Continuous Integration

(Oltre Lo) Sviluppo Software

Stefano Antonelli

Diego Michelotto

28/04/2021

Summary

- CI/CD in generale
- CI/CD in GitLab
  - Hands-on GitLab CI

CI/CD

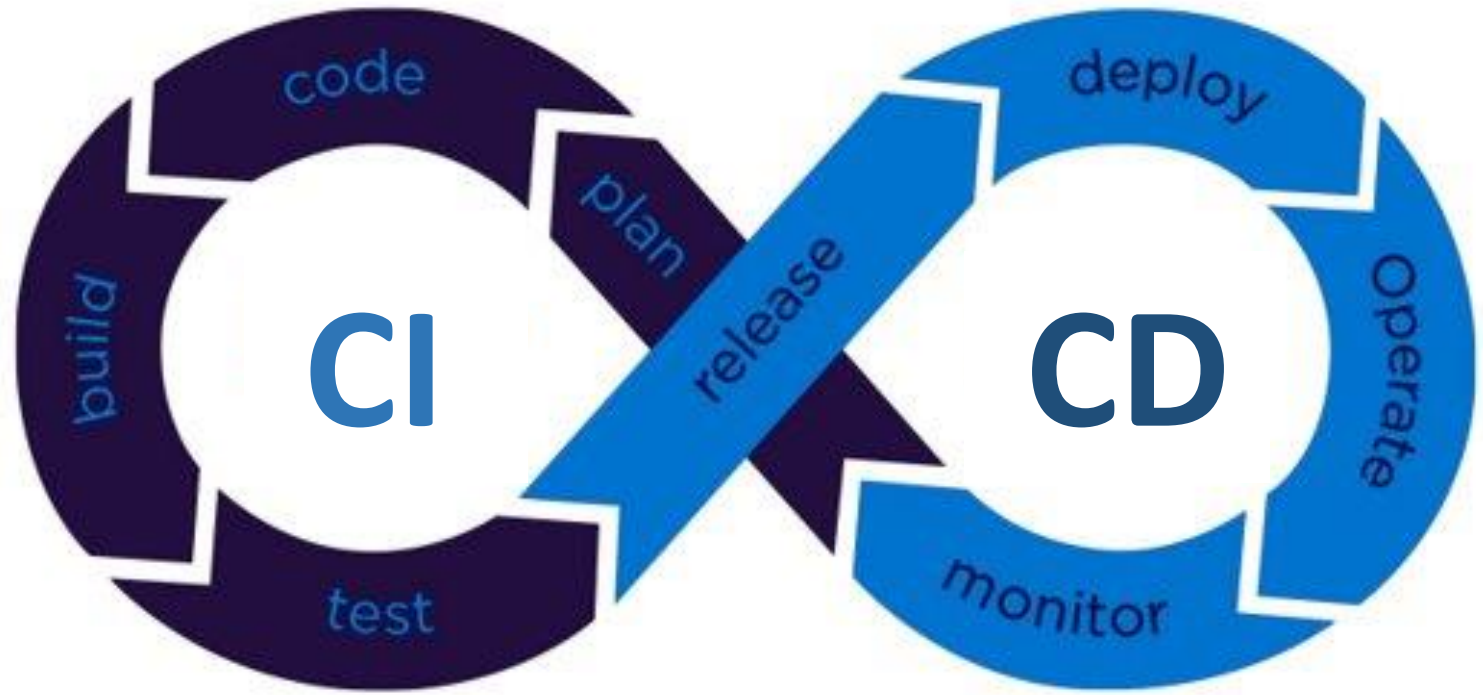
CI/CD

- La CI/CD (continuous integration/continuous delivery) sono i passaggi che devono essere eseguiti per fornire una nuova versione software.
- Le pipeline di CI/CD sono procedure pensate per ottimizzare l'erogazione di software attraverso l'approccio DevOps o Site Reliability Engineering (SRE).
- Il flusso CI/CD introduce il monitoraggio e l'automazione per ottimizzare il processo di sviluppo delle applicazioni
  - fasi di integrazione
  - test
  - distribuzione
  - deployment
- Il vantaggio principale della CI/CD sta nell'automazione delle procedure

CI/CD Pipeline

- Una pipeline CI/CD si suddivide in sottoinsiemi distinti di attività. Le fasi di pipeline tipiche includono:
  - **Build**: la fase di compilazione dell'applicazione.
  - **Test**: la fase in cui il codice viene testato. Qui l'automazione può far risparmiare tempo e fatica.
  - **Rilascio**: la fase in cui l'applicazione viene fornita al repository.
  - **Deployment**: in questa fase il software viene distribuito in produzione.
  - **Convalida e conformità**: i passaggi per convalidare una build dipendono dalle esigenze dell'organizzazione. Gli strumenti di scansione per la sicurezza del software confrontandolo con vulnerabilità note (CVE).

CI/CD Pipeline



CI/CD Pipeline

- Una pipeline CI/CD si suddivide in sottoinsiemi distinti di attività. Le fasi di pipeline tipiche includono:
  - **Build**: la fase di compilazione dell'applicazione.
  - **Test**: la fase in cui il codice viene testato. Qui l'automazione può far risparmiare tempo e fatica. I test possono essere di diverso tipo:
    - Unit test: test unità elementari del software
    - Integration test: test di interazione tra più elementari del software
    - Functional test: test che dato un input il Software (Backbox) dia l'output atteso
  - **Rilascio**: la fase in cui l'applicazione viene fornita al repository.
  - **Deployment**: in questa fase il software viene distribuito in staging.
  - **Test**: la fase in cui il software viene testato in esecuzione.
    - Deployment test: test di installazione del software, per esempio su piattaforme differenti
    - Load/Stress test: test di carico del software
  - **Convalida e conformità**:
    - Acceptance test: convalida dei requisiti
    - Security test: confronto con vulnerabilità note (CVE).
  - **Deployment**: il software viene distribuito in produzione
    - Installazione in preproduzione
    - Smoke test
    - Installazione in produzione

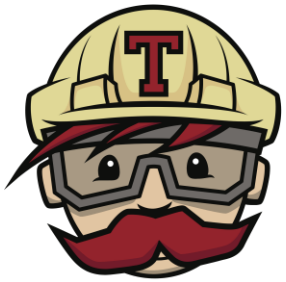
CI/CD Pipeline

- Una pipeline CI/CD si suddivide in sottoinsiemi distinti di attività. Le fasi di pipeline tipiche includono:
  - **Build**: la fase di compilazione dell'applicazione.
  - **Test**: la fase in cui il codice viene testato. Qui l'automazione può far risparmiare tempo e fatica. I test possono essere di diverso tipo:
    - Unit test: test unità elementari del software
    - Integration test: test di interazione tra più elmentari del software
    - Functional test: test che dato un input il Sofware (Backbox) dia l'output atteso
  - **Rilascio**: la fase in cui l'applicazione viene fornita al repository.
  - **Deployment**: in questa fase il software viene distribuito in staging
  - **Test**: la fase in cui il software viene testato in esecuzione
    - Deployment test: test di installazione del sofware, per esempio su piattaforme differenti
    - Load/Stress test: test di carico del sofware
  - **Convalida e conformità**:
    - Acceptance test: convalida dei requisiti
    - Security test: confronto con vulnerabilità note (CVE).
  - **Deployment**: il software viene distribuito in produzione
    - Installazione in preproduzione
    - Smoke test
    - Installazione in produzione

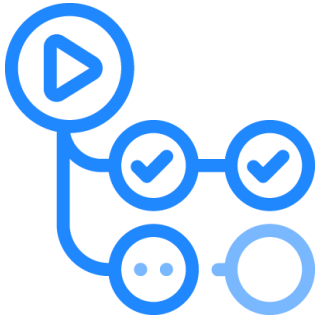
La situazione si può complicare ulteriormente

Tools

- **GitLab CI**
- **GitHub Actions**
- Jenkins
- Travis CI
- Bamboo/Bitbucket Pipelines (Atlassian)
- …



Travis CI



Jenkins

Bamboo

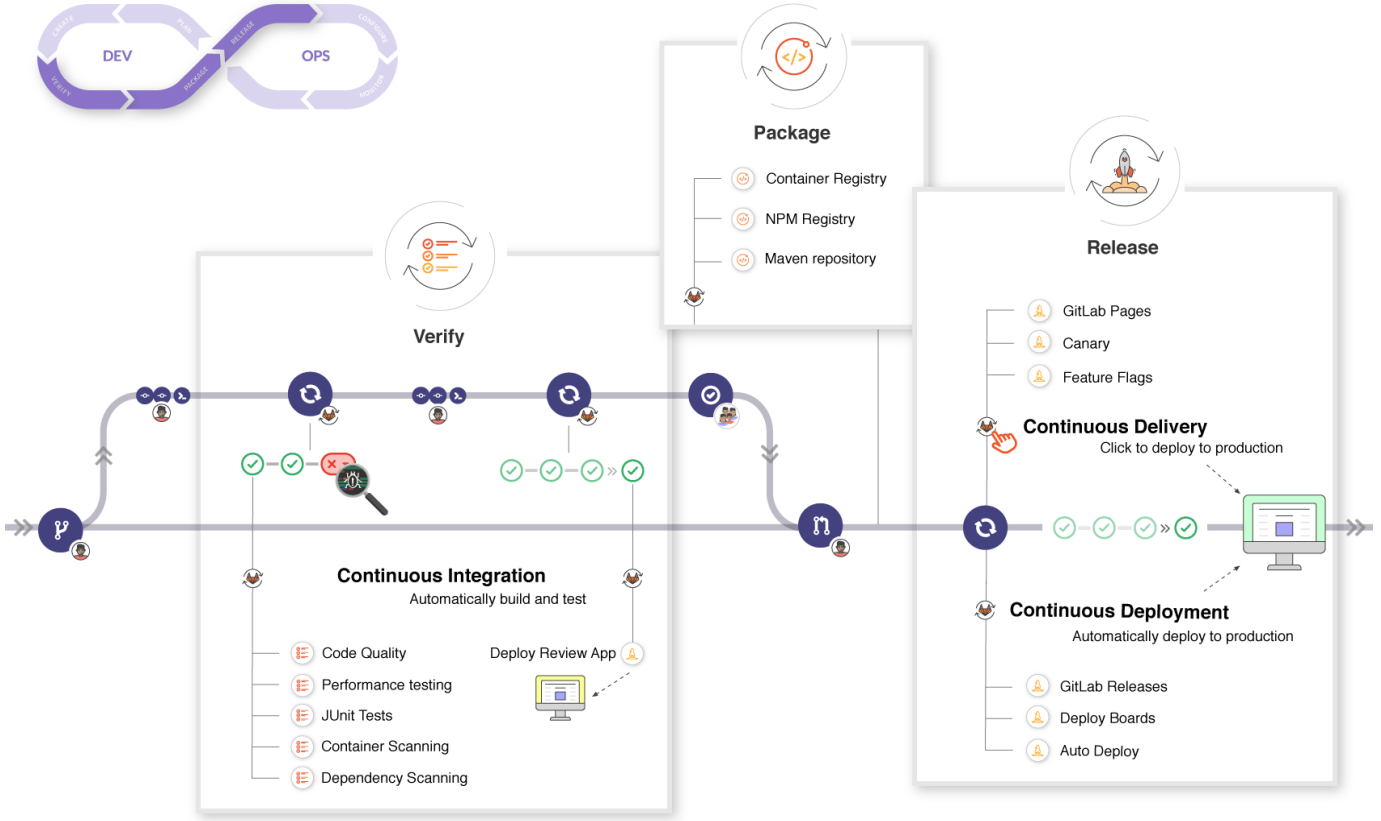
Bitbucket Pipelines

CI/CD in GitLab

CI/CD in GitLab

- La CI/CD è integrata in GitLab e consente lo sviluppo software secondo le metodologie
  - Continuous integration CI
    Uno sviluppatore ha il suo repository in GitLab ed ad ogni "push" sul repository, eventualmente anche per i branch di sviluppo, parte una serie di scripts che compilano e testano l'applicazione
  - Continuous delivery CD
    - Un passo ulteriore rispetto alla CI; l'applicazione viene rilasciata per il repository
  - Continuous deployment CD
    - Il rilascio dell'applicazione viene messo in produzione automaticamente senza intervento manuale

https://docs.gitlab.com/ee/ci/quick_start/index.html



CI/CD in GitLab

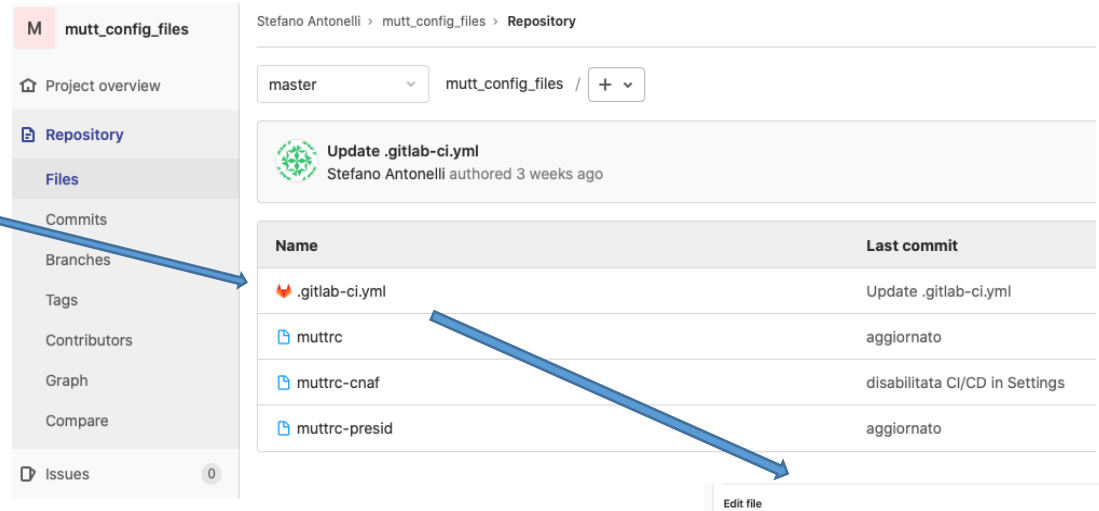
Cosa serve? Un repository ed un file.

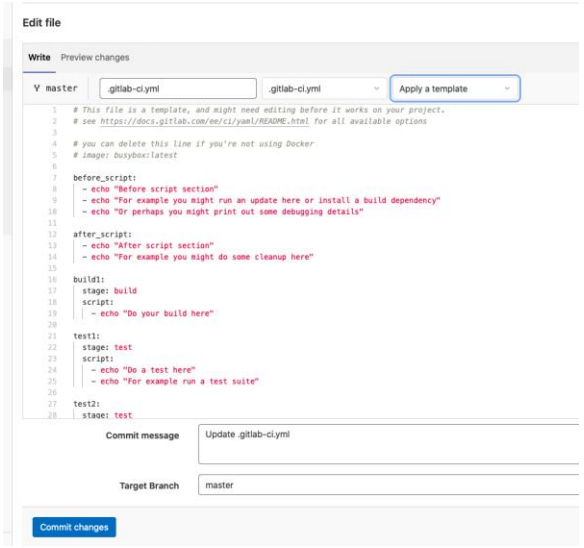
La CI/CD è configurata tramite un file .gitlab-ci.yml presente nella *root* del repository. Ad un *push* sul repository, il file esegue una *pipeline*, un set di istruzioni che eseguono dei *jobs* eseguiti su dei *runner*

Il file *.gitlab-ci.yml* è scritto in *YAML*

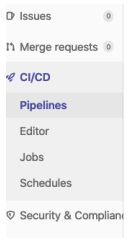
- https://en.wikipedia.org/wiki/YAML
- https://yaml.org/

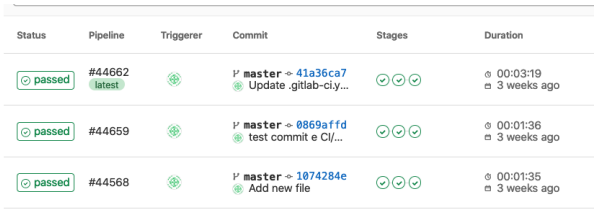
Se utilizzo CI/CD alternative (e.g. Jenkins, Drone) è consigliabile disabilitare la CI in GitLab per evitare conflitti





...e controllare la CI/CD dalla GUI di GitLab



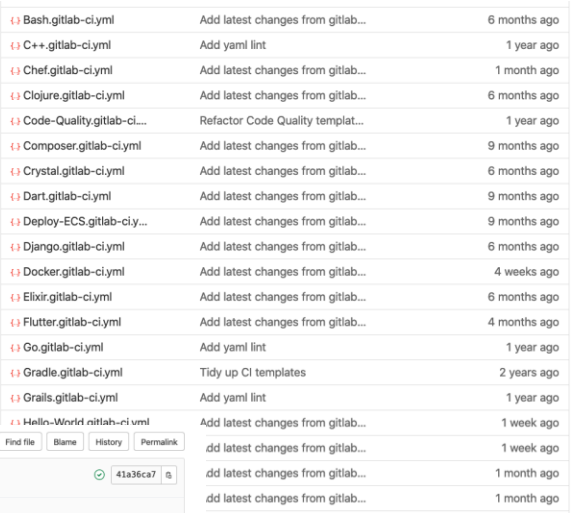


CI/CD in GitLab

File .gitlab-ci.yml

Esistono diversi template di .gitlab-ci.yml disponibili dalla GUI

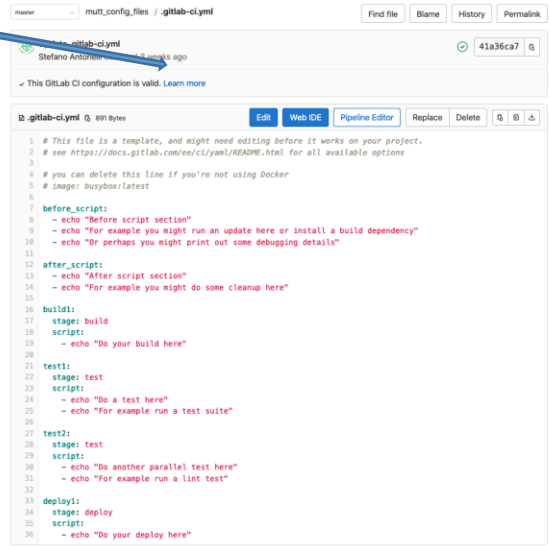
https://gitlab.com/gitlab-org/gitlab-foss/tree/master/lib/gitlab/ci/templates



Posso editare il file dall'editor di GitLab

URL per *keywords* per *pipeline* (e.g. uno stage è def. dalla *keyword stage*)

https://docs.gitlab.com/ee/ci/yaml/README.html#variables



Editare la *pipeline*

Visualizzare le *pipeline*

Verificare la sintassi

CI/CD in GitLab

Pipeline: è il componente di livello più alto della CI/CD. Una pipeline comprende

- *Jobs* cosa viene fatto
- *Stages* quando viene fatto

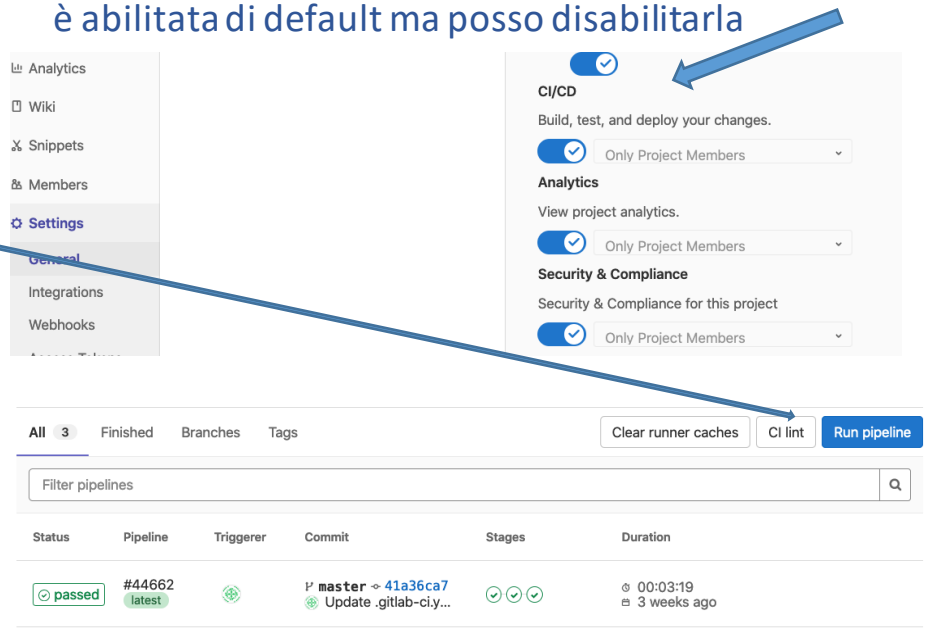
In generale parte in automatico in seguito ad un push ma può essere eseguita anche manualmente

URL che mostrano i tipi di pipeline configurabili

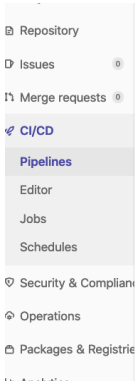
https://docs.gitlab.com/ee/ci/pipelines/index.html#types-of-pipelines

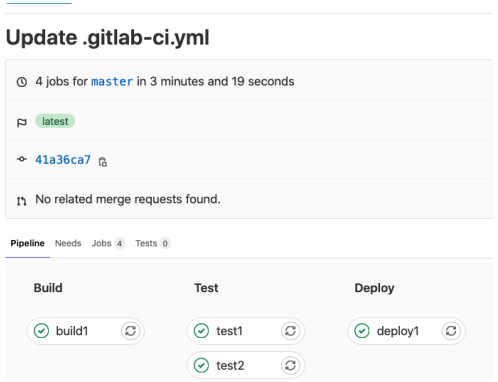
https://docs.gitlab.com/ee/ci/pipelines/pipeline_architectures.html

è abilitata di default ma posso disabilitarla



Dalla GUI posso controllare lo stato delle *pipeline*





CI/CD in GitLab

Pipeline: è il componente di livello più alto della CI/CD.

- Una pipeline semplice
- Una pipeline complessa
  - https://gitlab.com/gitlab-org/gitlab/blob/master/.gitlab-ci.yml

```
29
30 ### Test simple pipeline
31
32 simple:
33   script:
34     - echo "bellashell"
```

```
14 Skipping Git submodules setup
15 Executing "step_script" stage of the job script
16 Using docker image sha256:300e315adb2f96afe5f0b2
   b8c04efc1 ...
17 $ echo "bellashell"
18 bellashell
19 Cleaning up file based variables
20 Job succeeded
```

```
1  stages:
2    - sync
3    - prepare
4    - build-images
5    - fixtures
6    - test
7    - post-test
8    - review-prepare
9    - review
10   - dast
11   - qa
12   - post-qa
13   - pages
14   - notify
15
```

```
53 variables:
54   RAILS_ENV: "test"
55   NODE_ENV: "test"
56   # we override the max_old_space_size to prevent OOM errors
57   NODE_OPTIONS: --max_old_space_size=3584
58   SIMPLECOV: "true"
59   GIT_DEPTH: "20"
60   GIT_SUBMODULE_STRATEGY: "none"
61   GET_SOURCES_ATTEMPTS: "3"
62   KNAPSACK_RSPEC_SUITE_REPORT_PATH: knapsack/report-master.json
63   FLAKY_RSPEC_SUITE_REPORT_PATH: rspec_flaky/report-suite.json
64   RSPEC_TESTS_MAPPING_PATH: crystalball/mapping.json
65   RSPEC_PACKED_TESTS_MAPPING_PATH: crystalball/packed-mapping.json
66   BUILD_ASSETS_IMAGE: "false"
67   ES_JAVA_OPTS: "-Xms256m -Xmx256m"
68   ELASTIC_URL: "http://elastic:changeme@elasticsearch:9200"
69   DOCKER_VERSION: "20.10.1"
70   CACHE_CLASSES: "true"
71
```

```
include:
  - local: .gitlab/ci/build-images.gitlab-ci.yml
  - local: .gitlab/ci/cache-repo.gitlab-ci.yml
  - local: .gitlab/ci/cng.gitlab-ci.yml
  - local: .gitlab/ci/docs.gitlab-ci.yml
  - local: .gitlab/ci/frontend.gitlab-ci.yml
  - local: .gitlab/ci/global.gitlab-ci.yml
  - local: .gitlab/ci/memory.gitlab-ci.yml
  - local: .gitlab/ci/pages.gitlab-ci.yml
  - local: .gitlab/ci/qa.gitlab-ci.yml
  - local: .gitlab/ci/reports.gitlab-ci.yml
  - local: .gitlab/ci/rails.gitlab-ci.yml
  - local: .gitlab/ci/vendored-gems.gitlab-ci.yml
  - local: .gitlab/ci/review.gitlab-ci.yml
  - local: .gitlab/ci/rules.gitlab-ci.yml
  - local: .gitlab/ci/setup.gitlab-ci.yml
  - local: .gitlab/ci/dev-fixtures.gitlab-ci.yml
  - local: .gitlab/ci/test-metadata.gitlab-ci.yml
  - local: .gitlab/ci/yaml.gitlab-ci.yml
  - local: .gitlab/ci/releases.gitlab-ci.yml
  - local: .gitlab/ci/notify.gitlab-ci.yml
  - local: .gitlab/ci/dast.gitlab-ci.yml
  - local: .gitlab/ci/workhorse.gitlab-ci.yml
  - local: .gitlab/ci/graphql.gitlab-ci.yml
```

CI/CD in GitLab

Pipeline

Oltre le *keywords* da utilizzare per "costruire" la *pipeline*, ci sono delle variabili ambientali che vengono definite nell'esecuzione dei *jobs* e sono utili per

- controllare il comportamento dei *jobs* e della *pipeline*
- assumere un valore che posso riutilizzare nel *job*
- evitare valori *hard-coded* nel file .gitlab-ci.yml
- https://docs.gitlab.com/ee/ci/variables/predefined_variables.html
- posso anche definire delle variabili *custom*
- La *keyword stage:* definisce quale *job* viene eseguito durante quello *stage.* Qui viene descritta la *keyword stage* https://docs.gitlab.com/ee/ci/yaml/README.html#stage

```
test_variable:
  stage: test
  script:
    - echo $CI_JOB_STAGE
```

```
variables:
  TEST_VAR: "All jobs can use this variable's value"

job1:
  variables:
    TEST_VAR_JOB: "Only job1 can use this variable's value"
  script:
    - echo $TEST_VAR and $TEST_VAR_JOB
```

Esempio di:

- *pipeline* che genera un *artifact* (un prodotto del job)

L'*artifact* è un pdf che sarà rimosso dopo una settimana *expire_in* ed è nella directory *paths* dove *paths* è relativo al repository dove è eseguito il job

N.B. In caso di *artifact* inserire il più possibile la *keyword expire_in*: (seguita da xx hr, day, week etc.) per evitare che gli *artifact* occupino spazio e restino lì per un tempo indeterminato

```
pdf:
  script: xelatex mycv.tex
  artifacts:
    paths:
      - mycv.pdf
    expire_in: 1 week
```

CI/CD in GitLab

Keywords stages & stage

Stages

- Definisce gli *stage* che contengono gruppi di *job*
- L'ordine degli argomenti definisce l'ordine di esecuzione dei *job*
- E' definita globalmente
- Se non è definita, gli stage di default sono build, test, deploy

```
stages:
  - build
  - test
  - deploy
```

Stage

- Definisce quale *job* è eseguito in quello *stage*
- Se *stages* non è definita ci sono 5 stage di default (eseguiti in ordine) *.pre, build, test, deploy, .post*
- Ad un *job* senza *stage* viene assegnato lo stage *test*

```
stages:
  - build
  - test
  - deploy

job 0:
  stage: .pre
  script: make something useful before build stage

job 1:
  stage: build
  script: make build dependencies

job 2:
  stage: build
  script: make build artifacts

job 3:
  stage: test
  script: make test

job 4:
  stage: deploy
  script: make deploy
```

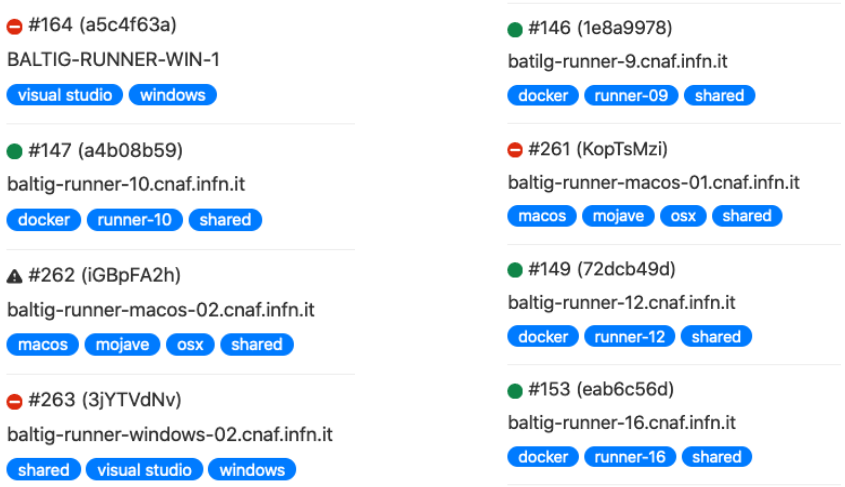
CI/CD in GitLab

Chi esegue il codice definito in *.gitlab-ci.yml*?

- Questo compito è svolto dai *runner*
- In GitLab INFN ci sono gli *shared runners*, 16 *runner* a disposizione degli utenti

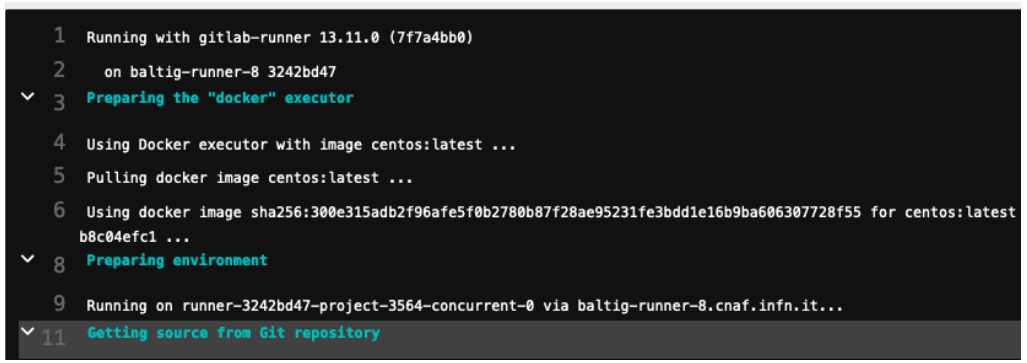
Quando utilizzo la CI controllo se ci sono *runner* disponibili

https://docs.gitlab.com/ee/ci/quick_start/index.html#ensure-you-have-runners-available



Il *job* viene eseguito su uno dei *runner* a disposizione

La *dashboard* segnala il tempo di esecuzione di ogni *job*



CI/CD in GitLab

La dashboard consente la gestione della CI (pipeline, job...)