CI/CD

LA SOLUTION GITLAB CI

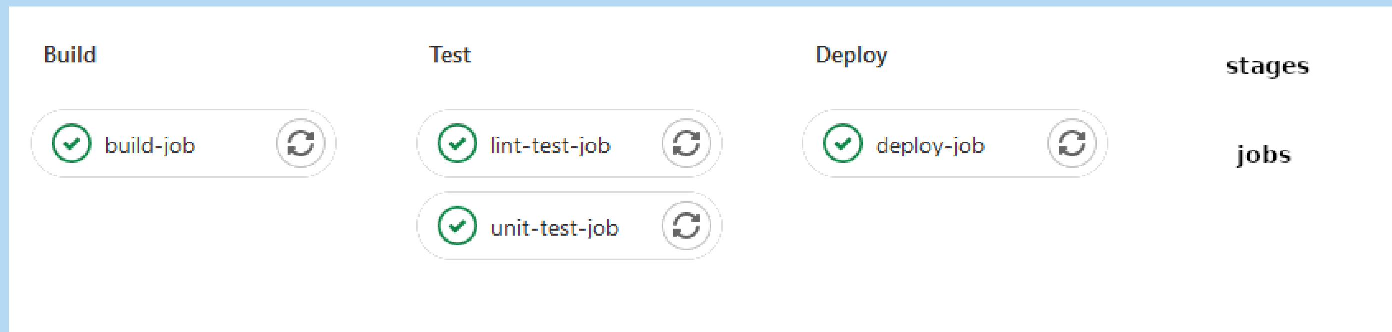
LA SOLUTION GITLAB CI

GitLab CI/CD, c'est quoi ?

- Une des fonctionnalités de la suite logicielle GitLab
- Une solution pour automatiser l'intégration et le déploiement de vos projets logiciels
- Une fonctionnalité qui repose sur plusieurs notions clés
  - Le pipeline
  - le stage
  - le job
  - le runner

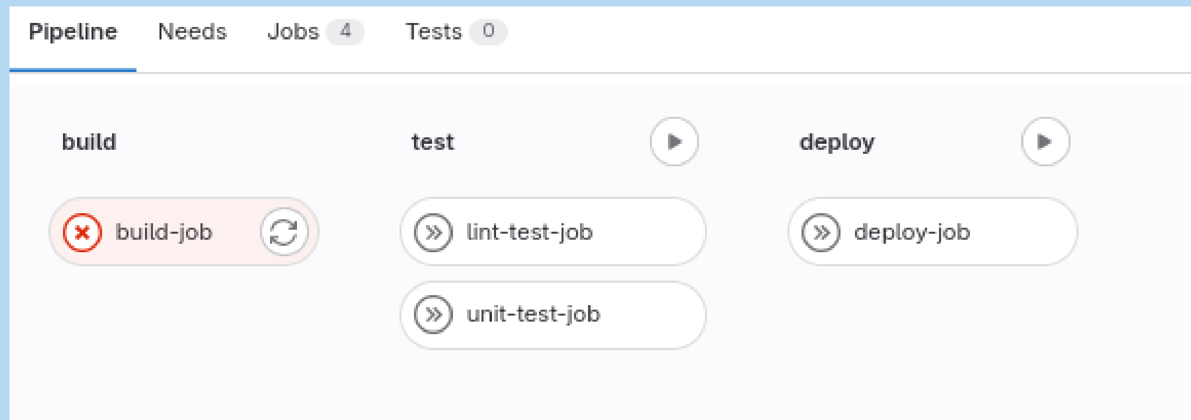
LE PIPELINE

- Le pipeline est le composant de premier niveau de l’intégration, de la livraison et du déploiement continus de GitLab.
- Un pipeline est constitué de différentes étapes séquentielles nommées `stage`
- Chaque stage pourra contenir N tâches que l'on appelera `job`
- Ces jobs pourront s'exécuter en parallèle au sein d'un `stage`



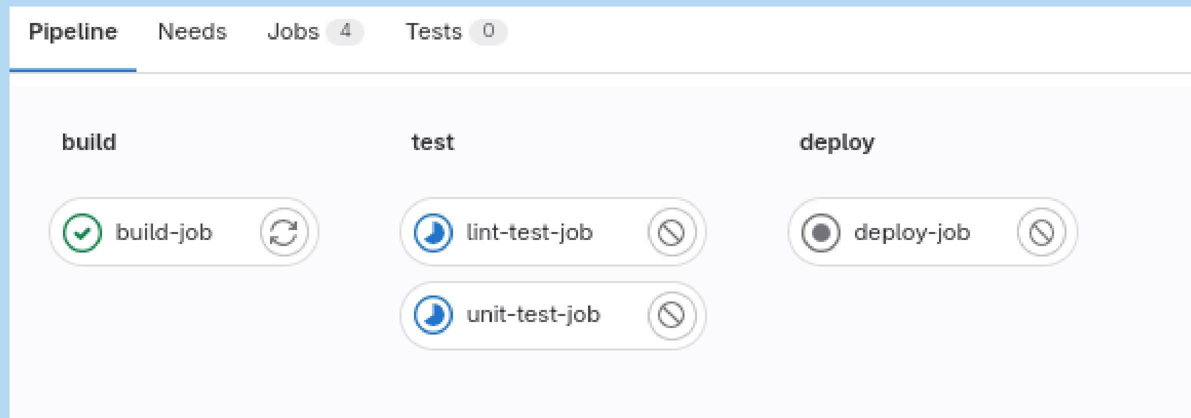
LE STAGE

- `Les stages` sont simplement une division logique entre des ensembles de `jobs`
- `Le stages` suivant ne commence que lorsque toutes les tâches de l’étape précédente sont terminées sans erreurs.
- Il suffit que l’un des `stage` pour que l’ensemble du pipeline échoue à part quelques cas particuliers



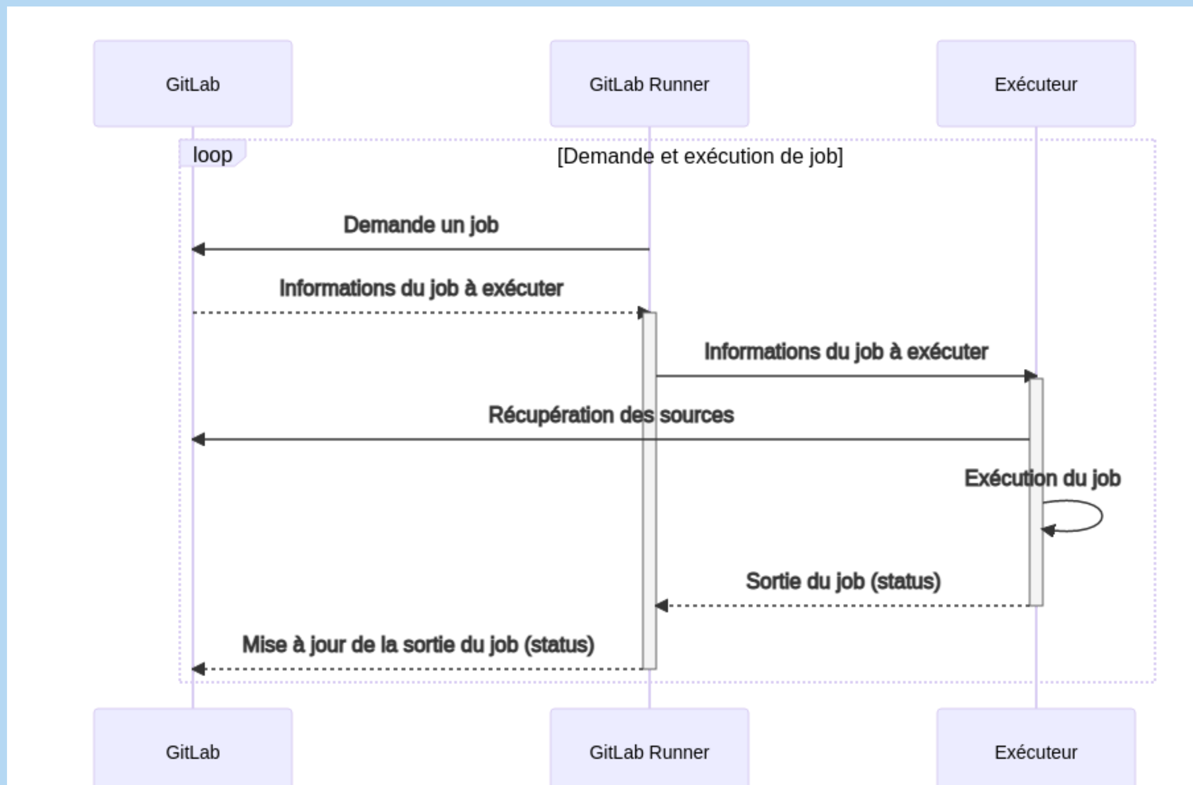
LE JOB

- Un job est un ensemble d’instructions qu’un runner doit exécuter
- Il est systématiquement rattaché à un `stage`
- Chaque stage pourra contenir N tâches que l'on appelera `job`
- Ces jobs pourront s'exécuter en parallèle au sein d'un `stage`
- Ces jobs pourront produire des fichiers et/ou dossiers qui vont être stockés au sein des pipelines pour être utilisés par d’autres tâches : les `artefacts`



LE RUNNER

- Un `job` s'exécute via un `GitLab Runner` qui est une application indépendante de GitLab.
- Le runner est un simple démon qui attend les jobs
- `Un runner` va déléguer l'exécution d'un job à un exécuteur.



L'EXÉCUTEUR

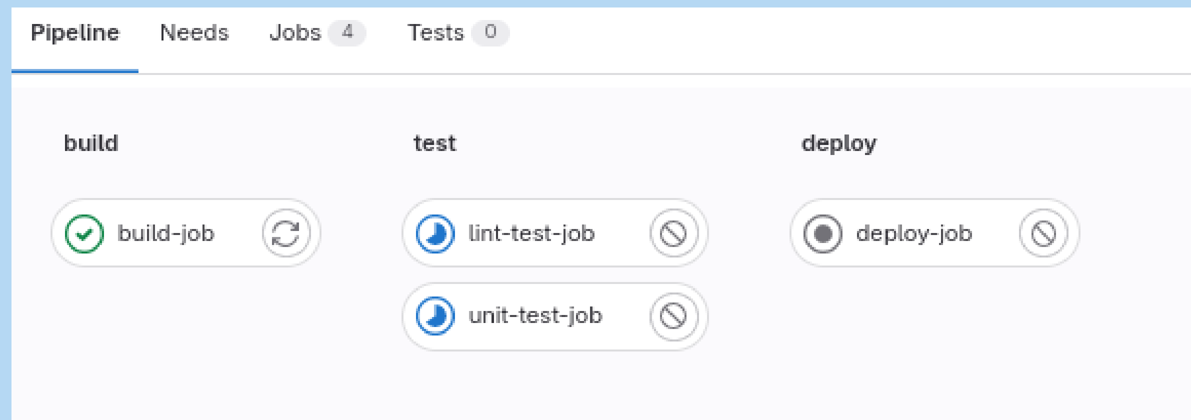
- Les exécuteurs sont des sous-processus qui vont se charger de faire les commandes (scripts) que vous avez définies dans votre gitlab-ci
- L'exécuteur peut être de plusieurs types :
  - `Shell` : sos scripts seront lancés sur la machine qui possède le Runner
  - `Parallels, VirtualBox` : Le Runner va créer une machine virtuelle pour exécuter les script
  - `Docker` : Utilise Docker pour créer / exécuter vos scripts et traitement
  - `Docker Machine` : Identique à docker, mais dans un environnement Docker multimachine avec auto-scaling
  - `Kubernetes` : Lance vos builds dans un cluster Kubernetes
  - `SSH` : il permet à Gitlab-CI de gérer l'ensemble des configurations possibles via des commandes ssh
- Pour nos cas d'usages et le TP, nous utiliserons un exécuteur `Docker`

LES RUNNERS - USAGE

- Par défaut , des runners partagés par les projets et utilisateurs sont à votre disposition sans aucune installation, ni configuration : les Shared runners
- Pour optimiser vos temps d'attentes et d'exécution, ou installer des configurations spécifiques, vous pouvez déployer vos propres runners pour un projet, un groupe
- 1ère étape - Installation de votre runner : https://docs.gitlab.com/runner/install/
- 2 ème étape - Enregistrement de votre runner : https://docs.gitlab.com/runner/register/index.html

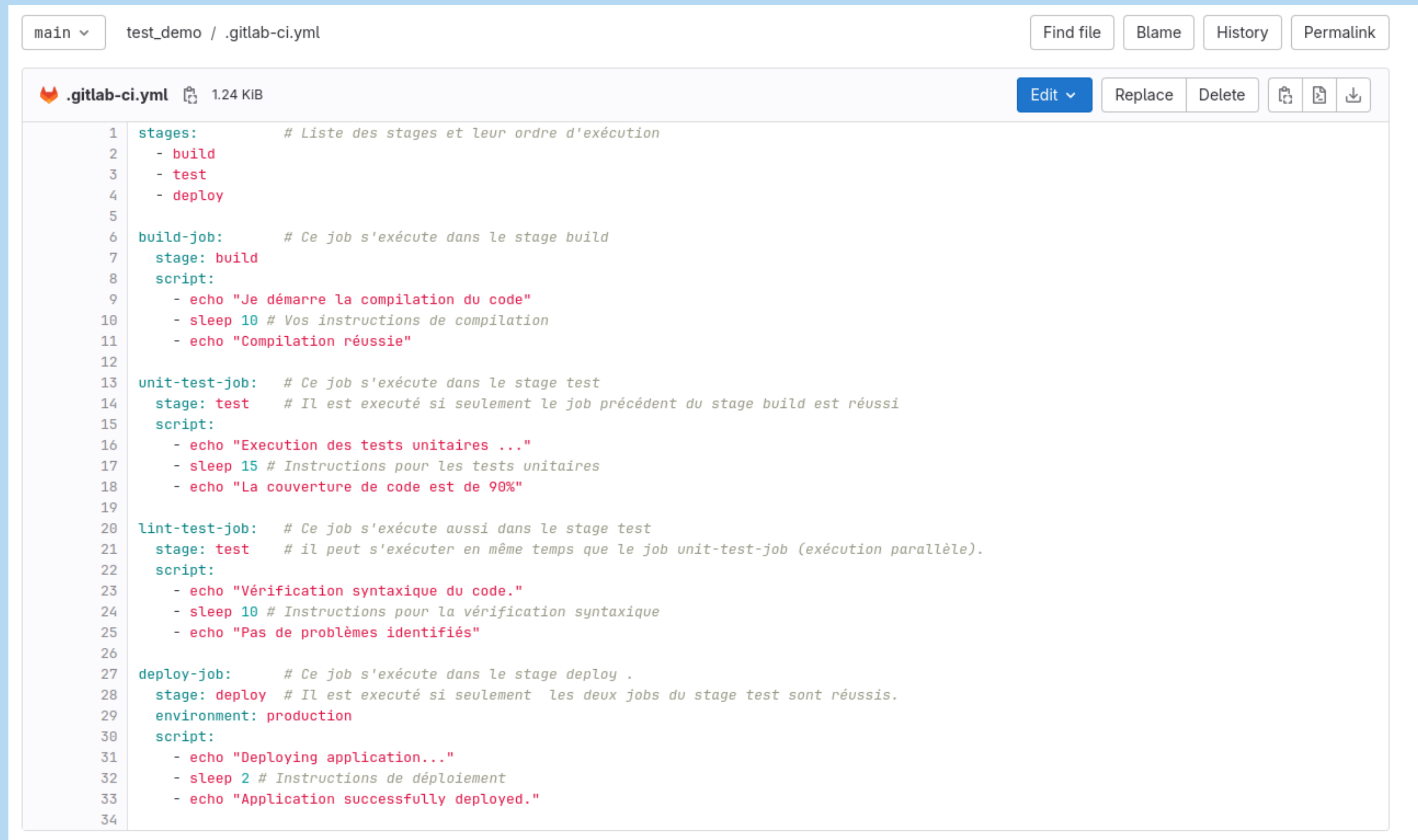
GITLAB CI : DÉROULEMENT

- Par défaut, les stages se déroulent de façon séquentielle dans l'ordre de déclaration
- Les jobs d'un stage s'exécutent de façon parallèle si les ressources le permettent
- Par défaut, au sein d'un job chaque script qui échoue arrête l’exécution du pipeline
- Le critère d'échec d'une execution de script est un exit code différent de 0



MISE EN PLACE D'UN PIPELINE : DÉMARRAGE

- Il faut déclarer un manifeste .gitlab-ci.yml à la racine du projet - éditable dans Gitlab
- Ce fichier basé sur une syntaxe yaml contient la description du pipeline
- Dans ce manifeste vous allez pouvoir définir des stages, et des jobs




```yaml
stages:          # Liste des stages et leur ordre d'exécution
  - build
  - test
  - deploy

build-job:       # Ce job s'exécute dans le stage build
  stage: build
  script:
    - echo "Je démarre la compilation du code"
    - sleep 10 # Vos instructions de compilation
    - echo "Compilation réussie"

unit-test-job:   # Ce job s'exécute dans le stage test
  stage: test    # Il est executé si seulement le job précédent du stage build est réussi
  script:
    - echo "Execution des tests unitaires ..."
    - sleep 15 # Instructions pour les tests unitaires
    - echo "La couverture de code est de 90%"

lint-test-job:   # Ce job s'exécute aussi dans le stage test
  stage: test    # il peut s'exécuter en même temps que le job unit-test-job (exécution parallèle).
  script:
    - echo "Vérification syntaxique du code."
    - sleep 10 # Instructions pour la vérification syntaxique
    - echo "Pas de problèmes identifiés"

deploy-job:      # Ce job s'exécute dans le stage deploy .
  stage: deploy  # Il est executé si seulement  les deux jobs du stage test sont réussis.
  environment: production
  script:
    - echo "Deploying application..."
    - sleep 2 # Instructions de déploiement
    - echo "Application successfully deployed."

```

ETAPE 1 : DÉFINISSEZ LES ÉTAPES DU PIPELINE

- Dans un premier temps, je définis les **étapes** de mon pipeline avec le mot clé `stages`.
- Ce mot clé permet de définir l'ordre des étapes.
- Ici, la première étape va être le `build`, et ensuite le stage `test` puis le stage `deploy`

```
1  stages:          # Liste des stages et leur ordre d'exécution
2    - build
3    - test
4    - deploy
5
6  build-job:       # Ce job s'exécute dans le stage build
7    stage: build
8    script:
9      - echo "Je démarre la compilation du code"
10     - sleep 10 # Vos instructions de compilation
11     - echo "Compilation réussie"
12
13 unit-test-job:   # Ce job s'exécute dans le stage test
14   stage: test    # Il est executé si seulement le job précédent du stage build est réussi
15   script:
16     - echo "Execution des tests unitaires ..."
17     - sleep 15 # Instructions pour les tests unitaires
18     - echo "La couverture de code est de 90%"
19
20 lint-test-job:   # Ce job s'exécute aussi dans le stage test
21   stage: test    # il peut s'exécuter en même temps que le job unit-test-job (exécution parallèle).
22   script:
23     - echo "Vérification syntaxique du code."
24     - sleep 10 # Instructions pour la vérification syntaxique
25     - echo "Pas de problèmes identifiés"
26
27 deploy-job:      # Ce job s'exécute dans le stage deploy .
28   stage: deploy  # Il est executé si seulement  les deux jobs du stage test sont réussis.
```

ETAPE 2 : DÉFINISSEZ LES JOBS À EFFECTUER

- `stage` : le nom de l'étape qui va apparaître dans notre pipeline d'intégration continue. Cela correspond aussi au `stage` auquel sera exécuté le job
- `script` : ce sont les lignes de script à lancer afin d'exécuter l'étape. L'installation et l'exécution de pylint pour la partie `lint` et dans la partie `test`, nous lançons les tests unitaires. Si un seul de ces tests échoue, le pipeline s'arrête.

```
1  stages:          # Liste des stages et leur ordre d'exécution
2    - build
3    - test
4    - deploy
5
6  build-job:       # Ce job s'exécute dans le stage build
7    stage: build
8    script:
9      - echo "Je démarre la compilation du code"
10     - sleep 10 # Vos instructions de compilation
11     - echo "Compilation réussie"
12
13 unit-test-job:   # Ce job s'exécute dans le stage test
14   stage: test    # Il est executé si seulement le job précédent du stage build est réussi
15   script:
16     - echo "Execution des tests unitaires ..."
17     - sleep 15 # Instructions pour les tests unitaires
18     - echo "La couverture de code est de 90%"
19
20 lint-test-job:   # Ce job s'exécute aussi dans le stage test
21   stage: test    # il peut s'exécuter en même temps que le job unit-test-job (exécution parallèle).
22   script:
23     - echo "Vérification syntaxique du code."
24     - sleep 10 # Instructions pour la vérification syntaxique
25     - echo "Pas de problèmes identifiés"
26
27 deploy-job:      # Ce job s'exécute dans le stage deploy .
28   stage: deploy  # Il est executé si seulement  les deux jobs du stage test sont réussis.
```

LE MOT CLÉ : `image`

- Par défaut et si le mot clé n'est pas spécifié l'image docker:latest est utilisée pour exécuter les scripts
- Sinon vous pouvez la spécifier globalement ou pour chaque `stage`

```yaml
default:
   image: python:latest # déclaration de l'image executée par défaut
stages:
  - build

build-job:
  stage: build
  script:
    - echo "Je démarre la compilation du code"
    - sleep 10
    - echo "Compilation réussie"
```

```yaml
stages:
  - build

build-job:
  stage: build
  image: python:3 # déclaration de l'image executée pour ce stage
  script:
    - echo "Je démarre la compilation du code"
    - sleep 10
    - echo "Compilation réussie"
```

USAGE DES VARIABLES DANS LE MANIFESTE

- Variables customisées déclarées dans les préférences CI/CD du projet ou du groupe
- Variables prédéfinies liées à l'environnement Gitlab

```
stage: test
  script:
    - echo "$CI_JOB_STAGE"
    - echo "$CI_PROJECT_NAME" > project_name.txt
    - echo "$GITLAB_USER_ID"
    - echo "$DB_PASSWORD"
```

- Déclaration dans l'interface Gitlab : Settings > CI/CD > Variables

Variables

Collapse

Variables are applied to environments via the runner. They can be protected by only exposing them to protected branches or tags. You can use variables for passwords, secret keys, or whatever you want.

Key	Value		Protected	Environment
BD_PASSWORD	db_password_production	Protected	on	production
DB_PASSWORD	db_password_demo	Protected	on	demo
DB_PASSWORD	db_password_no_env	Protected	off	All environments
Input variable key	Input variable value	Protected	off	All environments

Save variables | Hide values

CONTRAINTES : ONLY ET EXCEPT

- A l'aide des mots clés `only`, `except` vous pouvez créer des contraintes sur l'exécution d’une tâche
  - `branches` déclenche le `job` en cas de push sur la branche spécifiée.
  - `tags` déclenche le `job` quand un tag est créé.
  - `api` déclenche le `job` suite à un appel via l'API
  - `external` déclenche le `job` grâce à un service de CI/CD autre que GitLab.
  - `merge_requests` déclenche le `job` suite à une merge request
  - `pushes` déclenche le `job` quand un `push` est effectué par un utilisateur.
  - `schedules` déclenche le `job` par rapport à une planification (UI)
  - `triggers` déclenche le `job` par rapport à un jeton de déclenchement.
  - `web` déclenche le `job` par rapport au bouton `Run pipeline` (UI)

CONTRAINTES ONLY ET EXCEPT (DÉPRÉCIÉS) : EXEMPLES

```yaml
stages:
  - build

build-job-prod:
  stage: build
  script:
    - ./build_prod.sh
  only:
    - main # Le job sera effectué uniquement sur la branche main
    - merge_requests # et sur merge requests

build-job-except-prod:
  stage: build
  script:
    - ./build.sh
  except:
    - main # Le job sera effectué sur mes branches sauf main
    - tags # sauf en cas de tags
    variables:
      - $RELEASE == "prod" # sauf quand la variable release == 'prod'
```

LES RULES

- méchanisme de remplacement des only/except
- permet de tester les variables et de faire des combinaisons avec && et ||
- Exemple 1 : une regex sur le message de commit et le nom de la branche

```
build-job:
  stage: build
  script:
    - echo "Je démarre la compilation du code en prod"
    - sleep 10 # Vos instructions de compilation
    - echo "Compilation réussie
  rules:
    - if: '$CI_COMMIT_MESSAGE =~ /my_commit_message/ && $CI_COMMIT_BRANCH == "master"'
```

- Exemple 2 : sur changement de fichiers ou sur la présence de fichier

```
build-job:
  stage: build
  script:
    - echo "Je démarre la compilation du code en prod"
    - sleep 10 # Vos instructions de compilation
    - echo "Compilation réussie
  rules:
      - changes:
        - src/*/*
      - exists:
        - build.war
```

CONTRAINTES : WHEN

la directive `when` exprime contrainte sur l'exécution de la tâche.

- `on_success` : le job sera exécuté uniquement si tous les `jobs` du stage précédent sont passés
- `on_failure` : le job sera exécuté uniquement si un job est en échec
- `always` : le job s'exécutera quoi qu'il se passe (même en cas d’échec)
- `manual` : le job s'exécutera uniquement par une action manuelle
- `delayed` le job sera retardé pendant une durée spécifiée.
- `never` le job ne sera pas exécuté dans ce cas

CONTRAINTES WHEN : EXEMPLES

```
stages:
- build
- test
- report
- clean

job:build:
stage: build
script:
- make build

job:test:
stage: test
script:
- make test
when: on_success # s'exécutera uniquement si le job `job:build` passe

job:report:
stage: report
script:
- make report
when: on_failure # s'exécutera si le job `job:build` ou `job:test` ne passe pas

job:clean:
stage: clean
script:
- make clean # s'exécutera quoi qu'il se passe
when: always
```

LES TEMPLATES

Vous pouvez factoriser des actions grâce au mot clé `extends` et un job caché (.*)

```yaml
stages :
     - build

.build :
     stage : build
     variables :
          VERSION : 1.0
     script :
          - echo $BUILD_NAME
          - echo $VERSION

buildA :
     extends : .build
     variables :
          BUILD_NAME : "Build A"

buildB :
     extends : .build
     variables :
          BUILD_NAME : "Build B"
          VERSION : 1.1
```

LES ANCRES

- Déclarer une configuration et l'appeler plusieurs fois

```yaml
.job_template: &job_definition
  image: ruby:2.6
  services:
    - postgres
    - redis

test1:
  <<: *job_definition
      script:
      - test1 project

      test2:
  <<: *job_definition
      script:
      - test2 project
```

PARAMÉTRER UN TEMPLATE

```yaml
# template-ci/install.yml

.install:
  variables:
    INSTALL_DIRECTORY: '.'
  script:
    - cd $INSTALL_DIRECTORY
    - npm install
```

```yaml
# mon-app1/.gitlab-ci.yml

include:
  - project: 'template-ci'
    file: 'install.yml'
    ref: 'master'

projet1:
  extends: .install
  variables:
    INSTALL_DIRECTORY: 'projet1/'

projet2:
  extends: .install
  variables:
    INSTALL_DIRECTORY: 'projet2/'
```

LES ARTEFACTS

- Fichiers ou/et répertoires stockés au sein des pipelines et utilisés par d’autres tâches
- Produits qu'en cas de réussite du job et sont déclarés dans la CI
- `paths*` : permet de spécifier la liste des fichiers et/ou dossiers à mettre en artifact
- `name:` permet de donner un nom à l'artifact. Par défaut elle sera nommée artifacts.zip
- `untracked:` permet d'ignorer les fichiers définis dans le fichier .gitignore
- `when:` , permet de définir quand l'artifact doit être créé. Trois choix possibles on_success, on_failure, always. La valeur on_success est la valeur par défaut.
- `expire_in:` permet de définir un temps d'expiration

```yaml
job:
  script: make build
  artifacts:
    paths:
      - dist
    name: artifact:build
    when: on_success
    expire_in: 1 weeks
```

LES GITLAB PAGES ET LA CI

Permet de publier des sites statiques directement à partir d'un repository Gitlab

- il suffit de publier du contenu html dans le répertoire `public` sous forme d'artefact
- il faut ajouter un `job` spécifique intitulé `pages`
- et il faut configurer les pages dans l'onglet `Pages` du menu `Parameters`

```
image: monachus/hugo

pages :
  script :
    - hugo
  artifacts :
    paths :
      - public
  only:
    - master
```

CONTAINER REGISTRY

- La container registry vous permet de gérer des images Docker ou podman
- `Deploy > Container Registry`
- Il est nécessaire de s'authentifier et de posséder les droits adéquats (personnal access token, projet deploy token, job token)
- Puis on peut donc builder, stocker et réutiliser une image

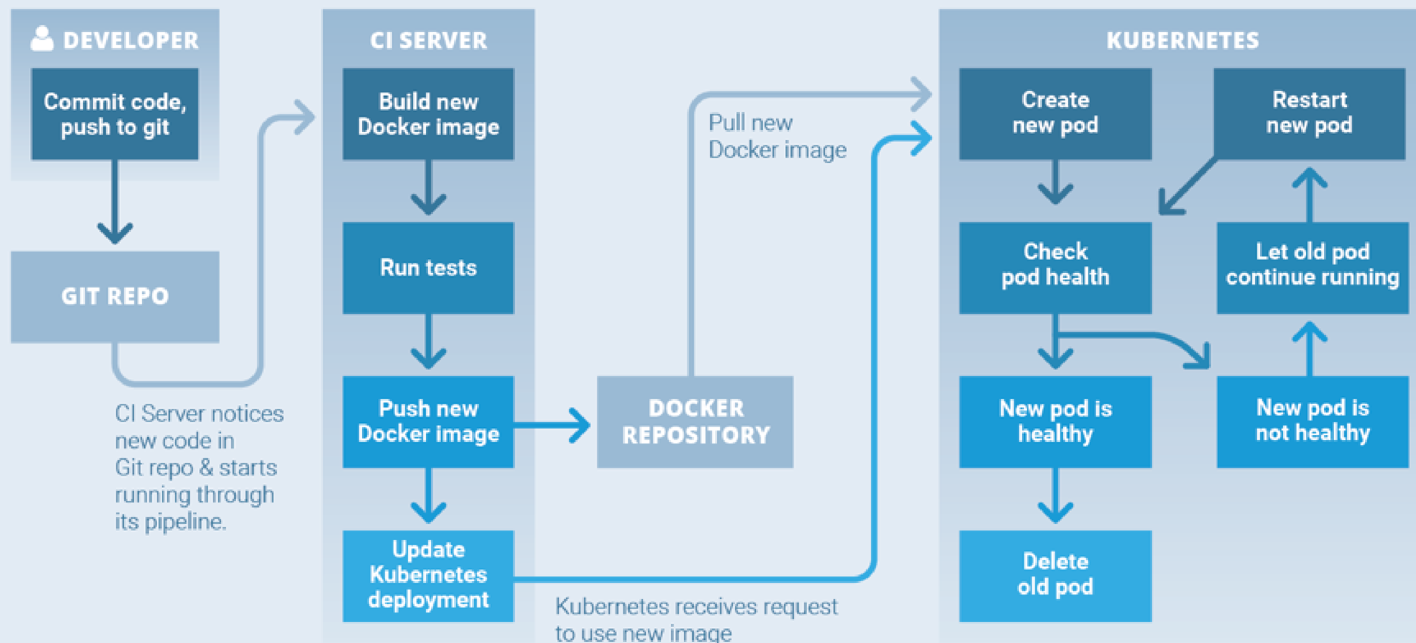
PACKAGE REGISTRY

- Le package registry vous permet de gérer des packages Maven, NuGet, npm, Conan, Helm, et PyPI ou des dépendances avec Composer
- `Deploy > Package Registry`
- Il est nécessaire de s'authentifier et de posséder les droits adéquats (personnal access token, projet deploy token, job token)
- Puis on peut donc builder, stocker et réutiliser un package

UN PAS PLUS LOIN - DÉPLOIEMENT D'UNE APPLI WEB

```
stages:
  - php
  - nginx
  - test

variables:
  IMAGE_PHP: $CI_REGISTRY_IMAGE/php-fpm:$CI_COMMIT_REF_SLUG-$CI_COMMIT_SHORT_SHA
  IMAGE_NGINX: $CI_REGISTRY_IMAGE/nginx:$CI_COMMIT_REF_SLUG-$CI_COMMIT_SHORT_SHA

default:
  services:
    - docker:dind
  image: docker:latest

php-image-build:
  stage: php
  before_script:
    - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" $CI_REGISTRY
  script:
    - docker build --pull -t "$IMAGE_PHP" -f ./docker/php-fpm/Dockerfile .
    - docker cp -a $(docker create "$IMAGE_PHP"):/var/www/html/. php-build
    - docker push "$IMAGE_PHP"
  artifacts:
    name: "$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
    paths:
      - php-build

nginx-image-build:
```

LE DÉPLOIEMENT CONTINU : UN PAS PLUS LOIN



DOCUMENTATION DE LA CI/CD GITLAB

- https://docs.gitlab.com/ee/ci/

LES MOTS CLÉS DE GITLAB CI

- https://docs.gitlab.com/ee/ci/yaml/

DES EXEMPLES DE MANIFESTS

- https://docs.gitlab.com/ee/ci/examples/

LA LISTE DES VARIABLES PRÉDÉFINIES

- https://docs.gitlab.com/ee/ci/variables/predefined_variables.html