Déploiement continu

Florent Pruvost (modifié par Pierre Ramet)

September 20, 2026

Contents

- Déployer automatiquement ses paquets : release source, paquet Debian, image Docker
  - réfléchir aux notions d'intégration continue vs. distribution continue vs. déploiement continu
  - discuter du versioning sémantique : `https://semver.org/`
- Déployer automatiquement sa documentation via GitLab Pages (gitlab.com avec miroir ou remote)

1 Introduction

Vérifiez que vos pipelines sont toujours fonctionnels.

Pour cela, trois prérequis à vérifier :

1. le jeton d'accès personnel (**Personal Access Token**) GitLab,
2. le service `gitlab-runner`,
3. le service `podman`.

Pour avoir accès à votre jeton et à la bonne configuration podman, vous pouvez utiliser le script setenv-podman.sh, commande à taper dans chaque nouveau terminal ou à ajouter dans votre `.bashrc` :

```
source ~/setenv-podman.sh
# rappel : un personal access token valide doit se trouver dans le fichier ~/.gitlabtoken
```

Ensuite, pour lancer le service `gitlab-runner` :

- soit le service est déjà configuré et fonctionnel

```
systemctl --user status gitlab-runner.service
# répond : active (running)
```

- soit vous le lancez manuellement dans un nouveau terminal

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

Enfin, pour valider que les pipelines fonctionnent, lancez un pipeline. Par exemple manuellement via la page Build -> Pipelines -> Run pipeline, en choisissant la branche 'auto' du projet **Heat** sur laquelle vous avez déjà travaillé.

2 Distribuer la dernière version stable

Le processus de développement d'un logiciel implique de faire des releases régulièrement. Par `release` on entend une version bien identifiée, testée, avec une interface fixée, une documentation, un `changelog` (document texte rappelant les modifications entre deux versions) et une ou plusieurs archives disponibles pour l'installation.

Le processus de mise à disposition de ces éléments peut être automatisé afin de gagner du temps humain à chaque création d'une nouvelle release. Ainsi cela permet d'en faire plus facilement et donc plus souvent.

2.1 Créer une release

À partir de votre branche 'auto', on vous demande de créer une nouvelle release source et binaire de Heat et de la publier sur GitLab.

2.1.1 Manuellement via l'interface web GitLab

Pour cela, commencez par générer une archive source. CMake utilise CPack pour faciliter la création de packages source et binaires. Utilisez par exemple ces commandes pour générer les archives source et binaire (compatible Debian) :

```
rm -r build
cmake -S . -B build -DHEAT_DOC=ON
cmake --build build --target doc
cp -r build/doc/html doc/
cmake --build build --target package_source # génère build/heat-1.0.0.tar.gz
cmake --build build --target package # génère build/heat-1.0.0-Linux.deb
```

Petit rappel concernant l'utilisation de CMake ici.

Pour information, la version du paquet, ici 1.0.0, est définie par la variable CMake `HEAT_VERSION` dans le fichier `CMakeLists.txt`. Remarquez que l'on génère la documentation afin de l'incorporer dans l'archive source.

Sauvegardez les 6 lignes précédentes de création de paquets (les commandes avec rm, cmake, cp) dans un script `./tools/release.sh` avec comme en-tête (1re ligne) `#!/bin/sh`. Testez son fonctionnement :

```
chmod +x ./tools/release.sh
./tools/release.sh
ll build/heat-1.0.0*
```

Maintenant, sauvegardez votre état actuel via un tag Git, ex. v1.0.0 :

```
git add ./tools/release.sh
git commit -m "Ajout script release.sh"
git tag v1.0.0
git push origin --tags
```

Visitez ensuite la page Code -> Tags de votre projet et vérifiez que vous trouvez bien votre tag **v1.0.0**. Le tag Git permet d'identifier cette version au sens Git et donc de retrouver cette version précise du code source avec `git checkout v1.0.0`. Utilisez `git checkout auto` (ou `git switch auto`) pour revenir sur votre branche **auto**.

Sur la page des tags, vous pouvez créer manuellement une release au sens GitLab à partir d'un tag existant. Cliquez sur 'Create release' à droite, puis ajoutez quelques commentaires dans la section 'Release notes' :

```
- Add .gitlab-ci.yml to use pipelines
- Use a dockerfile-testing file to define the testing environment
```

Vous pouvez ajouter des liens de type URL et des fichiers attachés. Utilisez le bouton "Attach a file or image" (trombone dans la partie **Release notes**) afin d'y ajouter les fichiers .tar.gz et .deb que vous venez de générer avec CPack. Sauvegarder. Vous avez une nouvelle release sur la page Deploy -> Releases avec le code source téléchargeable ainsi que vos deux paquets associés.

2.1.2 Via l'API GitLab

La création de release GitLab peut être effectuée via une commande curl afin d'interagir avec l'API REST de GitLab, l'opération peut ainsi être scriptable et, nous le verrons par la suite, automatisable.

Lors du TP précédent, vous aviez créé un Personal Access Token sauvegardé dans le fichier `~/.gitlabtoken`. Ce jeton vous permet d'exécuter des commandes pour interagir avec GitLab et de réaliser des opérations sur vos projets.

Exemple pour interroger GitLab sur la liste des membres d'un projet spécifique. Le projet est identifiable via son numéro identifiant unique que l'on trouve sur la page principale (cf. "Project ID", trois petits points verticaux en haut à droite, à côté de Star et Fork) :

```
export TOKEN=‘cat ~/.gitlabtoken‘
export PROJECT_ID="votre numéro de projet"
curl --header "PRIVATE-TOKEN: $TOKEN" "https://gitlab-ce.iut.u-bordeaux.fr/api/v4/projects/$PROJECT_ID/

# sortie json structurée
curl --header "PRIVATE-TOKEN: $TOKEN" "https://gitlab-ce.iut.u-bordeaux.fr/api/v4/projects/$PROJECT_ID/
```

Vous pouvez accéder à l'API sur les releases (lister, créer, supprimer) :

```
# afficher les releases existantes
curl --header "PRIVATE-TOKEN: $TOKEN" "https://gitlab-ce.iut.u-bordeaux.fr/api/v4/projects/$PROJECT_ID/

# afficher leurs noms
curl --header "PRIVATE-TOKEN: $TOKEN" "https://gitlab-ce.iut.u-bordeaux.fr/api/v4/projects/$PROJECT_ID/

# supprimer la release v1.0.0
export TAG_NAME=v1.0.0
curl --header "PRIVATE-TOKEN: $TOKEN" --request DELETE "https://gitlab-ce.iut.u-bordeaux.fr/api/v4/proj
```

Activer le Package Registry : Settings -> General -> Visibility, project features -> Package registry -> enable and save changes.

Vous pouvez maintenant téléverser des fichiers de type paquets binaires via l'API du Generic Package Repository. Vous pouvez visiter la page Deploy -> Package Registry et y voir vos paquets. Vous pouvez aussi les supprimer à la main.

```
RELEASE_NUM=‘echo $TAG_NAME | sed -e "s#v##"‘
# téléverser heat-1.0.0.tar.gz sur le package registry
curl --header "PRIVATE-TOKEN: $TOKEN" --upload-file ./build/heat-$RELEASE_NUM.tar.gz "https://gitlab-ce

# téléverser heat-1.0.0-Linux.deb sur le package registry
curl --header "PRIVATE-TOKEN: $TOKEN" --upload-file ./build/heat-$RELEASE_NUM-Linux.deb "https://gitlab

# pour retrouver l'id du package
# PACKAGE_ID=‘curl --header "PRIVATE-TOKEN: $TOKEN" "https://gitlab-ce.iut.u-bordeaux.fr/api/v4/project
# echo $PACKAGE_ID

# pour supprimer le package à partir de id (ne pas le faire, car on va avoir besoin de ce package)
# curl --header "PRIVATE-TOKEN: $TOKEN" --request DELETE "https://gitlab-ce.iut.u-bordeaux.fr/api/v4/pr
```

Vous pouvez les télécharger ensuite simplement via `wget` :

```
wget "https://gitlab-ce.iut.u-bordeaux.fr/api/v4/projects/$PROJECT_ID/packages/generic/release/$RELEASE
wget "https://gitlab-ce.iut.u-bordeaux.fr/api/v4/projects/$PROJECT_ID/packages/generic/release/$RELEASE
ll heat-$RELEASE_NUM*
rm heat-$RELEASE_NUM.tar.gz
rm heat-$RELEASE_NUM-Linux.deb
```

Si vous n'avez pas coché la case "*Allow anyone to pull from package registry*" dans les Settings -> General -> Permissions, il faudra ajouter l'authentification pour pouvoir télécharger les paquets :

```
wget "https://$USER:$TOKEN@gitlab-ce.iut.u-bordeaux.fr/api/v4/projects/$PROJECT_ID/packages/generic/rel
wget "https://$USER:$TOKEN@gitlab-ce.iut.u-bordeaux.fr/api/v4/projects/$PROJECT_ID/packages/generic/rel
```

Admettons que le tag est bien déjà créé et que les fichiers sont téléversés sur le package registry, nous allons maintenant procéder à la création de la release finale via l'API :

```
CMD=‘echo curl --header \"Content-Type: application/json\" \
       --header \"PRIVATE-TOKEN: $TOKEN\" \
       --data \’{ \"name\": \"$TAG_NAME\", \"tag_name\": \"$TAG_NAME\", \
  \"assets\": { \"links\": [{ \"name\": \"heat-$RELEASE_NUM.tar.gz\", \"url\": \"https://gitlab-ce.iut.
    { \"name\": \"heat-$RELEASE_NUM-Linux.deb\", \"url\": \"https://gitlab-ce.iut.u-bordeaux.fr/api/v4/
}\’ \
       --request POST \"https://gitlab-ce.iut.u-bordeaux.fr/api/v4/projects/$PROJECT_ID/releases\"‘
eval $CMD | jq
```

Vous pouvez visiter la page Deploy -> Releases.

Modifiez maintenant le script `release.sh` afin de créer la release v1.0.0 via l'API GitLab en y ajoutant les commandes pour :

1. téléverser les fichiers de packages et
2. créer la release dans la foulée avec l'évaluation de la commande curl ci-dessus.

Sauvegarder le script `./tools/release.sh` :

```
git commit ./tools/release.sh -m "release.sh: publication packages et release"
git push
```

2.1.3 Via la CLI GitLab

GitLab-CLI (`glab`) est un outil en ligne de commande pour GitLab. Il s'intègre dans votre terminal sans avoir à jongler entre les fenêtres et les onglets de votre navigateur. Il permet par exemple de créer, gérer et supprimer des releases.

Pour disposer de cet outil, vous pouvez utiliser l'image docker officielle, disponible aussi sur le registre de l'Université de Bordeaux :

```
podman pull registry.u-bordeaux.fr/tthor/glab:latest
```

Pour utiliser `glab`, il faut d'abord le configurer pour qu'il puisse se connecter à l'instance GitLab de l'IUT. Voici un template à adapter à votre configuration et à placer dans le fichier `~/.config/glab-cli/config.yml` :

```
# Default GitLab hostname to use.
host: gitlab-ce.iut.u-bordeaux.fr
# Allow glab to automatically check for updates and notify you when there are new updates.
check_update: true
# Configuration specific for GitLab instances.
hosts:
    gitlab-ce.iut.u-bordeaux.fr:
user: __MON_LOGIN_GITLAB_IUT__
token: __MON_TOKEN_GITLAB_IUT__
container_registry_domains: gitlab-ce.iut.u-bordeaux.fr,gitlab-ce.iut.u-bordeaux.fr:443,registry.gitlab
api_host: gitlab-ce.iut.u-bordeaux.fr
git_protocol: ssh
api_protocol: https
```

Il faudra aussi modifier les droits par défaut de ce fichier :

```
chmod 600 ~/.config/glab-cli/config.yml
```

Vous pouvez télécharger le template config-glab.yml.

On vous recommande ensuite de créer une fonction et un alias pour lancer `glab` dans un conteneur podman :

```
function do-glab {
  podman run --rm -it -v ~/.config/glab-cli:/root/.config/glab-cli registry.u-bordeaux.fr/tthor/glab:la
}

alias glab="do-glab"
```

Pour disposer de cet alias, vous pouvez utiliser le script config-glab.sh de la manière suivante :

```
source config-glab.sh
```

Vous pouvez maintenant tester `glab` en listant les releases de votre projet avec :

```
glab release list -R $USER/heat
```

Vous pouvez ensuite essayer la création d'une release (après avoir supprimé l'une des versions existantes) en suivant les options détaillées dans cette documentation.

2.2 Distribution en continu

Le but ici est d'automatiser la création et la distribution d'une release par un job gitlab-ci.

Lors d'un pipeline, GitLab CI fournit des variables d'environnement contenant des informations : URL du projet Git, référence Git exécutée, jeton GitLab temporaire (CI_JOB_TOKEN), etc. On va donc modifier le script `release.sh` pour utiliser les variables prédéfinies des pipelines existantes. Remplacer les noms de variables d'environnement dans le script, en respectant la casse :

- `PRIVATE-TOKEN: $TOKEN` par `JOB-TOKEN: $CI_JOB_TOKEN`
- `TAG_NAME` par `CI_COMMIT_TAG`
- `PROJECT_ID` par `CI_PROJECT_ID`

Attention à bien conserver le champ `tag_name` en minuscule dans la commande `curl` avec le '–data'.

Maintenant, modifier `.gitlab-ci.yml` pour ajouter un job `release` (ajouter aussi le stage *release* après *test* tout en haut) qui permet de créer la release automatiquement lorsque l'utilisateur pousse un nouveau tag (respectant une certaine syntaxe sémantique vX.Y.Z) sur GitLab.

Deux approches sont possibles pour réaliser cette étape :

2.2.1 Approche 1 : Script personnalisé avec curl et l'API REST

C'est l'approche que nous avons mise en place avec le script `release.sh` :

```
release_curl:
  stage: release
  rules:
    - if: $CI_PIPELINE_SOURCE == "push" && $CI_COMMIT_TAG =~ /^v[0-9]\.[0-9]\.[0-9]$/
  artifacts:
    name: "$CI_JOB_NAME-$CI_COMMIT_REF_SLUG"
    paths:
      - build/heat-*.tar.gz
      - build/heat-*.deb
  script:
    - ./tools/release.sh
```

2.2.2 Approche 2 : Outil officiel natif GitLab release-cli

GitLab fournit désormais une image officielle et un mot-clé dédié `release:` dans le fichier `.gitlab-ci.yml` via l'outil release-cli. Cette approche est plus déclarative, plus sécurisée et évite d'avoir à manipuler manuellement des chaînes JSON complexes avec curl et jq :

```
release:
  stage: release
  image: registry.gitlab.com/gitlab-org/release-cli:latest
  rules:
    - if: $CI_COMMIT_TAG =~ /^v[0-9]\.[0-9]\.[0-9]$/
  # Le mot-clé needs permet d'exécuter ce job dès que le job de build est terminé (DAG)
  needs:
    - job: build
      artifacts: true
  script:
    - echo "Création de la release $CI_COMMIT_TAG avec release-cli"
  release:
    tag_name: $CI_COMMIT_TAG
    description: 'Release automatique générée par GitLab CI pour $CI_COMMIT_TAG'
    assets:
      links:
- name: "heat-$CI_COMMIT_TAG.tar.gz"
  url: "${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/generic/release/${CI_COMMIT_TAG}/heat-${CI_
- name: "heat-$CI_COMMIT_TAG-Linux.deb"
  url: "${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/generic/release/${CI_COMMIT_TAG}/heat-${CI_
```

2.2.3 Comparaison des deux approches

Critère	Approche curl + API REST	Approche native re
Dépendances dans l'image	Nécessite 'curl', 'jq', 'bash' dans l'image de test	Image légère dédiée
Robustesse syntaxique	Concaténation JSON fragile avec quotes échappées	Déclaratif en YAM
Authentification	Gestion manuelle de l'en-tête 'JOB-TOKEN: $CI_JOB_TOKEN'	Intégrée nativement
Pédagogie	Formateur pour comprendre les requêtes HTTP/REST	Standard industriel

2.2.4 Optimisation du pipeline : Directed Acyclic Graph (DAG) avec 'needs:'

Par défaut, GitLab CI exécute les stages de façon strictement séquentielle : tous les jobs du stage 'test' doivent être terminés avant que le moindre job du stage 'release' ou 'deploy' ne démarre.

Le mot-clé needs: permet de briser cette dépendance rigide en définissant un **graphe orienté acyclique** :

- Un job peut démarrer dès que les jobs listés dans son 'needs:' sont validés, sans attendre les autres jobs du même stage.
- Si un job n'a besoin d'aucun artefact préalable (ex. déploiement d'une documentation statique ou linter rapide), on peut spécifier `needs: []` pour qu'il s'exécute immédiatement dès le début du pipeline !

2.2.5 Application sur le projet Heat

Modifier le fichier `dockerfile-testing` afin d'ajouter les paquets `bash curl jq dpkg graphviz` (dépendances nécessaires pour la création de l'archive).

Utiliser la règle suivante dans le job **buildenv** à la place de vos anciennes expérimentations (only: changes, when: manual, only: schedule). Commiter, pousser sur la branche auto. Cela devrait reconstruire l'image de test, tout en évitant de reconstruire l'image pour d'autres scénarios (nouvelle étiquette, pipeline manuel, schedule) :

```
rules:
  - if: $CI_PIPELINE_SOURCE == "push" && $CI_COMMIT_TAG == null
    changes:
      paths:
        - dockerfile-testing
```

Modifier le numéro de version dans le CMakeLists.txt afin d'être à 1.1.0 :

```
set(HEAT_VERSION_MAJOR "1")
set(HEAT_VERSION_MINOR "1")
set(HEAT_VERSION_PATCH "0")
```

Commiter et pousser :

```
git add -u
git commit -m "Release v1.1.0"
git push
```

Tester à présent la création d'une étiquette et la pousser sur GitLab :

```
git tag v1.1.0
git push origin --tags
```

Si cela ne fonctionne pas, après avoir analysé les logs du job en échec, vous pouvez supprimer l'étiquette (en local et sur le serveur), puis modifier vos sources, commiter, pousser pour retester la procédure :

```
git tag -d v1.1.0
git push --delete origin v1.1.0
git tag v1.1.0
git push origin --tags
```

Ne pas hésiter à lire ceci à propos du versioning des releases : `https://semver.org/`.

Pour garantir que le numéro de tag est cohérent avec le numéro de version du projet, cf. `HEAT_VERSION` dans le CMakeLists.txt principal qui donne la version aux fichiers générés .tar.gz/.deb, on peut ajouter ces lignes dans `release.sh` avant de téléverser les archives .tar.gz/.deb avec curl :

```
    PACKAGE=`ls build/heat-*.tar.gz`
    PACKAGE_NUM=`echo $PACKAGE | grep -o "[0-9].[0-9].[0-9]"`
    if [ $PACKAGE_NUM != $RELEASE_NUM ]; then
      echo "PACKAGE=$PACKAGE"
      echo "PACKAGE_NUM=$PACKAGE_NUM"
      echo "RELEASE_NUM=$RELEASE_NUM"
      echo "(CI_COMMIT_TAG=$CI_COMMIT_TAG)"
      echo "Package number does not match RELEASE_NUM (computed from CI_COMMIT_TAG, without v), exit"
      exit 1
    fi
```

3 Déploiement en continu

Jusqu'à présent, nous avons pu mettre en œuvre les tests unitaires automatiques (intégration continue) ainsi que la distribution d'une version stable via le processus de release (distribution continue). Pour certaines applications, par exemple des services web, il est en plus nécessaire d'installer la dernière version mise à jour dans son environnement de production : on parle alors de déploiement continu.

3.1 Déploiement de son application via Docker

Le fichier `Dockerfile` sert de recette au déploiement de notre application.

Ajouter un fichier `Dockerfile` au Git afin d'installer l'application dans l'image docker :

```
ARG IMAGE_FROM
FROM $IMAGE_FROM
RUN mkdir -p heat/
COPY . heat/
RUN cd heat/ && cmake -B build && cmake --build build
USER root
RUN cd heat/ && cmake --install build && cd /builds && rm -r heat/
USER gitlab
```

Pour tester localement :

```
rm build -rf
podman build -t heat -f Dockerfile --build-arg IMAGE_FROM=gitlab-ce.iut.u-bordeaux.fr:5050/$USER/heat/t
```

Remarquez qu'on se base ici sur notre image de test pour éviter la réinstallation de tous les paquets.

Modifier le `.gitlab-ci.yml` afin d'y ajouter le job de déploiement (ajouter aussi **deploy** comme dernière étape dans les `stages` en début de fichier) :

```
deploy:
  stage: deploy
  rules:
    - if: $CI_PIPELINE_SOURCE == "push"
  dependencies: []
  image: quay.io/podman/stable
  variables:
    IMAGE_TAG: $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG
  before_script:
    - podman login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" $CI_REGISTRY
  script:
    - podman build -t $IMAGE_TAG -f Dockerfile --build-arg IMAGE_FROM=$CI_REGISTRY_IMAGE/testing .
    - podman push $IMAGE_TAG
```

Notez que le nom de notre image est constitué de la racine de notre registre `CI_REGISTRY_IMAGE` + le nom de la référence Git (branche, tag) `CI_COMMIT_REF_SLUG`. On peut ainsi construire des environnements spécifiques par branche.

> **Note** : Le mot-clé *dependencies: []* est utilisé pour indiquer qu'on ne souhaite pas télécharger les artefacts des jobs des étapes précédentes puisqu'ils ne sont pas nécessaires ici.

> **Note** : On aurait pu utiliser la même règle `rules` que pour le job release (tag push), mais on souhaite ici pouvoir déployer l'environnement de production pour chaque évolution dans les branches. Éventuellement, on pourrait déployer seulement sur la branche `master` et pour des `tags`, en ajoutant :

```
    - if: $CI_PIPELINE_SOURCE == "push" && $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
    - if: $CI_PIPELINE_SOURCE == "push" && $CI_COMMIT_TAG
```

Visiter la page Deploy -> Container Registry et vérifier la présence de votre image de production.
Tester l'image en local :

```
export TOKEN=‘cat ~/.gitlabtoken‘
podman login gitlab-ce.iut.u-bordeaux.fr:5050 -u $USER -p $TOKEN
podman run gitlab-ce.iut.u-bordeaux.fr:5050/$USER/heat:auto /usr/local/bin/heat_seq 33 33 33 1 0
```

3.2 Déploiement de la documentation

Un autre exemple courant de déploiement continu est la documentation. On cherche par exemple à maintenir une page de documentation à jour par rapport au code de la branche principale.

Sur le GitLab de l'IUT, la fonctionnalité de *Pages* (site web de pages HTML/CSS statiques associé au projet Git) n'est pas activée, il faut donc tester cette fonctionnalité sur un autre GitLab.

- Créer un compte sur GitLab.com : `https://gitlab.com/users/sign_in`.
- Créer un projet vide (sans README initial) *heat*.

Dans votre projet Heat local, commencez par pousser la branche master sur le serveur GitLab.com :

```
export LOGIN="votre login gitlab.com"
git remote add gitlabcom https://gitlab.com/$LOGIN/heat.git
git switch master
git push gitlabcom master
```

Ensuite, positionnez-vous sur votre branche *auto*, créez une nouvelle branche *pages* et ajoutez un script `./tools/pages.sh` générant la documentation :

```
#!/bin/sh

set -x

mkdir public

cmake -B build -DHEAT_DOC=ON
cmake --build build
cp -r build/doc/html public/doxygen

./build/heat_seq 33 33 500 1 0
python3 heat_plot.py
ffmpeg -i heat.avi -c:v libtheora -q:v 7 -c:a libvorbis -q:a 4 heat.ogv

cp tools/index.html public/
cp heat.ogv public/
```

On pourra utiliser le fichier `./tools/index.html` suivant :

```
<video style="display:block; margin: 0 auto;" width="900" height="900" controls>
  <source src="heat.ogv" type="video/mp4">
  Your browser does not support the video tag.
</video>

See the doxygen documentation <a href=./doxygen/>here</a>.
```

Modifiez le fichier `.gitlab-ci.yml` pour y ajouter le job de construction de la documentation :

```
pages:
  stage: deploy
  image: fpruvost/heat
  needs: [] # Le job s'exécute immédiatement sans attendre les tests (DAG)
  rules:
    - if: $CI_COMMIT_BRANCH == "master" || $CI_COMMIT_BRANCH == "pages"
```

```
  artifacts:
    paths:
      - public
  script:
    - ./tools/pages.sh
```

Par convention, le job doit être nommé exactement **pages** et les fichiers ajoutés dans les artefacts seront automatiquement téléversés sur le site web associé au projet sur GitLab.com (voir la page Deploy -> Pages).

Pour information, l'image docker `fpruvost/heat` a été construite avec ce Dockerfile (voir sur Docker Hub) :

```
FROM ubuntu:24.04

# 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 build-essential clang cmake curl doxygen ffmpeg gcovr git python-is-python3
RUN apt-get autoremove -y
```

Commentez les autres jobs (avec un point devant le nom de chaque job) puis testez le job *pages* en poussant votre branche *pages* sur GitLab.com :

```
git add ...
git commit -m ...
git push gitlabcom pages
```

Dans vos pipelines, un nouveau job *pages:deploy* sera créé automatiquement et se chargera de la mise à jour du site web avec les nouveaux fichiers.

Vous pouvez maintenant visiter la page de votre site dont l'URL est indiquée sur Deploy -> Pages, quelque chose comme : `https://\protect\T1\textdollarLOGIN.gitlab.io/heat/`.