CI / CD avec moteur de production

Plan

Plan

Un pipeline CICD ne scripte pas tout !

Les scripts n'effectuent pas d'opérations bas-niveau

- on ne scripte pas la compilation ou l'exécution des tests
- ni la création de la documentation
- ni les dépendances
- $\Rightarrow$ on appelle les *tasks* du moteur de production utilisé pour le projet, e.g. Gradle ou Maven

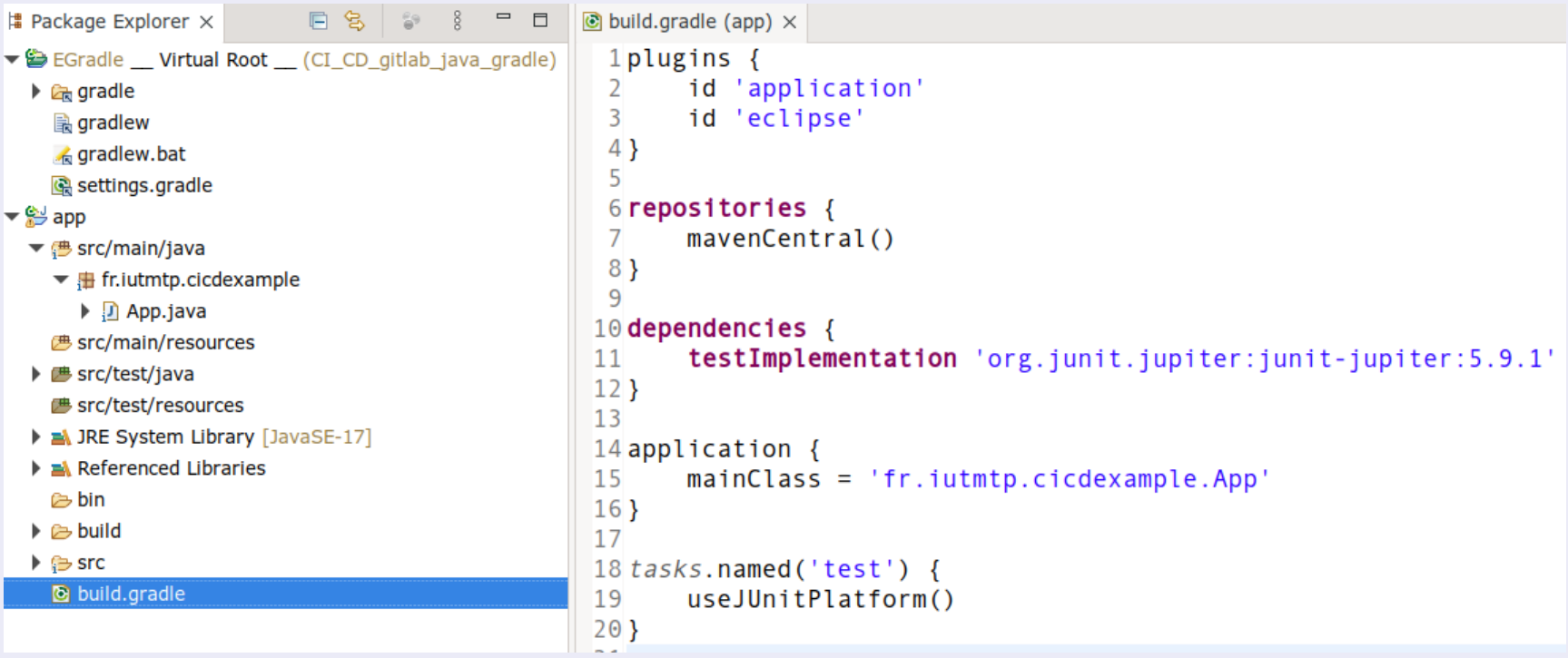
Plan

Gradle comme moteur de production

Gradle

Soit un fichier de build minimal pour un projet Gradle :

build.gradle



Le pipeline appelle les tasks du build Gradle

`.gitlab-ci.yml` (simplification de ▸ Gradle CI template)

Fichier Édition Affichage Rechercher Terminal Aide

```
CI_CD_gitlab_java_gradle on  1_first_pipeline_with_Gradle via v7.6.3 via ☕ v17.0.9
❯ bat .gitlab-ci.yml
```

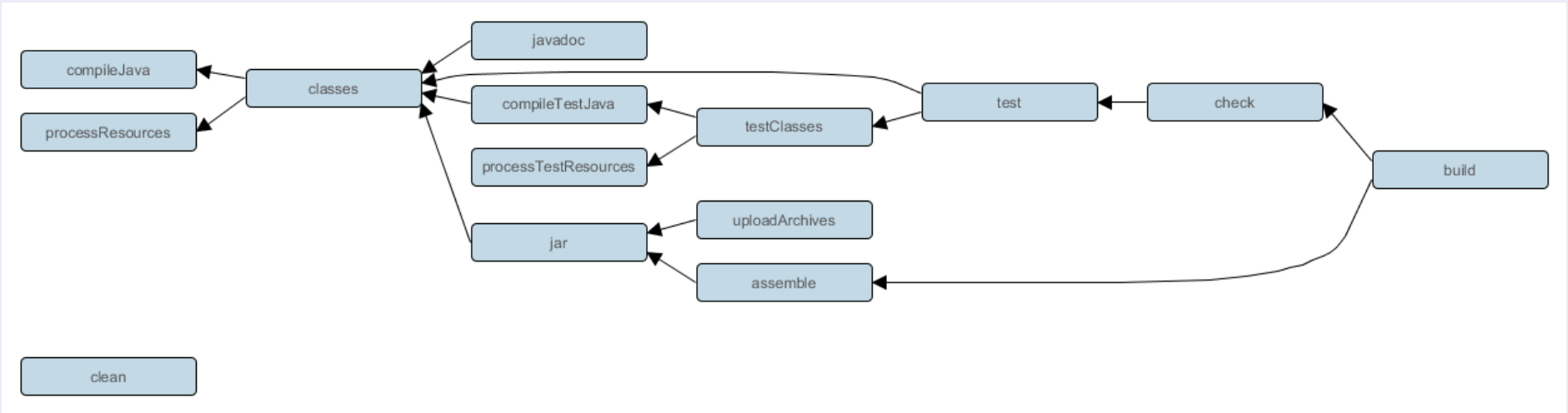
File: .gitlab-ci.yml

```yaml
image: gradle:alpine

before_script:
  - GRADLE_USER_HOME="$(pwd)/.gradle"
  - export GRADLE_USER_HOME

build:
  stage: build
  script: gradle --build-cache assemble
  cache:
    paths:
      - build
      - .gradle

test:
  stage: test
  script: gradle check
  cache:
    paths:
      - build
      - .gradle
```

```
CI_CD_gitlab_java_gradle on  1_first_pipeline_with_Gradle via v7.6.3 via ☕ v17.0.9
❯
```

Tasks fournies par le plugin application de Gradle

Gradle tasks et leurs dépendances



Résultat du job de build

```
Resolving secrets 00:00
Preparing the "docker" executor
Preparing environment 00:03
Getting source from Git repository 00:01
Restoring cache 00:00
Executing "step_script" stage of the job script 00:09
Using docker image sha256:a09086cffe0d4dfe4dba034da3448e77effbfbd0172d343f79dbea21897d4173 for gradle:alpine with digest gradle@sha256:8c855da901503d4bd6ea098a2574766f6469dd0105c7c74e4724d8377cbe23ae ...
$ GRADLE_USER_HOME="$(pwd)/.gradle"
$ export GRADLE_USER_HOME
$ gradle --build-cache assemble
Welcome to Gradle 8.4!
Here are the highlights of this release:
 - Compiling and testing with Java 21
 - Faster Java compilation on Windows
 - Role focused dependency configurations creation
For more details see https://docs.gradle.org/8.4/release-notes.html
Starting a Gradle Daemon (subsequent builds will be faster)
> Task :app:compileJava
> Task :app:processResources NO-SOURCE
> Task :app:classes
> Task :app:jar
> Task :app:startScripts
> Task :app:distTar
> Task :app:distZip
> Task :app:assemble
BUILD SUCCESSFUL in 7s
5 actionable tasks: 5 executed
Saving cache for successful job 00:01
Creating cache default-non_protected...
WARNING: build: no matching files
.gradle: found 136 matching files and directories
No URL provided, cache will be not uploaded to shared cache server. Cache will be stored only locally.
Created cache
Cleaning up project directory and file based variables 00:01
Job succeeded
```

Résultat du job de test

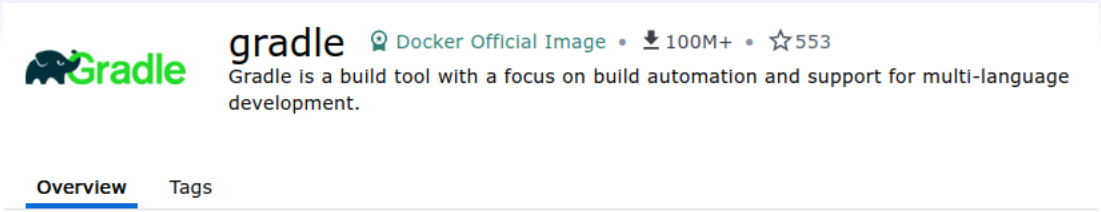
```
25 Executing "step_script" stage of the job script 00:12
26 Using docker image sha256:a09086cffe0d4dfe4dba034da3448e77effbfbd0172d343f79dbea21897d4173 for gradle:alpine with digest gradle@sha256:8c855da901503
   d4bd6ea098a2574766f6469dd0105c7c74e4724d8377cbe23ae ...
27 $ GRADLE_USER_HOME="$(pwd)/.gradle"
28 $ export GRADLE_USER_HOME
29 $ gradle check
30 Welcome to Gradle 8.4!
31 Here are the highlights of this release:
32  - Compiling and testing with Java 21
33  - Faster Java compilation on Windows
34  - Role focused dependency configurations creation
35 For more details see https://docs.gradle.org/8.4/release-notes.html
36 Starting a Gradle Daemon (subsequent builds will be faster)
37 > Task :app:compileJava
38 > Task :app:processResources NO-SOURCE
39 > Task :app:classes
40 > Task :app:compileTestJava
41 > Task :app:processTestResources NO-SOURCE
42 > Task :app:testClasses
43 > Task :app:test
44 > Task :app:check
45 Deprecated Gradle features were used in this build, making it incompatible with Gradle 9.0.
46 You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own scripts or plugins.
47 For more on this, please refer to https://docs.gradle.org/8.4/userguide/command_line_interface.html#sec:command_line_warnings in the Gradle document
   ation.
48 BUILD SUCCESSFUL in 11s
49 3 actionable tasks: 3 executed
51 Saving cache for successful job 00:01
52 Creating cache default-non_protected...
53 WARNING: build: no matching files
54 .gradle: found 262 matching files and directories
55 No URL provided, cache will be not uploaded to shared cache server. Cache will be stored only locally.
56 Created cache
58 Cleaning up project directory and file based variables 00:01
60 Job succeeded
```

En fait, plusieurs problèmes se posent

Le fichier de pipeline n'est pas assez précis

- Son exécution n'utilise pas la version de Gradle définie dans le projet : ici 7.6.3
- Et utilise une version de Java différente !
- $\Rightarrow$ utiliser la bonne image docker : ▸ Gradle docker images ?

Utiliser la bonne image docker ?



gradle Docker Official Image • 100M+ • ☆553

Gradle is a build tool with a focus on build automation and support for multi-language development.

Overview Tags

Quick reference

- Maintained by:
  Keegan Witt (of the Groovy Project), with the Gradle Project's approval
- Where to get help:
  the Docker Community Slack, Server Fault, Unix & Linux, or Stack Overflow

Supported tags and respective `Dockerfile` links

- `8.4.0-jdk8`, `8.4-jdk8`, `8-jdk8`, `jdk8`, `8.4.0-jdk8-jammy`, `8.4-jdk8-jammy`, `8-jdk8-jammy`, `jdk8-jammy`
- `8.4.0-jdk8-focal`, `8.4-jdk8-focal`, `8-jdk8-focal`, `jdk8-focal`
- `8.4.0-jdk11`, `8.4-jdk11`, `8-jdk11`, `jdk11`, `8.4.0-jdk11-jammy`, `8.4-jdk11-jammy`, `8-jdk11-jammy`, `jdk11-jammy`
- `8.4.0-jdk11-focal`, `8.4-jdk11-focal`, `8-jdk11-focal`, `jdk11-focal`
- `8.4.0-jdk11-alpine`, `8.4-jdk11-alpine`, `8-jdk11-alpine`, `jdk11-alpine`
- `8.4.0-jdk17`, `8.4-jdk17`, `8-jdk17`, `jdk17`, `8.4.0-jdk`, `8.4-jdk`, `8-jdk`, `jdk`, `8.4.0`, `8.4`, `8`, `latest`, `8.4.0-jdk17-jammy`, `8.4-jdk17-jammy`, `8-jdk17-jammy`, `jdk17-jammy`, `8.4.0-jdk-jammy`, `8.4-jdk-jammy`, `8-jdk-jammy`, `jdk-jammy`, `8.4.0-jammy`, `8.4-jammy`, `8-jammy`, `jammy`
- `8.4.0-jdk17-focal`, `8.4-jdk17-focal`, `8-jdk17-focal`, `jdk17-focal`, `8.4.0-jdk-focal`, `8.4-jdk-focal`, `8-jdk-focal`, `jdk-focal`, `8.4.0-focal`, `8.4-focal`, `8-focal`, `focal`

Spécifier l'image docker utilisée

`gradle:7.6.3-jdk17-alpine`

File: .gitlab-ci.yml

```yaml
image: gradle:7.6.3-jdk17-alpine

before_script:
  - GRADLE_USER_HOME="$(pwd)/.gradle"
  - export GRADLE_USER_HOME

build:
  stage: build
  script: gradle --build-cache assemble
  cache:
    paths:
      - build
      - .gradle

test:
  stage: test
  script: gradle check
  cache:
    paths:
      - build
      - .gradle
```

Résultat du job de build

```
22 Checking cache for default-non_protected...
23 No URL provided, cache will not be downloaded from shared cache server. Instead a local version of cache will be extracted.
24 Successfully extracted cache
26 Executing "step_script" stage of the job script
27 Using docker image sha256:f0333ac9633841520e3584b9fad4bdbdbcb5798daefa1ffb8545e59002d9cbe1 for gradle:7.6.3-jdk17-alpine with
   e24729a9397c1751fc60cdc50dd7f01acc69581c6c5f74b95079cd71e8bd297 ...
28 $ GRADLE_USER_HOME="$(pwd)/.gradle"
29 $ export GRADLE_USER_HOME
30 $ gradle --build-cache assemble
31 Starting a Gradle Daemon, 1 incompatible and 1 stopped Daemons could not be reused, use --status for details
32 > Task :app:compileJava
33 > Task :app:processResources NO-SOURCE
34 > Task :app:classes
35 > Task :app:jar
36 > Task :app:startScripts
37 > Task :app:distTar
38 > Task :app:distZip
39 > Task :app:assemble
40 BUILD SUCCESSFUL in 6s
41 5 actionable tasks: 5 executed
43 Saving cache for successful job
44 Creating cache default-non_protected...
45 WARNING: build: no matching files
46 .gradle: found 613 matching files and directories
47 No URL provided, cache will be not uploaded to shared cache server. Cache will be stored only locally.
48 Created cache
50 Cleaning up project directory and file based variables
52 Job succeeded
```

En fait, c'est une fausse bonne idée !

Ce n'est pas au script du pipeline de fixer les choses

- Cela doit se régler au niveau du fichier de build et de son appel dans le pipeline
- Sinon, le script ne suivra pas correctement les évolutions du build

File: .gitlab-ci.yml

```yaml
image: gradle:7.6.3-jdk17-alpine

before_script:
  - GRADLE_USER_HOME="$(pwd)/.gradle"
  - export GRADLE_USER_HOME

build:
  stage: build
  script: gradle --build-cache assemble
  cache:
    paths:
      - build
      - .gradle

test:
  stage: test
  script: gradle check
  cache:
    paths:
      - build
      - .gradle
```

Par ex. mettre à jour la version du wrapper Gradle du projet, implique de modifier le `.gitlab-ci.yml` ⇒ il faut éviter ce type de dépendance

1. Mieux paramétrer le build

1. Spécifier la version de Java dans le build : ▸ Java toolchain

build.gradle (app) ×

```
1 plugins {
2     id 'application'
3     id 'eclipse'
4 }
5
6 repositories {
7     mavenCentral()
8 }
9
10 java {
11     toolchain {
12         languageVersion = JavaLanguageVersion.of(17)
13     }
14 }
15
16 dependencies {
17     testImplementation 'org.junit.jupiter:junit-jupiter:5.9.1'
18 }
19
20 application {
21     mainClass = 'fr.iutmtp.cicdexample.App'
22 }
23 tasks.named('test') {
24     useJUnitPlatform()
25 }
```

2. Améliorer le script `.gitlab-ci.yml`

Appel de `./gradlew` dans le script

File: .gitlab-ci.yml

```yaml
image: gradle:8.4.0-alpine

before_script:
  - GRADLE_USER_HOME="$(pwd)/.gradle"
  - export GRADLE_USER_HOME

build:
  stage: build
  script: ./gradlew --build-cache assemble
  cache:
    paths:
      - build
      - .gradle

test:
  stage: test
  script: ./gradlew check
  cache:
    paths:
      - build
      - .gradle
```

Ce qui n'empêche pas de préciser un tag pour l'image utilisée, au contraire !

Résultat du job de build

```
26 Executing "step_script" stage of the job script                                    00:08
27 Using docker image sha256:a09086cffe0d4dfe4dba034da3448e77effbfbd0172d343f79dbea21897d4173 for gradle:8.4.0-alpine with digest gradle@sha256:8c855da
   901503d4bd6ea098a2574766f6469dd0105c7c74e4724d8377cbe23ae ...
28 $ GRADLE_USER_HOME="$(pwd)/.gradle"
29 $ export GRADLE_USER_HOME
30 $ ./gradlew --build-cache assemble
31 Starting a Gradle Daemon, 1 incompatible and 3 stopped Daemons could not be reused, use --status for details
32 > Task :app:compileJava FROM-CACHE
33 > Task :app:processResources NO-SOURCE
34 > Task :app:classes UP-TO-DATE
35 > Task :app:jar
36 > Task :app:startScripts
37 > Task :app:distTar
38 > Task :app:distZip
39 > Task :app:assemble
40 Deprecated Gradle features were used in this build, making it incompatible with Gradle 8.0.
41 You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own scripts or plugins.
42 See https://docs.gradle.org/7.6.3/userguide/command_line_interface.html#sec:command_line_warnings
43 BUILD SUCCESSFUL in 7s
44 5 actionable tasks: 4 executed, 1 from cache
46 Saving cache for successful job                                                      00:05
53 Cleaning up project directory and file based variables                             00:01
55 Job succeeded
```

Ce n’est pas fini : on a un warning !

Spécifier la Java toolchain jusqu'au bout !

En fait, il faut même spécifier comment résoudre la JVM qui doit être utilisée : ▸ Toolchain resolvers

```
CI_CD_gitlab_java_gradle on  3_specifying_Gradle_and_Java_versions via  v7.6.3 via ☕ v17.0.9
❯ ./gradlew --warning-mode all check
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=lcd
Java toolchain auto-provisioning enabled, but no java toolchain repositories declared by the build. Will rely on the built-in repository. This behaviour has been deprecated
and is scheduled to be removed in Gradle 8.0. In order to declare a repository for java toolchains, you must edit your settings script and add one via the toolchainManagemen
t block. See https://docs.gradle.org/7.6.3/userguide/toolchains.html#sec:provisioning for more details.

BUILD SUCCESSFUL in 367ms
3 actionable tasks: 3 up-to-date
```

Dans `settings.gradle` :

```
CI_CD_gitlab_java_gradle on  4_specifying_the_toolchain_completely via  v7.6.3 via ☕ v17.0.9
❯ bat settings.gradle
       File: settings.gradle
   1   /*
   2    * This file was generated by the Gradle 'init' task.
   3    */
   4   plugins {
   5       id 'org.gradle.toolchains.foojay-resolver-convention' version '0.7.0'
   6   }
   7
   8   rootProject.name = 'CI_CD_gitlab_java_gradle'
   9   include('app')
```

Plan

Quoi de neuf ?

`before_script` ?

`cache` ?

File: .gitlab-ci.yml

```yaml
image: gradle:8.4.0-alpine

before_script:
  - GRADLE_USER_HOME="$(pwd)/.gradle"
  - export GRADLE_USER_HOME

build:
  stage: build
  script: ./gradlew --build-cache assemble
  cache:
    paths:
      - build
      - .gradle

test:
  stage: test
  script: ./gradlew check
  cache:
    paths:
      - build
      - .gradle
```

before_script

Liste de commandes effectuées avant tous les scripts :

`before_script`

Use `before_script` to define an array of commands that should run before each job's `script` commands, but after artifacts are restored.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs: An array including:

- Single line commands.
- Long commands split over multiple lines.
- YAML anchors.

CI/CD variables are supported.

Example of `before_script`:

```
job:
  before_script:
    - echo "Execute this command before any 'script:' commands."
  script:
    - echo "This command executes after the job's 'before_script' commands."
```

Additional details:

- Scripts you specify in `before_script` are concatenated with any scripts you specify in the main `script`. The combined scripts execute together in a single shell.
- Using `before_script` at the top level, but not in the `default` section, is deprecated.

Définir GRADLE_USER_HOME

Variable d'environnement utilisée par Gradle pour sa configuration globale, le téléchargement des dépendances, ...

File: .gitlab-ci.yml

```yaml
image: gradle:8.4.0-alpine

before_script:
  - GRADLE_USER_HOME="$(pwd)/.gradle"
  - export GRADLE_USER_HOME

build:
  stage: build
  script: ./gradlew --build-cache assemble
  cache:
    paths:
      - build
      - .gradle

test:
  stage: test
  script: ./gradlew check
  cache:
    paths:
      - build
      - .gradle
```

Elle existe sur votre PC, pas dans l'image docker $\Rightarrow$ Il faut définir valeur pour pouvoir optimiser ensuite le pipeline

Au fait, appliquons le principe RTFM

Une lecture attentive de la documentation montre qu'on utilise une forme dépréciée dans l'écriture du script : ▸ `before_script`

`before_script`

Use `before_script` to define an array of commands that should run before each job's `script` commands, but after artifacts are restored.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs: An array including:

- Single line commands.
- Long commands split over multiple lines.
- YAML anchors.

CI/CD variables are supported.

Example of `before_script`:

```yaml
job:
  before_script:
    - echo "Execute this command before any 'script:' commands."
  script:
    - echo "This command executes after the job's 'before_script' commands."
```

Additional details:

- Scripts you specify in `before_script` are concatenated with any scripts you specify in the main `script`. The combined scripts execute together in a single shell.
- Using `before_script` at the top level, but not in the `default` section, is deprecated.

Utilisation du mot-clé `default`

Après correction :

File: .gitlab-ci.yml

```yaml
default:
  image: gradle:8.4.0-alpine

  before_script:
    - GRADLE_USER_HOME="$(pwd)/.gradle"
    - export GRADLE_USER_HOME

build:
  stage: build
  script: ./gradlew --build-cache assemble
  cache:
    paths:
      - build
      - .gradle

test:
  stage: test
  script: ./gradlew check
  cache:
    paths:
      - build
      - .gradle
```

Plan

`cache` et `cache:paths`

Objectif : accélérer (beaucoup) le pipeline ▸ `cache`

Use `cache` to specify a list of files and directories to cache between jobs. You can only use paths that are in the local working copy.

Caches are:

- Shared between pipelines and jobs.
- By default, not shared between protected and unprotected branches.
- Restored before artifacts.
- Limited to a maximum of four different caches.

You can disable caching for specific jobs, for example to override:

- A default cache defined with `default`.
- The configuration for a job added with `include`.

For more information about caches, see Caching in GitLab CI/CD.

`cache:paths`

Use the `cache:paths` keyword to choose which files or directories to cache.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Définir un identifiant par cache

`cache:key`

▸ cache:key

Use the `cache:key` keyword to give each cache a unique identifying key. All jobs that use the same cache key use the same cache, including in different pipelines.

If not set, the default key is `default`. All jobs with the `cache` keyword but no `cache:key` share the `default` cache.

Must be used with `cache: paths`, or nothing is cached.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- A string.
- A predefined CI/CD variable.
- A combination of both.

Example of `cache:key`:

```
cache-job:
  script:
    - echo "This job uses a cache."
  cache:
    key: binaries-cache-$CI_COMMIT_REF_SLUG
    paths:
      - binaries/
```

Définition globale du cache

Simplification avec la définition du cache dans `default` :

File: **.gitlab-ci.yml**

```yaml
default:
  image: gradle:8.4.0-alpine

  cache:
    key: gradle-$CI_COMMIT_REF_SLUG
    paths:
      - build
      - .gradle

  before_script:
    - GRADLE_USER_HOME="$(pwd)/.gradle"
    - export GRADLE_USER_HOME

build:
  stage: build
  script: ./gradlew --build-cache assemble

test:
  stage: test
  script: ./gradlew check
```

cache:policy

Plusieurs politiques possibles pour la mise à jour du cache

▸ cache:policy

`cache:policy`

To change the upload and download behavior of a cache, use the `cache:policy` keyword. By default, the job downloads the cache when the job starts, and uploads changes to the cache when the job ends. This caching style is the `pull-push` policy (default).

To set a job to only download the cache when the job starts, but never upload changes when the job finishes, use `cache:policy:pull`.

To set a job to only upload a cache when the job finishes, but never download the cache when the job starts, use `cache:policy:push`.

Use the `pull` policy when you have many jobs executing in parallel that use the same cache. This policy speeds up job execution and reduces load on the cache server. You can use a job with the `push` policy to build the cache.

Must be used with `cache: paths`, or nothing is cached.

Keyword type: Job keyword. You can use it only as part of a job or in the `default` section.

Possible inputs:

- `pull`
- `push`
- `pull-push` (default)

Exemple de définition `cache:policy`

Pas la peine de remplir le cache avec le résultat des tests :

Example of `cache:policy`:

```yaml
prepare-dependencies-job:
  stage: build
  cache:
    key: gems
    paths:
      - vendor/bundle
    policy: push
  script:
    - echo "This job only downloads dependencies and builds the cache."
    - echo "Downloading dependencies..."

faster-test-job:
  stage: test
  cache:
    key: gems
    paths:
      - vendor/bundle
    policy: pull
  script:
    - echo "This job script uses the cache, but does not update it."
    - echo "Running tests..."
```

Redéfinition locale de la politique

Redéfinition pour un job :

Inherit global configuration, but override specific settings per job

You can override cache settings without overwriting the global cache by using anchors. For example, if you want to override the `policy` for one job:

```yaml
default:
  cache: &global_cache
    key: $CI_COMMIT_REF_SLUG
    paths:
      - node_modules/
      - public/
      - vendor/
    policy: pull-push

job:
  cache:
    # inherit all global cache settings
    <<: *global_cache
    # override the policy
    policy: pull
```

▸ YAML anchors

Au final

File: .gitlab-ci.yml

```yaml
default:
  image: gradle:8.4.0-alpine

  cache: &global_cache
    key: gradle-$CI_COMMIT_REF_SLUG
    paths:
      - build
      - .gradle

  before_script:
    - GRADLE_USER_HOME="$(pwd)/.gradle"
    - export GRADLE_USER_HOME

build:
  stage: build
  script: ./gradlew --build-cache assemble

test:
  stage: test
  cache:
    # inherit all global cache settings
    <<: *global_cache
    # override the policy
    policy: pull
  script: ./gradlew check
```

Et l'option `--build-cache` ?

Gestion du cache par Gradle

L'efficacité de Gradle $\Rightarrow$ mécanisme de cache avancé

Overview

The Gradle *build cache* is a cache mechanism that aims to save time by reusing outputs produced by other builds. The build cache works by storing (locally or remotely) build outputs and allowing builds to fetch these outputs from the cache when it is determined that inputs have not changed, avoiding the expensive work of regenerating them.

A first feature using the build cache is *task output caching*. Essentially, task output caching leverages the same intelligence as up-to-date checks that Gradle uses to avoid work when a previous local build has already produced a set of task outputs. But instead of being limited to the previous build in the same workspace, task output caching allows Gradle to reuse task outputs from any earlier build in any location on the local machine. When using a shared build cache for task output caching this even works across developer machines and build agents.

Gestion du cache par Gradle

Enable the Build Cache

By default, the build cache is not enabled. You can enable the build cache in a couple of ways:

Run with `--build-cache` on the command-line

Gradle will use the build cache for this build only.

Put `org.gradle.caching=true` in your `gradle.properties`

Gradle will try to reuse outputs from previous builds for all builds, unless explicitly disabled with `--no-build-cache`.

When the build cache is enabled, it will store build outputs in the Gradle User Home. For configuring this directory or different kinds of build caches see Configure the Build Cache.

Faut-il rajouter un fichier `gradle.properties` dans le projet, avec cette option, plutôt que de la rajouter dans le script ?

Les raisons de ne pas le faire

Cette option a une vraie plus value en local, sur votre poste, mais. . .

Cela peut ralentir le pipeline

- le cache peut devenir énorme au fil du temps
- les éléments inutiles vont encombrer le serveur
- plus le cache est gros, plus il est long à mettre en place dans un job
- $\Rightarrow$ il faut limiter son contenu au minimum

Cela peut rendre le pipeline indéterministe

- une même entrée doit produire les mêmes résultats
- si le cache est "trop partagé", son contenu peut devenir incohérent
- $\Rightarrow$ il faut distinguer les caches avec des clés

De l'importance de bien distinguer sa config locale de celle du pipeline

Le dernier exemple touche un point très important : bien distinguer pipeline CI vs. pipeline local

- Gradle fonctionne très bien tout seul, ce qui implique une configuration locale qui vous est propre
- $\Rightarrow$ localisation des identifiants
- $\Rightarrow$ cache des dépendances
- $\Rightarrow$ cache des build du projets
- $\Rightarrow$ etc.

Tout doit être recréé/géré dans un pipeline de CI

Plan

1. Upload du jar produit par Gradle

```yaml
stages:
  - build
  - test
  - package
  - deploy

default:
  image: gradle:8.4.0-alpine

  cache:
    key: $CI_COMMIT_REF_SLUG
    paths:
    - .gradle/wrapper
    - .gradle/caches
    - app/build

  before_script:
    - GRADLE_USER_HOME="$(pwd)/.gradle"
    - export GRADLE_USER_HOME

variables:
  VERSION: "1.0"
  PACKAGE_NAME: "app-$VERSION.jar"

build:
  stage: build
  script: ./gradlew --build-cache assemble


unittest-job:
  stage: test
  script: ./gradlew check

run-test-job:
  stage: test
  script: ./gradlew run

package-job:
  stage: package
  script: ./gradlew -Pversion="$VERSION" jar
  artifacts:
    paths:
      - app/build/libs/$PACKAGE_NAME

deploy:
  image: finalgene/openssh
  stage: deploy
  cache: {}
  script:
    - echo "<a href=\"$PACKAGE_NAME\">Awesome App Last version -> $PACKAGE_NAME</a>" > index.html
    - apk add lftp #install the lftp package on the alpine distro
    - lftp -u $IUT_USERNAME,$IUT_PASSWORD -e "set ftp:ssl-allow 0; cd public_html/cicd; put app/build/libs/$PACKAGE_NAME index.html; bye;" sftp://ftpinfo.iutmontp.univ-mo[illegible]2
```

Déploiement d'une image docker

Création d'un fichier `Dockerfile` dans le répertoire `app`

File: app/Dockerfile

```
1  FROM eclipse-temurin:17-jdk-alpine
2
3  WORKDIR Application
4
5  COPY gradle/ gradle
6  COPY .gradle/ .gradle
7  COPY app/ app
8  COPY gradlew settings.gradle ./
9
10 RUN pwd
11 RUN ls -lsa
12 RUN ./gradlew clean jar
13
14 CMD ["./gradlew","run"]
```

2. Déploiement d'une image docker

Ici nous utilisons le registry Harbor de l'IUT : ▸ Harbor

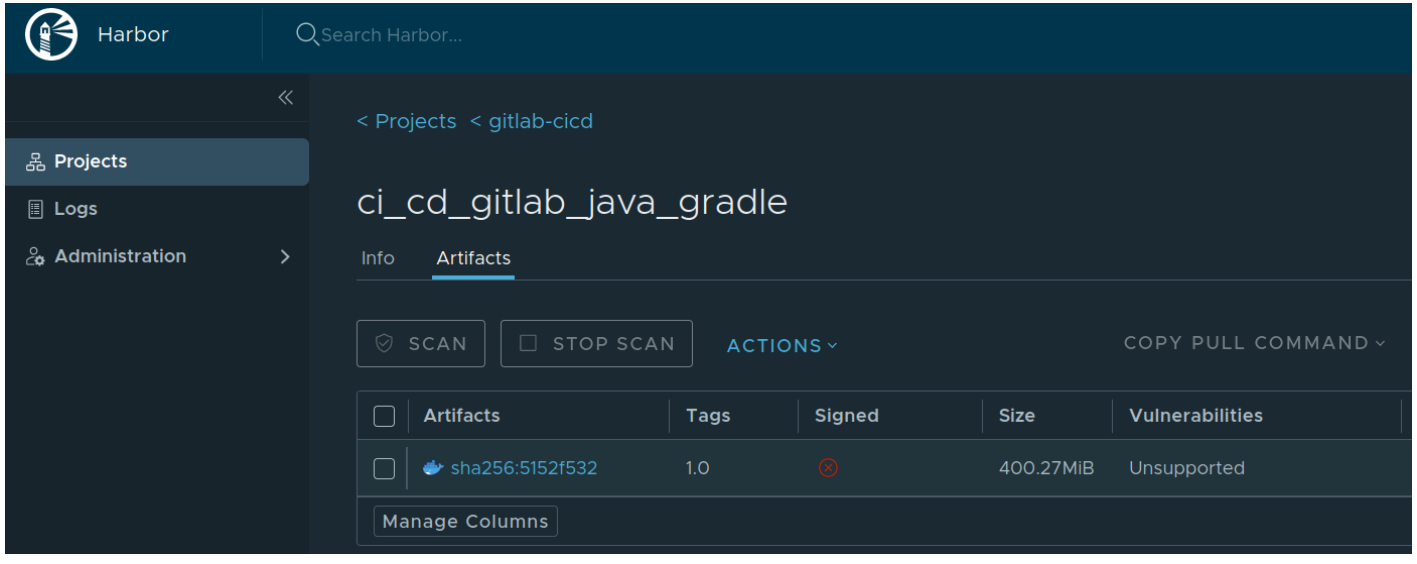
Cet exemple nécessite un projet `gitlab-cicd` sur Harbor

File: .gitlab-ci.yml

```yaml
default:
  image: gradle:8.4.0-alpine

  cache:
    key: $CI_COMMIT_REF_SLUG
    paths:
      - .gradle/wrapper
      - .gradle/caches
      - app/build

  before_script:
    - GRADLE_USER_HOME="$(pwd)/.gradle"
    - export GRADLE_USER_HOME

variables:
  VERSION: "1.0"

build:
  stage: build
  script: ./gradlew --build-cache assemble

test-job:
  stage: test
  script: ./gradlew check

deploy:
  stage: deploy
  image:
    name: gcr.io/kaniko-project/executor:debug
    entrypoint: ['']
  script:
    - mkdir -p /kaniko/.docker
    - echo "{\"auths\":{\"${CI_REGISTRY_URL}\":{\"auth\":\"$(echo -n ${CI_REGISTRY_USER}:${CI_REGISTRY_PASSWORD} | base64)\"}}}" > /kaniko/.docker/config.json
    - >-
      /kaniko/executor
      --context "${CI_PROJECT_DIR}"
      --dockerfile "${CI_PROJECT_DIR}/app/Dockerfile"
      --destination "${CI_REGISTRY_URL}/gitlab-cicd/${CI_PROJECT_NAME}:${VERSION}"
```

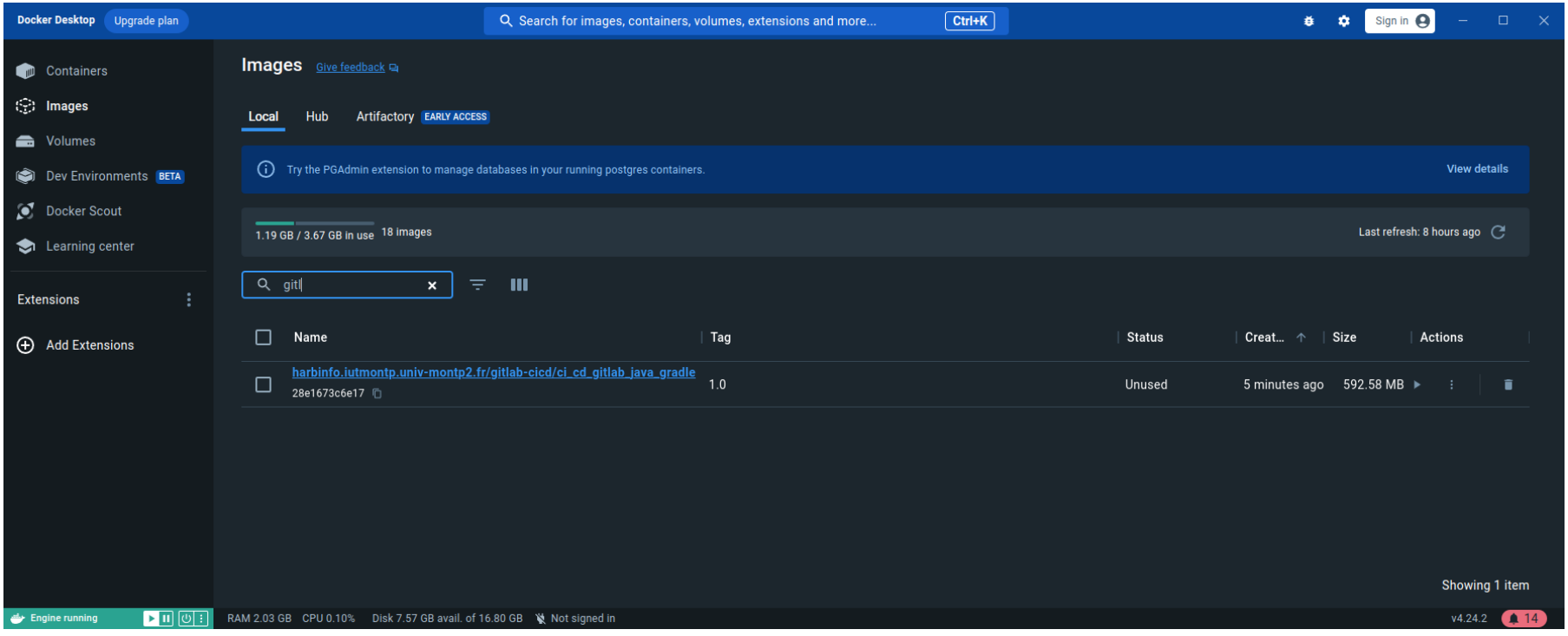
Méthode la plus simple : ▸ Use kaniko to build Docker images

2. Déploiement d'une image docker




```
docker pull harbinfo.iutmontp.univ-montp2.fr/gitlab-cicd/ci_cd_gitlab_java_gradle:1.0
1.0: Pulling from gitlab-cicd/ci_cd_gitlab_java_gradle
96526aa774ef: Already exists
cc3c4df72d74: Already exists
d7937dfaa0fb: Already exists
9ec44800a653: Already exists
eb5fcbc2e814: Already exists
9fa91ab38bbb: Pull complete
136c6b59a638: Pull complete
c09e4749e14a: Pull complete
3216b46886d1: Pull complete
85db817c9651: Pull complete
9441fe8717c1: Pull complete
Digest: sha256:5152f53223a797ee33badd7f425e2c7064ec05cc36ec62a770cf840f35f0f044
Status: Downloaded newer image for harbinfo.iutmontp.univ-montp2.fr/gitlab-cicd/ci_cd_gitlab_java_gradle:1.0
harbinfo.iutmontp.univ-montp2.fr/gitlab-cicd/ci_cd_gitlab_java_gradle:1.0

~ took 2m16s
```

2. ⇒ en local sous docker desktop



Taille de l'image : 592 Mb !!

3. Plus simple, plus léger

Pas besoin de Gradle dans l'image finale ; on utilise seulement le fichier jar produit
Du coup on prend une image plus petite :

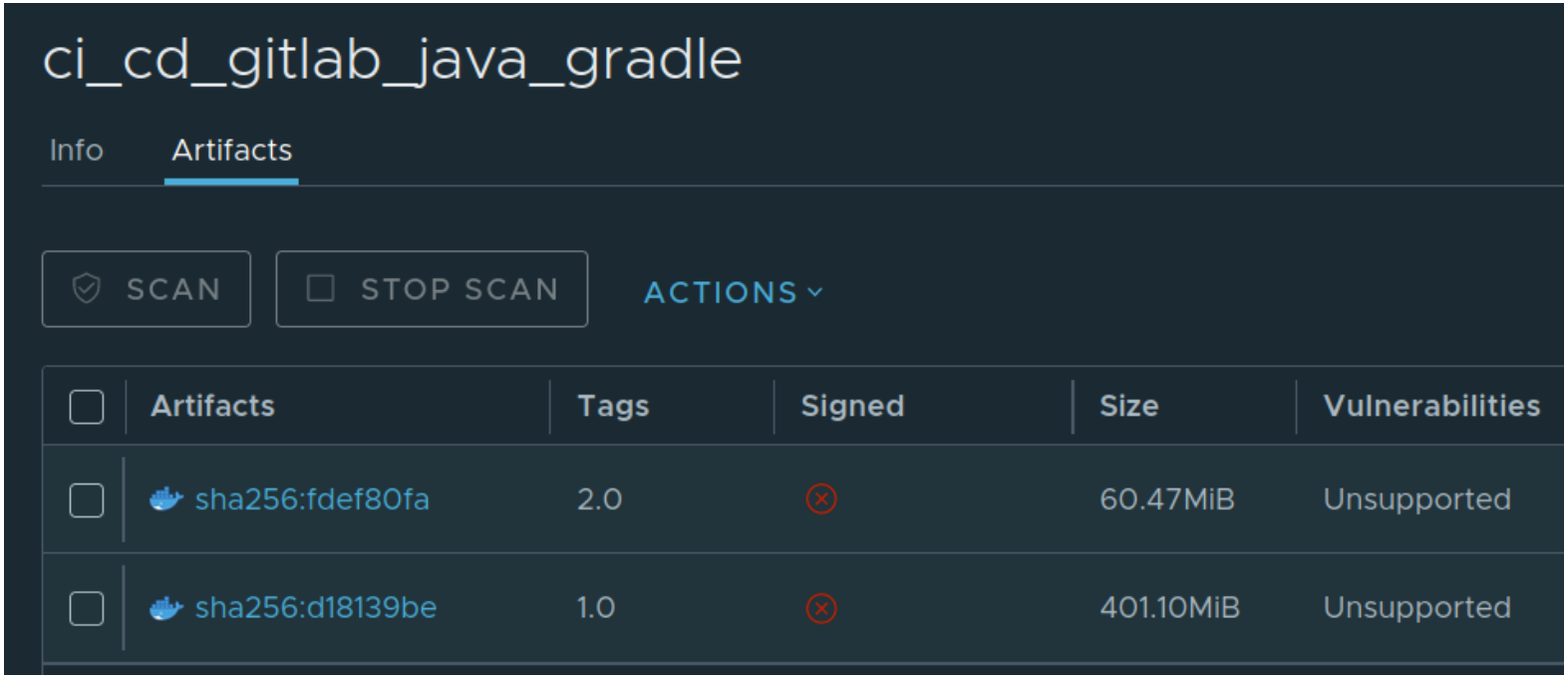
File: app/Dockerfile

```
1  FROM eclipse-temurin:17.0.9_9-jre-alpine
2
3  WORKDIR Application
4
5  ARG PACKAGE
6
7  RUN echo "pckg name is $PACKAGE"
8  ENV JAR_FILE="$PACKAGE"
9  RUN echo "jar file is $JAR_FILE"
10 COPY "app/build/libs/$PACKAGE" .
11 RUN ls -lsa
12
13 CMD java -jar "$JAR_FILE"
```

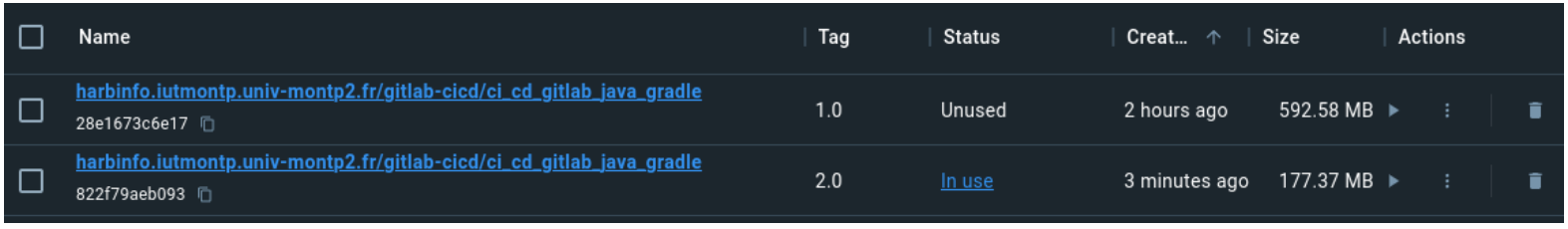
3. Plus simple, plus léger

```yaml
  - build
  - test
  - package
  - integration-test
  - deploy

default:
  image: gradle:8.4.0-alpine

  cache:
    key: $CI_COMMIT_REF_SLUG
    paths:
      - .gradle/wrapper
      - .gradle/caches
      - app/build

  before_script:
    - GRADLE_USER_HOME="$(pwd)/.gradle"
    - export GRADLE_USER_HOME

variables:
  VERSION: "2.0"
  PACKAGE_NAME: "app-$VERSION.jar"

build:
  stage: build
  script: ./gradlew --build-cache assemble

test-job:
  stage: test
  script: ./gradlew check

package-job:
  stage: package
  script: ./gradlew -Pversion="$VERSION" jar
  artifacts:
    paths:
      - app/build/libs/$PACKAGE_NAME

integration-test-job:
  stage: integration-test
  cache: {}
  script: java -jar app/build/libs/$PACKAGE_NAME

deploy:
  stage: deploy
  image:
    name: gcr.io/kaniko-project/executor:debug
    entrypoint: ['']
  script:
    - mkdir -p /kaniko/.docker
    - echo "{\"auths\":{\"${CI_REGISTRY_URL}\":{\"auth\":\"$(echo -n ${CI_REGISTRY_USER}:${CI_REGISTRY_PASSWORD} | base64)\"}}}" > /kaniko/.docker/config.json
    - >-
      /kaniko/executor
      --context "${CI_PROJECT_DIR}"
      --dockerfile "${CI_PROJECT_DIR}/app/Dockerfile"
      --destination "${CI_REGISTRY_URL}/gitlab-cicd/${CI_PROJECT_NAME}:${VERSION}"
      --build-arg PACKAGE=${PACKAGE_NAME}
      -v "info"
```

3. Plus simple, plus léger



En local, sous docker desktop :



4. On peut faire encore mieux !

Lire la documentation des images est toujours bénéfique !
On trouve une meilleure solution sur celle de ▸ eclipse-temurin

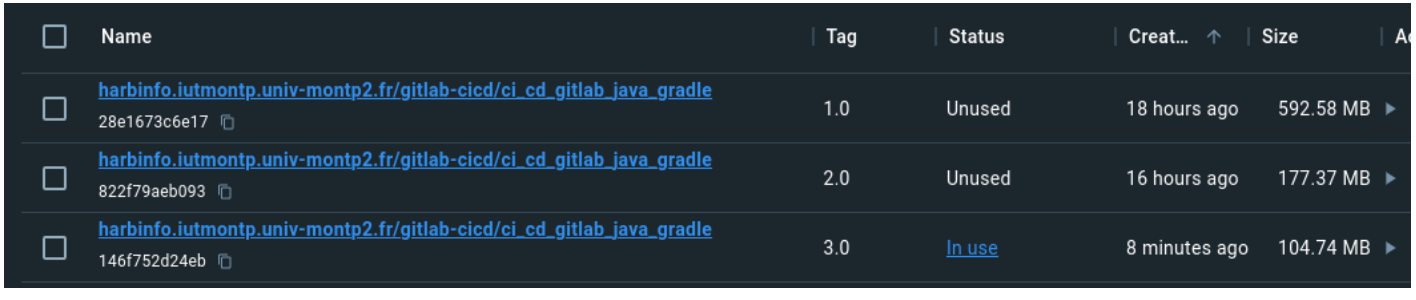
File: app/Dockerfile

```
1  FROM eclipse-temurin:17 as jre-build
2
3  # Create a custom Java runtime
4  RUN $JAVA_HOME/bin/jlink \
5          --add-modules java.base \
6          --strip-debug \
7          --no-man-pages \
8          --no-header-files \
9          --compress=2 \
10         --output /javaruntime
11
12
13 # Define your base image
14
15 FROM debian:buster-slim
16
17 # Must define the ARG in this second stage
18 ARG PACKAGE
19
20 ENV JAR_FILE_LOCATION="/opt/app/$PACKAGE"
21
22 ENV JAVA_HOME=/opt/java/openjdk
23 ENV PATH "${JAVA_HOME}/bin:${PATH}"
24 COPY --from=jre-build /javaruntime $JAVA_HOME
25
26 # Continue with your application deployment
27 RUN mkdir /opt/app
28 COPY app/build/libs/$PACKAGE /opt/app
29
30 CMD java -jar "$JAR_FILE_LOCATION"
```

4. On peut faire encore mieux !

Sur Harbor :

Sur Docker Desktop :



Conclusion

IRL, un pipeline doit utiliser un moteur de production

- $\Rightarrow$ éviter au maximum les scripts bas niveau
- $\Rightarrow$ appels des tâches fournies par le moteur
- $\Rightarrow$ les dépendances sont gérées par le moteur

L'important est de distinguer le pipeline du moteur de celui du CICD

- la frontière peut être floue, car un moteur de prod peut faire du CD (e.g. construire et pousser une image docker)
- En règle générale, le plus cohérent est de distinguer les étapes de construction/test/documentation, scriptées dans le build du moteur, des étapes de distribution/versioning, à scripter dans le pipeline CICD