la recette, c'est-à-dire le fichier `dockerfile-testing`, change.

Ajouter ce code dans **buildenv** (décommenter le job en enlevant le point de la ligne `.buildenv:`), commiter, pousser :

```
only:
  changes:
    - dockerfile-testing
```

Normalement, **buildenv** n'est pas déclenché.

Maintenant, ajouter le paquet **curl** à la liste des paquets à installer dans le `dockerfile-testing`, commiter, pousser. Observez que **buildenv** se déclenche bien.

Il est clair que si la recette change, il faut mettre à jour l'image. En revanche, si la recette ne change pas pendant longtemps, cela implique que l'environnement sera figé dans le même temps et peut devenir trop ancien. Il faudra donc mettre à jour l'image de temps en temps manuellement via un build+push docker (podman) à partir de sa machine.

4.3.2 Manuellement (bouton manual)

Une seconde méthode consiste à ajouter un déclencheur manuel dans le pipeline afin de laisser la liberté aux développeurs de mettre à jour quand ils le souhaitent.

Supprimer la partie `only: changes:` et ajouter `when: manual` au job **buildenv**, puis déclencher manuellement le job en allant sur la page des pipelines et en cliquant sur le bouton associé au job **buildenv**.

C'est une méthode flexible, mais le risque est de ne pas faire la mise à jour assez souvent.

4.3.3 Périodique (schedule)

Une troisième méthode consiste à déclencher le job périodiquement. Pour cela, il faut créer un pipeline spécifique de type *schedule*.

Dans GitLab, aller dans Build -> Pipeline schedules -> New schedule :

- Description : buildenv
- Interval Pattern : Every week
- Cron Timezone : UTC+2 Paris
- Target branch or tag : auto

Cliquer sur **Create pipeline schedule**.

> Remarquez la date de prochain déclenchement du pipeline dans la colonne *Next Run*.

Ensuite, modifier **buildenv** pour supprimer la partie `when: manual` et la remplacer par :

```
only:
  - schedules
```

Commiter, pousser. Le pipeline se déclenche sans **buildenv**. En effet, dans cette configuration, ce dernier ne sera lancé que lors du pipeline de type schedule, à la date décidée dans les paramètres du schedule **buildenv**. Il est possible de tester le schedule en cliquant sur le bouton Play dans la page des schedules. Vérifiez que cela fonctionne bien.

5 Optimisation de l'image

5.1 Réflexion sur la taille de l'image

Pour certains projets, il y a beaucoup de dépendances et elles sont lourdes en termes de stockage. Il est donc important d'optimiser au mieux la taille de l'image.

Vérifier la taille occupée par les images et conteneurs podman avec la commande `podman system df`. Si l'image de base utilisée est par exemple Ubuntu, cela peut peser plusieurs gigaoctets. Regarder image par image avec `podman images`.

Il existe fort heureusement des techniques pour réduire la taille des images docker :

- `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`
- `https://devopscube.com/reduce-docker-image-size/`

Testez avec le fichier `Dockerfile` suivant :

```
FROM registry.u-bordeaux.fr/vhub/debian:12-slim
#FROM ubuntu:20.04

# Installing as root: docker images are usually set up as root.
# Since some autotools scripts might complain about this being unsafe, we set
# FORCE_UNSAFE_CONFIGURE=1 to avoid configure errors.
ENV FORCE_UNSAFE_CONFIGURE=1
ENV DEBIAN_FRONTEND noninteractive

RUN apt-get update -y
RUN apt-get install -y git curl make gcc g++ cmake doxygen musl-dev openmpi-bin openmpi-common libopenm
RUN apt-get autoremove -y
```

Remarquez la nouvelle taille de l'image, environ **960 Mo pour Debian** (890 Mo pour Ubuntu) au lieu de **340 Mo dans le cas Alpine**.

5.2 Éviter l'exécution des jobs en mode root

Lorsqu'un programme ou service peut être lancé sans les privilèges *root*, il est préférable de créer un utilisateur Linux dans l'image et de lancer les commandes de test avec cette identité.

Modifier la recette du `dockerfile-testing` en ajoutant :

```
# Create a group and user
RUN addgroup -S gitlab && adduser -S gitlab -G gitlab

# Create a directory where ci jobs are performed
RUN mkdir /builds && \
    chown -R gitlab:gitlab /builds && \
    chmod g+s /builds

# default user
USER gitlab

# default working directory
WORKDIR /builds
```

Il est maintenant possible d'enlever les variables d'environnement lors des ctest :

```
#variables:
#  OMPI_ALLOW_RUN_AS_ROOT: "1"
#  OMPI_ALLOW_RUN_AS_ROOT_CONFIRM: "1"
```

Commit+push.

5.3 Construction multi-étapes (*multi-stage build*)

Une image de compilation a besoin d'outils lourds : compilateurs (GCC, Clang), outils de construction (CMake, Make), fichiers d'en-têtes et paquets de développement (*-dev). En revanche, l'image finale servant à *exécuter* l'application en production n'a besoin que du binaire compilé et de ses bibliothèques dynamiques d'exécution.

La technique des constructions multi-étapes (multi-stage builds) permet d'enchaîner plusieurs directives `FROM` dans un même `Dockerfile` :

```
# Étape 1 : Environnement de compilation lourd (builder)
FROM registry.u-bordeaux.fr/vhub/debian:12-slim AS builder
RUN apt-get update && apt-get install -y cmake gcc make libopenmpi-dev
COPY . /heat
WORKDIR /heat
RUN cmake -B build -DHEAT_USE_MPI=ON && cmake --build build

# Étape 2 : Image finale d'exécution minimale
FROM registry.u-bordeaux.fr/vhub/debian:12-slim
RUN apt-get update && apt-get install -y libopenmpi3 && rm -rf /var/lib/apt/lists/*
# On ne copie que l'exécutable produit depuis l'étape précédente
COPY --from=builder /heat/build/heat_seq /usr/local/bin/heat_seq
COPY --from=builder /heat/build/heat_par /usr/local/bin/heat_par
USER 1000:1000
ENTRYPOINT ["/usr/local/bin/heat_seq"]
```

Constat :

- L'image de build complète pèse près de **900 Mo**.
- L'image finale produite ne pèse qu'environ **35 Mo**, tout en garantissant une surface d'attaque de sécurité considérablement réduite.

6 Dépannage et FAQ Podman Rootless

Sur les machines de l'IUT, l'utilisation conjointe de Podman en mode sans privilèges (**rootless**), de quotas disque stricts et d'un stockage redirigé dans `/tmp/containers` peut générer des comportements spécifiques. Voici comment diagnostiquer et résoudre les problèmes les plus fréquents :

6.1 Quota disque saturé ou erreur « No space left on device »

Si vous obtenez une erreur lors du pull ou de la compilation d'une image, vérifiez l'espace disque consommé par Podman :

```
podman system df
```

Pour libérer l'espace disque occupé par les couches intermédiaires inutilisées, conteneurs arrêtés et volumes temporaires :

```
# Supprimer tous les conteneurs arrêtés et les images non référencées
podman system prune -a --volumes -f
# Si le dossier /tmp/containers/<login> reste encombré
rm -rf /tmp/containers/$USER/storage/tmp/*
```

6.2 Le socket Podman ne répond plus après une reconnexion SSH

L'émulation Docker repose sur le socket UNIX de Podman géré par le systemd utilisateur. Après une déconnexion ou un redémarrage de machine, ce service peut s'arrêter :

```
# Vérifier le statut du service
systemctl --user status podman.socket
# Redémarrer le socket
systemctl --user restart podman.socket
# Vérifier que la variable DOCKER_HOST pointe bien sur le socket actif
export DOCKER_HOST="unix://$XDG_RUNTIME_DIR/podman/podman.sock"
podman --remote info
```

6.3 Le runner ne parvient pas à joindre un service local (ex. SonarQube sur localhost:9000)

Par défaut, chaque conteneur lancé par le runner possède son propre espace de noms réseau (**network namespace**) isolé. Pour qu'un job s'exécutant dans un conteneur puisse joindre un service tournant sur la machine hôte (comme SonarQube sur le port 9000), il faut impérativement spécifier le mode réseau hôte dans la configuration du runner :

```
# dans ~/.gitlab-runner/config.toml
[runners.docker]
  network_mode = "host"
```

Puis redémarrer le service runner :

```
systemctl --user restart gitlab-runner.service
```

6.4 Problème de droits sur les fichiers créés dans un volume (UID mapping rootless)

En mode rootless, l'utilisateur intérieur du conteneur (UID 0) est mappé sur votre propre compte Linux à l'extérieur. Si vous créez un utilisateur non-root dans le conteneur (ex. UID 1000), les fichiers générés dans les volumes montés peuvent sembler appartenir à un UID inaccessible. Utilisez la commande `podman unshare` pour vous placer dans l'espace de noms du conteneur et inspecter ou réparer les droits des fichiers :

```
podman unshare chown -R 1000:1000 /builds
```