Workshop CI / CD - GitlabCI - Jenkins Pipelines - Part 1

CI/CD C'EST QUOI ?

CI : CONTINUOUS INTEGRATION

> *"L'intégration continue est un ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée. […]Le principal but de cette pratique est de détecter les problèmes d'intégration au plus tôt lors du développement. De plus, elle permet d'automatiser l'exécution des suites de tests et de voir l'évolution du développement du logiciel."*

CD : CONTINUOUS DELIVERY

> *"La livraison continue est une approche d'ingénierie logicielle dans laquelle les équipes produisent des logiciels dans des cycles courts, ce qui permet de le mettre à disposition à n'importe quel moment. Le but est de construire, tester et diffuser un logiciel plus rapidement. L'approche aide à réduire le coût, le temps et les risques associés à la livraison de changement en adoptant une approche plus incrémentale des modifications en production. Un processus simple et répétable de déploiement est un élément clé."*

GITLAB EN QUELQUES MOTS

Alors Gitlab c'est :

- Gitlab inc : la compagnie qui gère les développements des produits GitLab
- Gitlab : c'est une version que vous pouvez installer sur votre machine, serveur ou dans le cloud facilement avec le **Market place d'AWS**
- GitLab.com : c'est une version web comme GitHub ou BitBucket.

GitLab et GitLab.com sont des gestionnaires de repositories git basés sur le web avec des fonctionnalités comme :

- un wiki,
- un suivi d'issue,
- un registry docker,
- un suivi de code,
- une review de code
- une CI/CD,
- ...

GitLab est beaucoup plus fourni en fonctionnalités que GitHub dans sa version gratuite. Il est aussi possible d'avoir des dépôts privés sans avoir d'abonnement.

GitLab CI/CD va vous permettre d'automatiser les `builds`, les `tests`, les `déploiements`, etc de vos applications. L'ensemble de vos tâches peut-être divisé en étapes et l'ensemble des vos tâches et étapes constituent une pipeline.

Chaque tâche est exécutée grâce à des `runners`, qui fonctionnent grâce à un projet open source nommé **GitLab Runner** écrit en **GO**.

Vous pouvez avoir vos propres `runners` directement sur votre machine ou serveur. Pour plus d'information je vous laisse lire la documentation officielle :

- **Page du projet GitLab Runner**
- **Configuration de GitLab Runner**
- **Configuration avancée de GitLab Runner**

GitLab propose aussi des runners publics, qui vous épargnent une installation, mais attention, il y a des quotas suivant le type de compte dont vous diposez. En compte gratuit, vous avez le droit à 2000 minutes de temps de pipeline par mois. Les runners publics de gitlab.com sont exécutés sur AWS.

PRÉSENTATION DE GITLAB CI/CD

LE MANIFESTE

Pour que la CI/CD sur GitLab fonctionne il vous faut un manifeste `.gitlab-ci.yml` à la racine de votre projet.
Dans ce manifeste vous allez pouvoir définir des `stages`, des `jobs`, des `variables`, des `anchors`, etc.

Vous pouvez lui donner un autre nom mais il faudra changer le nom du manifeste dans les paramètres de l'interface web : `Settings > CI/CD > General pipelines > Custom CI config path`

LES JOBS

Dans le manifeste de GitLab CI/CD vous pouvez définir un nombre illimité de `jobs`, avec des contraintes indiquant quand ils doivent être exécutés ou non.

Voici comment déclarer un `job` le plus simplement possible :

```
job:
  script: echo 'my first job'
```

Et si vous voulez déclarer plusieurs `jobs` :

```
job:1:
  script: echo 'my first job'

job:2:
  script: echo 'my second job'
```

les noms des `jobs` doivent être uniques et ne doivent pas faire parti des mots réservés :

- `image`
- `services`
- `stages`
- `types`
- `before_script`
- `after_script`
- `variables`
- `cache`

Dans la définition d'un `job` seule la déclaration `script` est obligatoire.

SCRIPT

La déclaration `script` est donc la seule obligatoire dans un `job`. Cette déclaration est le coeur du `job` car c'est ici que vous indiquerez les actions à effectuer.

Il peut appeler un ou plusieurs script(s) de votre projet, voire exécuter une ou plusieurs ligne(s) de commande.

```
job:script:
  script: ./bin/script/my-script.sh ## Appel d'un script de votre projet
```

```
job:scripts:
  script: ## Appel de deux scripts de votre projet
    - ./bin/script/my-script-1.sh
    - ./bin/script/my-script-2.sh

job:command:
  script: printenv # Exécution d'une commande

job:commands:
  script: # Exécution de deux commandes
    - printenv
    - echo $USER
```

BEFORE_SCRIPT ET AFTER_SCRIPT

Ces déclarations permettront d'exécuter des actions avant et après votre script principal. Ceci peut être intéressant pour bien diviser les actions à faire lors des `jobs`, ou bien appeler ou exécuter une action avant et après chaque `job`

```
before_script: # Exécution d'une commande avant chaque `job`
  - echo 'start jobs'

after_script: # Exécution d'une commande après chaque `job`
  - echo 'end jobs'

job:no_overwrite: # Ici le job exécutera les action du `before_script` et `after_script` par défaut
  script:
    - echo 'script'
```

```yaml
job:overwrite:before_script:
  before_script:
  - echo 'overwrite' # N'exécutera pas l'action définie dans le `before_script` par défaut
  script:
    - echo 'script'

job:overwrite:after_script:
  script:
  - echo 'script'
  after_script:
    - echo 'overwrite' # N'exécutera pas l'action définie dans le `after_script` par défaut

job:overwrite:all:
  before_script:
  - echo 'overwrite' # N'exécutera pas l'action définie dans le`before_script` par défaut
  script:
  - echo 'script'
  after_script:
    - echo 'overwrite' # N'exécutera pas l'action définie dans le `after_script` par défaut
```

IMAGE

Cette déclaration est simplement l'image docker qui sera utilisée lors d'un job ou lors de tous les jobs

```yaml
image: alpine # Image utilisée par tous les `jobs`, ce sera l'image par défaut

job:node: # Job utilisant l'image node
  image: node
  script: yarn install
```

```yaml
job:alpine: # Job utilisant l'image par défaut
  script: echo $USER
```

STAGES

Cette déclaration permet de grouper des `jobs` en étapes. Par exemple on peut faire une étape de `build`, de `codestyling`, de `test`, de `code coverage`, de `deployment`, **...**

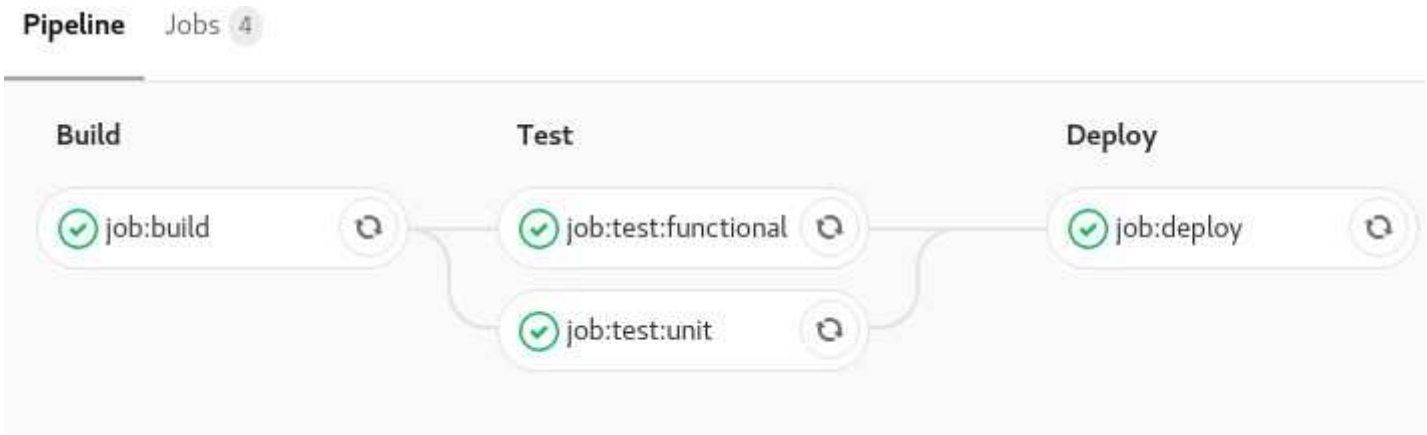
```yaml
stages: # Ici on déclare toutes nos étapes
  - build
  - test
  - deploy

job:build:
  stage: build # On déclare que ce `job` fait partie de l'étape build
  script: make build

job:test:unit:
  stage: test # On déclare que ce `job` fait partie de l'étape test
  script: make test-unit

job:test:functional:
  stage: test # On déclare que ce `job` fait partie de l'étape test
  script: make test-functional

job:deploy:
  stage: deploy # On déclare que ce `job` fait partie de l'étape deploy
  script: make deploy
```



ONLY ET EXCEPT

Ces deux directives permettent de mettre en place des contraintes sur l'exécution d'une tâche. Vous pouvez dire qu'une tâche s'exécutera uniquement sur l'événement d'un push sur master ou s'exécutera sur chaque push d'une branche saufmaster.

Voici les possibilités :

- branches déclenche le `job` quand un un push est effectué sur la branche spécifiée.
- tags déclenche le `job` quand un tag est créé.
- api déclenche le `job` quand une deuxième pipeline le demande grâce à API pipeline.
- external déclenche le `job` grâce à un service de CI/CD autre que GitLab.
- pipelines déclenche le `job` grâce à une autre pipeline, utile pour les multiprojets grâce à l'API et le token `CI_JOB_TOKEN`.

- pushes déclenche le `job` quand un `push` est effectué par un utilisateur.
- schedules déclenche le `job` par rapport à une planification à paramétrer dans l'interface web.
- triggers déclenche le `job` par rapport à un jeton de déclenchement.
- web déclenche le `job` par rapport au bouton `Run pipeline` dans l'interface utilisateur.

Je vais vous montrer trois exemples d'utilisation :

ONLY ET EXCEPT SIMPLE

Dans son utilisation la plus simple, le only et le except se déclarent comme ceci :

```
job:only:master:
  script: make deploy
  only:
    - master # Le job sera effectué uniquement lors d'un événement sur la branche master

job:except:master:
  script: make test
  except:master:
    - master # Le job sera effectué sur toutes les branches lors d'un événement sauf sur la branche master
```

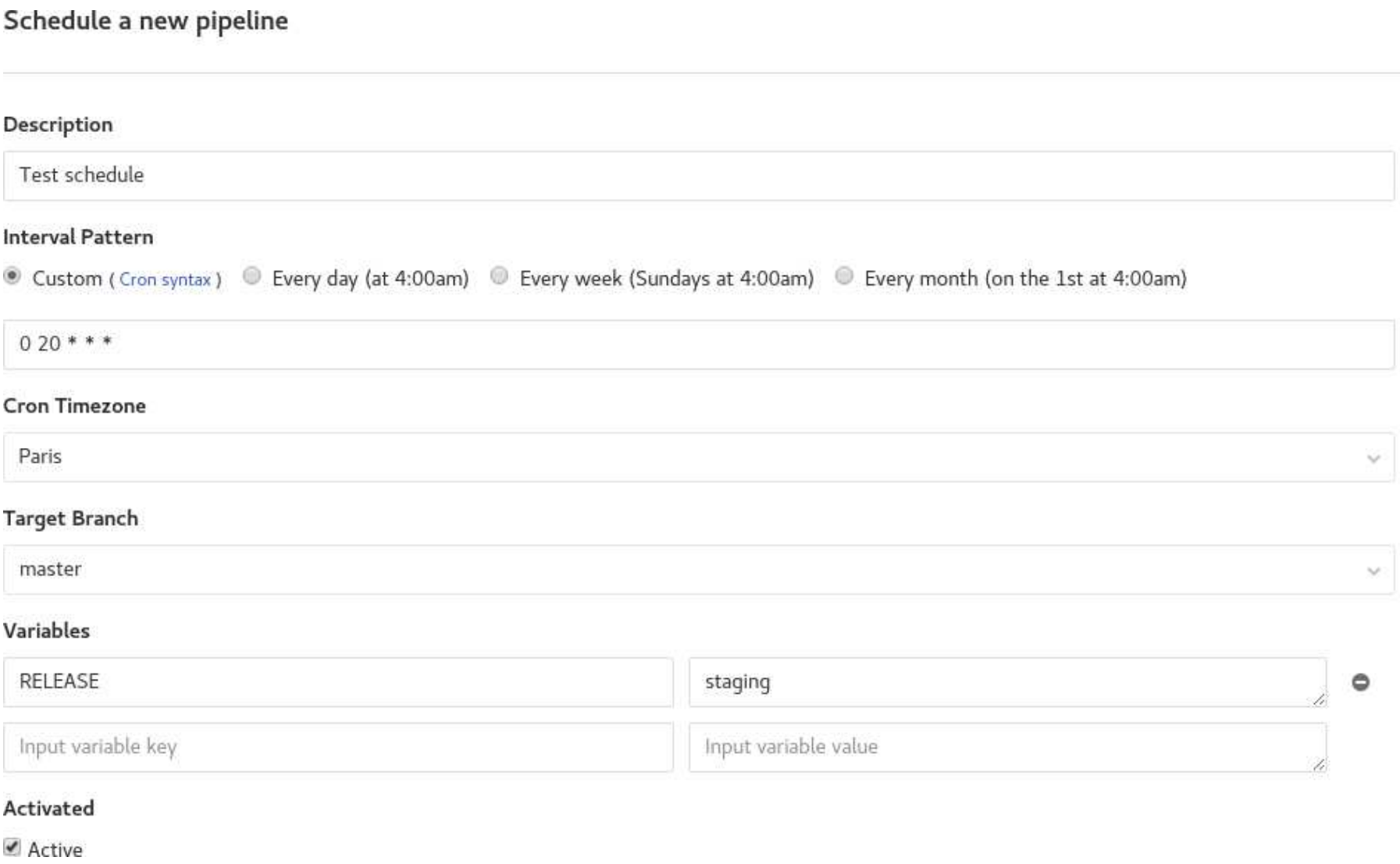
ONLY ET EXCEPT COMPLEX

Dans son utilisation la plus complexe, le only et le except s'utilisent comme ceci :

```yaml
job:only:master:
  script: make deploy
  only:
    refs:
    - master # Ne se fera uniquement sur master
    kubernetes: active # Kubernetes sera disponible
    variables:
      - $RELEASE == "staging" # On teste si $RELEASE vaut "staging"
      - $STAGING # On teste si $STAGING est défini
```

ONLY AVEC SCHEDULES

Pour l'utilisation de `schedules` il faut dans un premier temps définir des règles dans l'interface web. On peut les configurer dans l'interface web de Gitlab : `CI/CD -> Schedules` et remplir le formulaire.



Si vous souhaitez, vous pouvez définir un intervalle de temps personnalisé. C'est ce que j'ai fait dans mon exemple. La définition se fait comme un **cron**

WHEN

Comme pour les directives `only` et `except`, la directive `when` est une contrainte sur l'exécution de la tâche.
Il y a quatre modes possibles :

- `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

```yaml
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
```

ALLOW_FAILURE

Cette directive permet d'accepter qu'un job échoue sans faire échouer la pipeline.

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

...

stage: clean
  script:
    - make clean
    when: always
    allow_failure: true # Ne fera pas échouer la pipeline
...
```

TAGS

Comme je vous l'ai dit en début d'article, avec GitLab Runner vous pouvez héberger vos propres runners sur un serveur ce qui peut être utile dans le cas de configuration spécifique.

Chaque runner que vous définissez sur votre serveur à un nom, si vous mettez le nom du runner en `tags`, alors ce runner sera exécuté.

```
job:tag:
  script: yarn install
  tags:
    - shell # Le runner ayant le nom `shell` sera lancé
```

SERVICES

Cette déclaration permet d'ajouter des services (container docker) de base pour vous aider dans vos `jobs`. Par exemple si vous voulez utiliser une base de données pour tester votre application c'est dans `services` que vous le demanderez.

```
test:functional:
  image: registry.gitlab.com/username/project/php:test
  services:
    - postgres # On appel le service `postgres` comme base de données
 before_script:
   - composer install -n
```

```
  script:
    - codecept run functional
```

ENVIRONMENT

Cette déclaration permet de définir un environnement spécifique au déploiement. Vous pouvez créer un environnement dans l'interface web de GitLab ou tout simplement laisser GitLab CI/CD le créer automatiquement.

Il est possible de spécifier :

- un `name`,
- une `url` ,
- une condition `on_stop` ,
- une `action` en réponse de la condition précédente.

```
...

deploy:demo:
  stage: deploy
  environment: demo # Déclaration simple de l'environnement
  script:
    - make deploy

deploy:production:
  environment: # Déclaration étendue de l'environnement
    name: production
```

```
    url: 'https://blog.eleven-labs/fr/gitlab-ci/' # Url de l'application
  script:
    - make deploy
```

En déclarant des `environments` vous pouvez, depuis l'interface web de GitLab, déployer / redéployer votre application ou directement accéder à votre site si vous avez déclaré une `url`. Ceci se fait dans `Operations > Environment`.

Le bouton `undo` permet de redéployer, le bouton `external link` permet d'aller sur l'application et le bouton `remove` permet de supprimer l'environnement.

`on_stop` et `action` seront utilisés pour ajouter une action à la fin du déploiement, si vous souhaitez arrêter votre application sur commande. Utile pour les environnements de démonstration.

```
...

deploy:demo:
  script: make deploy
  environment:
    name: demo
    on_stop: stop:demo

stop:demo: # Ce job pourra être visible et exécuté uniquement après le job `deploy:demo`
  script: make stop
  environment:
    name: demo
    action: stop
```

Voici le lien officiel de la documentation sur les **environments** si vous souhaitez aller plus loin.

VARIABLES

Cette déclaration permet de définir des variables pour tous les `jobs` ou pour un `job` précis. Ceci revient à déclarer des variables d'environnement.

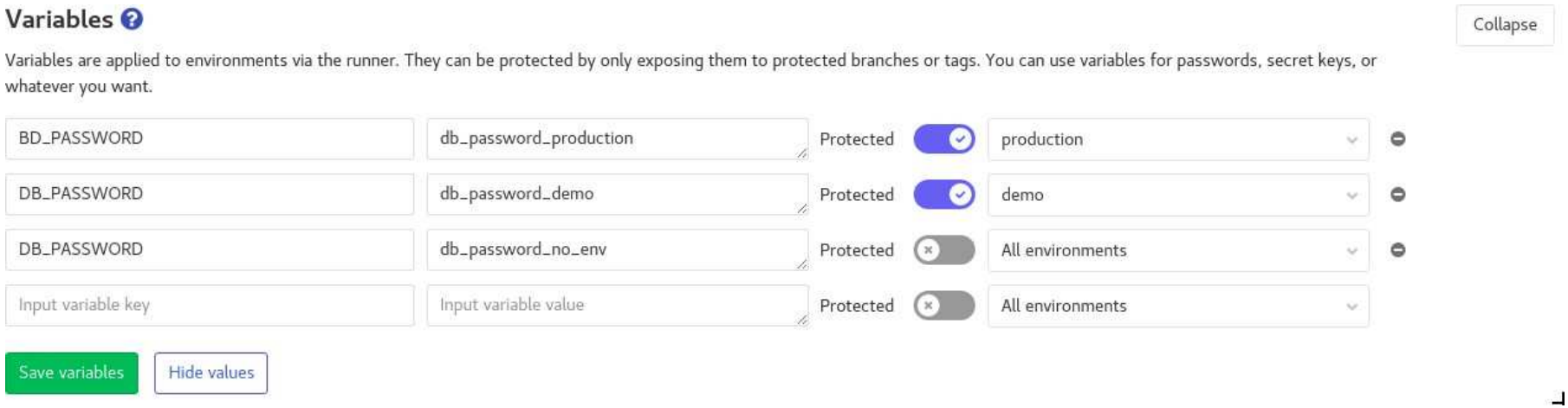
```yaml
...
variables: # Déclaration de variables pour tous les `job`
  SYMFONY_ENV: prod

build:
  script: echo ${SYMFONY_ENV} # Affichera "prod"

test:
  variables: # Déclaration et réécriture de variables globales pour ce `job`
    SYMFONY_ENV: dev
    DB_URL: '127.0.0.1'
  script: echo ${SYMFONY_ENV} ${DB_URL} # Affichera "dev 127.0.0.1"
```

Comme pour `environment` je vous laisse regarder la documentation officielle sur les **variables** si vous souhaitez aller plus loin.

Il est aussi possible de déclarer des variables depuis l'interface web de GitLab `Settings > CI/CD > Variables` et de leur spécifier un environnement.



CACHE

Cette directive permet de jouer avec du cache. Le cache est intéressant pour spécifier une liste de fichiers et de répertoires à mettre en cache tout le long de votre pipeline. Une fois la pipeline terminée le cache sera détruit.

Plusieurs sous-directives sont possibles :

- paths : obligatoire, elle permet de spécifier la liste de fichiers et/ou répertoires à mettre en cache
- key : facultative, elle permet de définir une clé pour la liste de fichiers et/ou de répertoires. Personnellement je n'en ai toujours pas vu l'utilité.
- untracked : facultative, elle permet de spécifier que les fichiers ne doivent pas être suivis par votre dépôt git en cas d'un `push` lors de votre pipeline.

- policy : facultative, elle permet de dire que le cache doit être récupéré ou sauvegardé lors d'un job (`push` ou `pull`).

```
stages:
  - build
  - deploy

job:build:
  stage: build
  image: node:8-alpine
  script: yarn install && yarn build
  cache:
    paths:
      - build # répertoire mis en cache
    policy: push # le cache sera juste sauvegardé, pas de récupération d'un cache existant

job:deploy:
  stage: deploy
  script: make deploy
  cache:
    paths:
      - build
    policy: pull # récupération du cache
```

ARTIFACTS

Les artefacts sont un peu comme du cache mais ils peuvent être récupérés depuis une autre pipeline. Comme pour le cache il faut définir une liste de fichiers ou/et répertoires qui seront sauvegardés par GitLab. Les fichiers sont sauvegardés uniquement si le `job` réussit.

Nous y retrouvons cinq sous-directives possibles :

- paths : obligatoire, elle permet de spécifier la liste des fichiers et/ou dossiers à mettre en `artifact`
- name: facultative, elle permet de donner un nom à l'`artifact`. Par défaut elle sera nommée `artifacts.zip`
- untracked : facultative, elle permet d'ignorer les fichiers définis dans le fichier `.gitignore`
- when : facultative, elle 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 : facultative, elle permet de définir un temps d'expiration

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

DEPENDENCIES

Cette déclaration fonctionne avec les `artifacts`, il rend un `job` dépendant d'un `artifact`. Si l'`artifact` a expiré ou a été supprimé / n'existe pas, alors la pipeline échouera.

```yaml
build:artifact:
  stage: build
  script: echo hello > artifact.txt
  artifacts: # On ajoute un `artifact`
    paths:
      - artifact.txt

deploy:ko:
  stage: deploy
  script: cat artifact.txt
  dependencies: # On lie le job avec 'build:artifact:fail' qui n'existe pas donc la pipeline échouera
    - build:artifact:fail

deploy:ok:
  stage: deploy
  script: cat artifact.txt
  dependencies: # On lie le job avec 'build:artifact' qui existe donc la pipeline n'échouera pas
    - build:artifact
```

COVERAGE

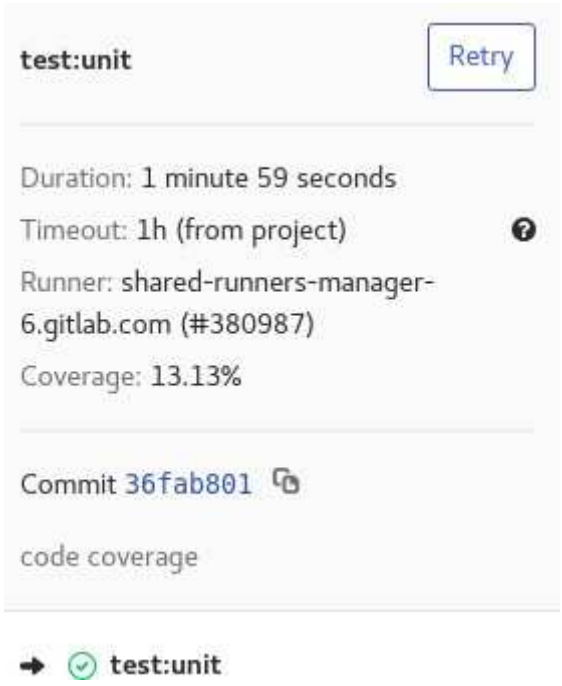
Cette déclaration permet de spécifier une expression régulière pour récupérer le code coverage pour un `job`.

```yaml
...

test:unit:
```

```
script: echo 'Code coverage 13.13'
coverage: '/Code coverage \d+\.\d+/'
```

Le code coverage sera visible dans les informations du `job` dans l'interface web de GitLab :



> *Si vous le souhaitez voici un autre article de notre blog écrit par l'astronaute* ***Pouzor*** *sur le code coverage :* ***Ajouter le code coverage sur les MR avec avec GitLab-CI***

RETRY

Cette déclaration permet de ré-exécuter le `job` en cas d'échec. Il faut indiquer le nombre de fois où vous voulez ré-exécuter le `job`

```
job:retry:
  script: echo 'retry'
  retry: 5
```

INCLUDE

Pour cette fonctionnalité il vous faudra un compte premium. Cette fonctionnalité permet d'inclure des "templates". les "templates" peuvent être en local dans votre projet ou à distance.

Les fichiers sont toujours évalués en premier et fusionnés récursivement. Vous pouvez surcharger ou remplacer des déclarations des"templates".

- template en local

```
# template-ci/.lint-template.yml

job:lint:
  stage: lint
  script:
    - yarn lint
```

- template à distance

```
# https://gitlab.com/awesome-project/raw/master/template-ci/.test-template.yml

job:test:
  stage: test
  script:
    - yarn test
```

- manifeste principal

```
# .gitlab-ci.yml

include:
  - '/template-ci/.lint-template.yml'
  - 'https://gitlab.com/awesome-project/raw/master/template-ci/.test-template.yml'

stages:
  - lint
  - test

image: node:9-alpine

job:lint:
  before_script:
    - yarn install

job:test:
  script:
    - yarn install
    - yarn unit
```

Voici ce que gitlab CI/CD va interpréter :

```
stages:
  - lint
  - test

image: node:9-alpine

job:lint:
  stage: lint
  before_script: # on surcharge `job:lint` avec `before_script`
  - yarn install
  script:
    - yarn lint

job:test:
  stage: test
  script: # on remplace la déclaration `script` du "template" https://gitlab.com/awesome-project/raw/maste
    - yarn install
    - yarn unit
```

Ceci peut être intéressant dans le cas où votre manifeste est gros, et donc plus difficile à maintenir.

ANCHORS

Cette fonctionnalité permet de faire des templates réutilisables plusieurs fois.

```
.test_template: &test_template
  stage: test
  image: registry.gitlab.com/username/project/php:test
  before_script:
  - composer install -n
  when: on_success

.db_template:
  services:
    - postgres
    - mongo

test:unit:
  <<: *test_template
  script:
    - bin/phpunit --coverage-text --colors=never tests/

test:functional:
  <<: *test_template
  services: *db_template
  script:
    - codecept run functional
```

Voici ce que gitlab CI/CD va interpréter :

```
test:unit:
  stage: test
  image: registry.gitlab.com/username/project/php:test
  before_script:
    - composer install -n
  script:
```

```
    - bin/phpunit --coverage-text --colors=never tests/
  when: on_success

test:functional:
  stage: test
  image: registry.gitlab.com/username/project/php:test
  services:
    - postgres
  - mongo
  before_script:
  - composer install -n
  script:
  - codecept run functional
  when: on_success
```