GitHub
actions ci/cd

risorse.students.cs.unibo.it/lab/2024

Alice Benatti, Samuele Musiani

>ADM staff

CI/CD continuous integration continuous delivery

per automatizzare la build e il testing del codice

La Continuous Integration (CI) è la pratica di **automatizzare la compilazione** e il test del codice ogni volta che viene apportata una modifica.

Ogni nuovo commit del codice attiva un **processo di compilazione** e test automatizzato, spesso chiamato "**pipeline**", per segnalare eventuali difetti rilevati durante la compilazione o il test il più rapidamente possibile.

Perché usarla?

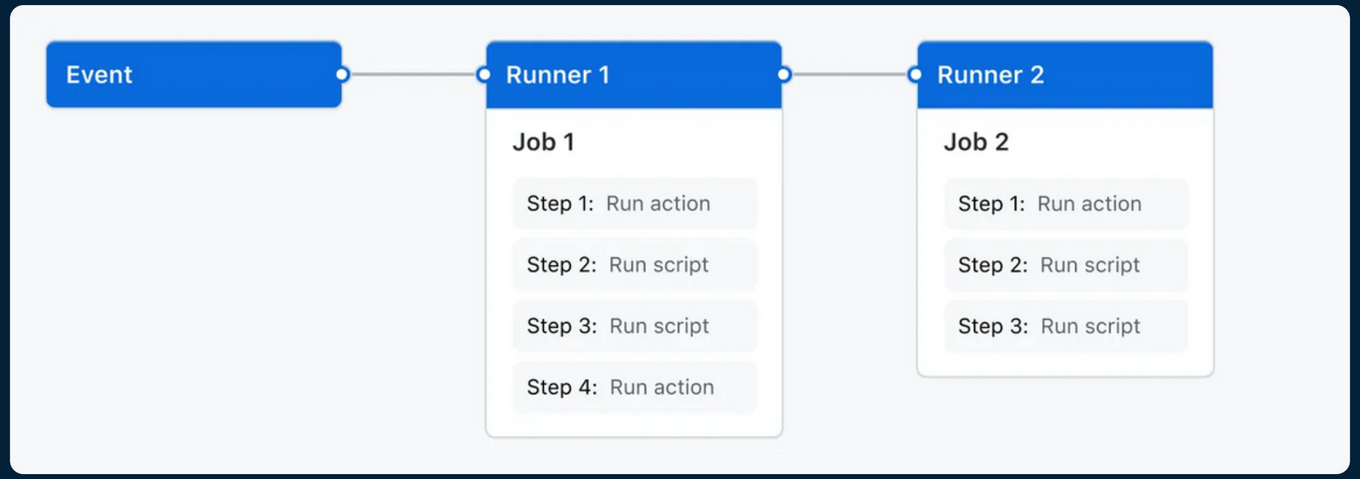
- Automaticamente vengono fatti test sul codice
- Test centralizzati su tutta la repository
- Semplifica il debugging
- Migliora la produttività

GitHub Actions

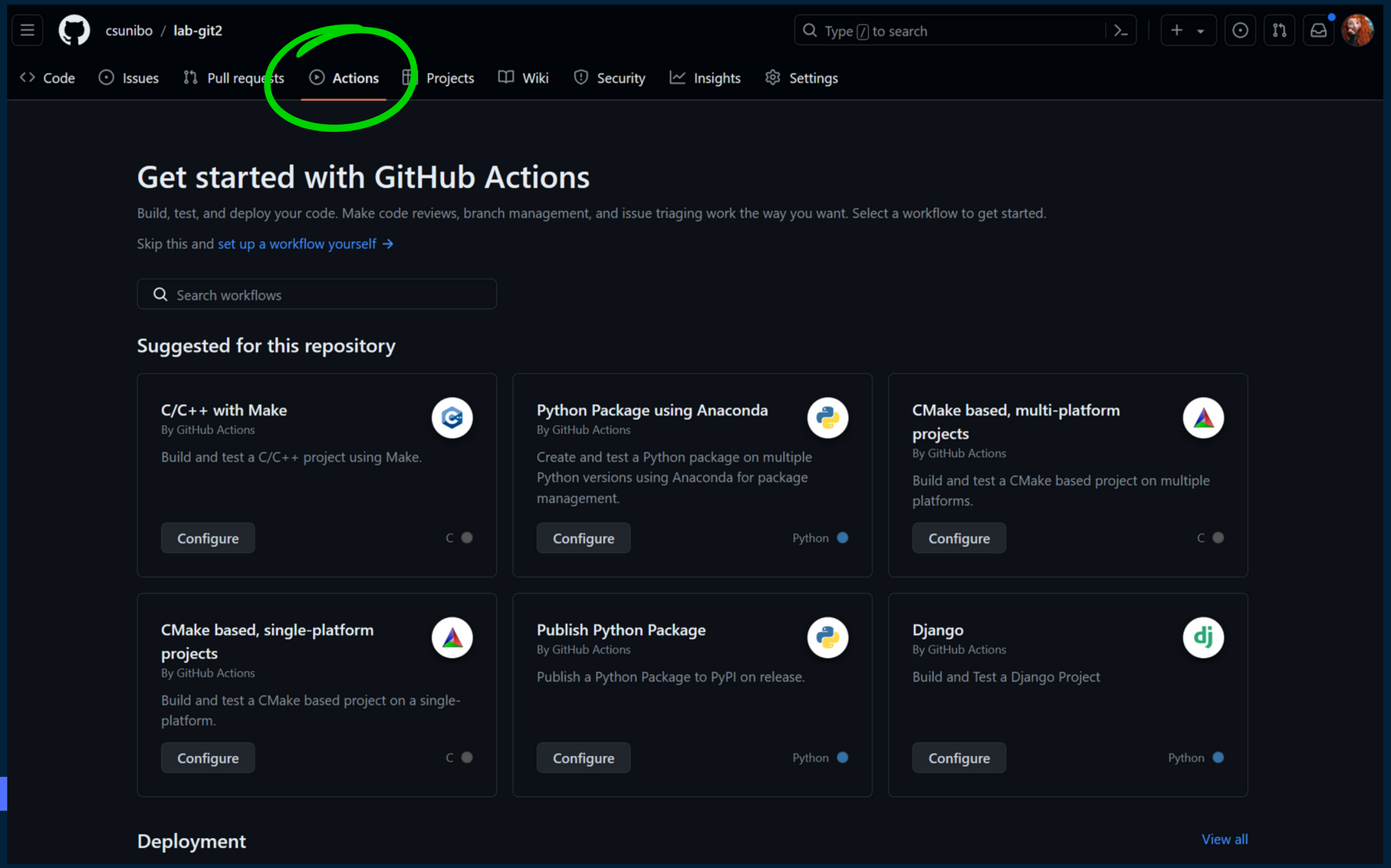
GitHub ci mette a disposizione le *actions* in modo da **automatizzare i processi** (CI)

Anche altri sistemi di hosting di git permettono di avere dei workflow simili alle GitHub Actions.

Contengono uno o più processi che possono essere eseguiti in ordine sequenziale o in parallelo. Ogni processo verrà eseguito all'interno del proprio runner di macchine virtuali o all'interno di un contenitore e ha uno o più passaggi che eseguono uno script definito dall'utente o un'azione.



GitHub Actions

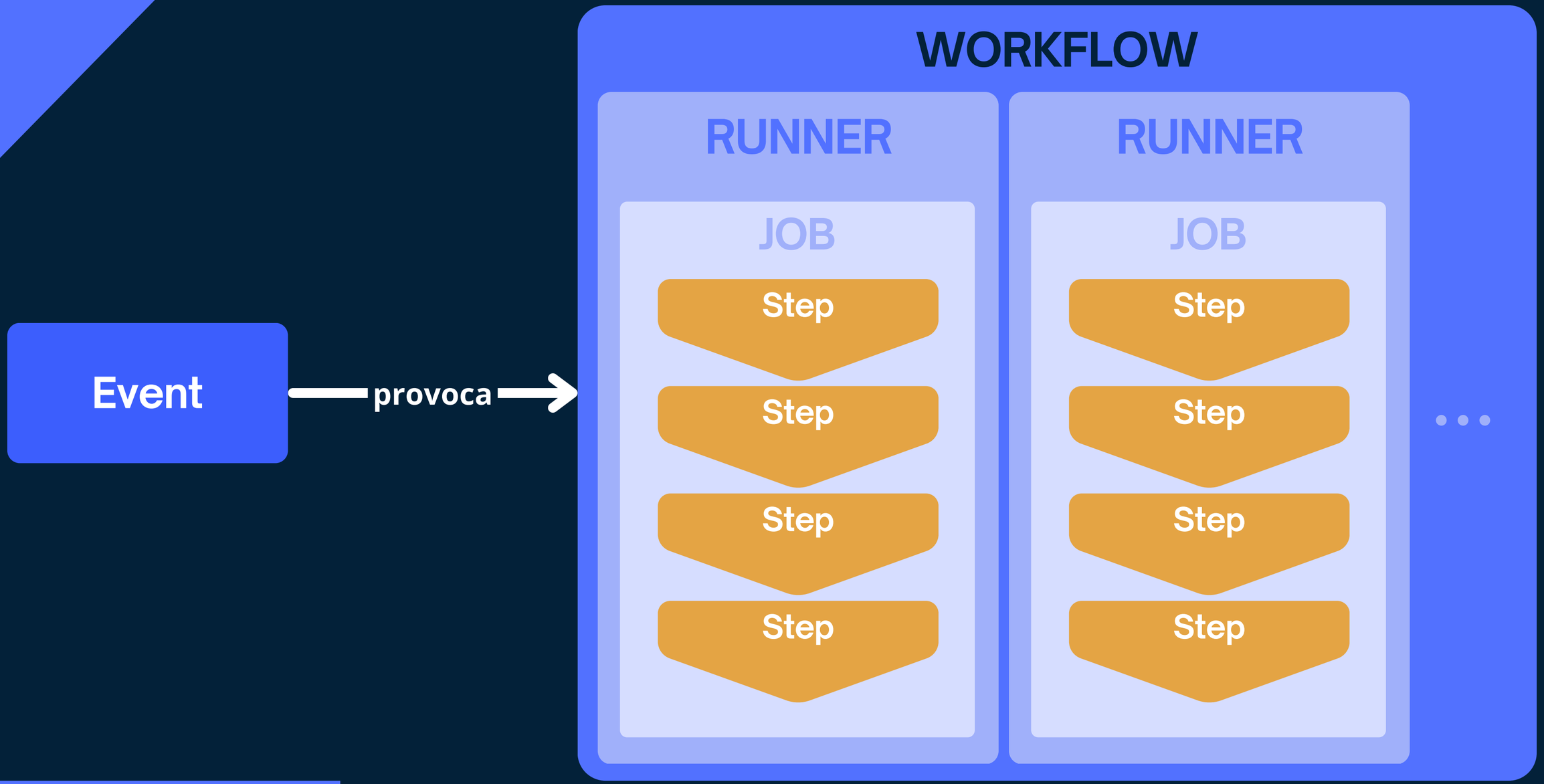


CONTINUOUS INTEGRATION

Build → Test → Merge

CONTINUOUS DEPLOYMENT

deploy automatico


Event
provoca
WORKFLOW
RUNNER
JOB
Step
Step
Step
Step
RUNNER
JOB
Step
Step
Step
Step

Events

Eventi che provocano l’esecuzione di un workflow

- **push**: tutte le volte che dei commit vengono pushati in remoto
- **pull_request**: in base all’*activity type* scelto [created, opened, closed, ...]
- **fork**: tutte le volte che si fa una fork
- **issues**: in base all’activity type scelto [edited, opened, closed, ...]
- **workflow_dispatch**: ci permette di eseguire l’action manualmente da github

Per info vedi la documentazione GitHub:
https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows

Events | filters

Filtri per restringere gli eventi

Alcuni eventi hanno dei filtri per avere un controllo maggiore su come viene provocato il workflow.

Per esempio *push*, può avere specificate alcune **branch**.
Posso quindi fare specifici workflows per main o branch che matchano un pattern definito.
Posso anche scegliere quali branch escludere.

Ci sono anche filtri sui **path** che permettono di provocare un workflow solo se alcuni file specifici/pattern vengono modificati.

Per info vedi la documentazione GitHub:
https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows

Jobs

compongono un workflow

Un workflow è composto da uno o più jobs.

Di predefinito i jobs vengono eseguiti in parallelo.

Si possono avere quanti jobs si vuole all'interno dello stesso workflow.

Un job è composto da una o più azioni (comandi).

Per info vedi la documentazione GitHub:
https://docs.github.com/en/actions/using-jobs/using-jobs-in-a-workflow

YAML

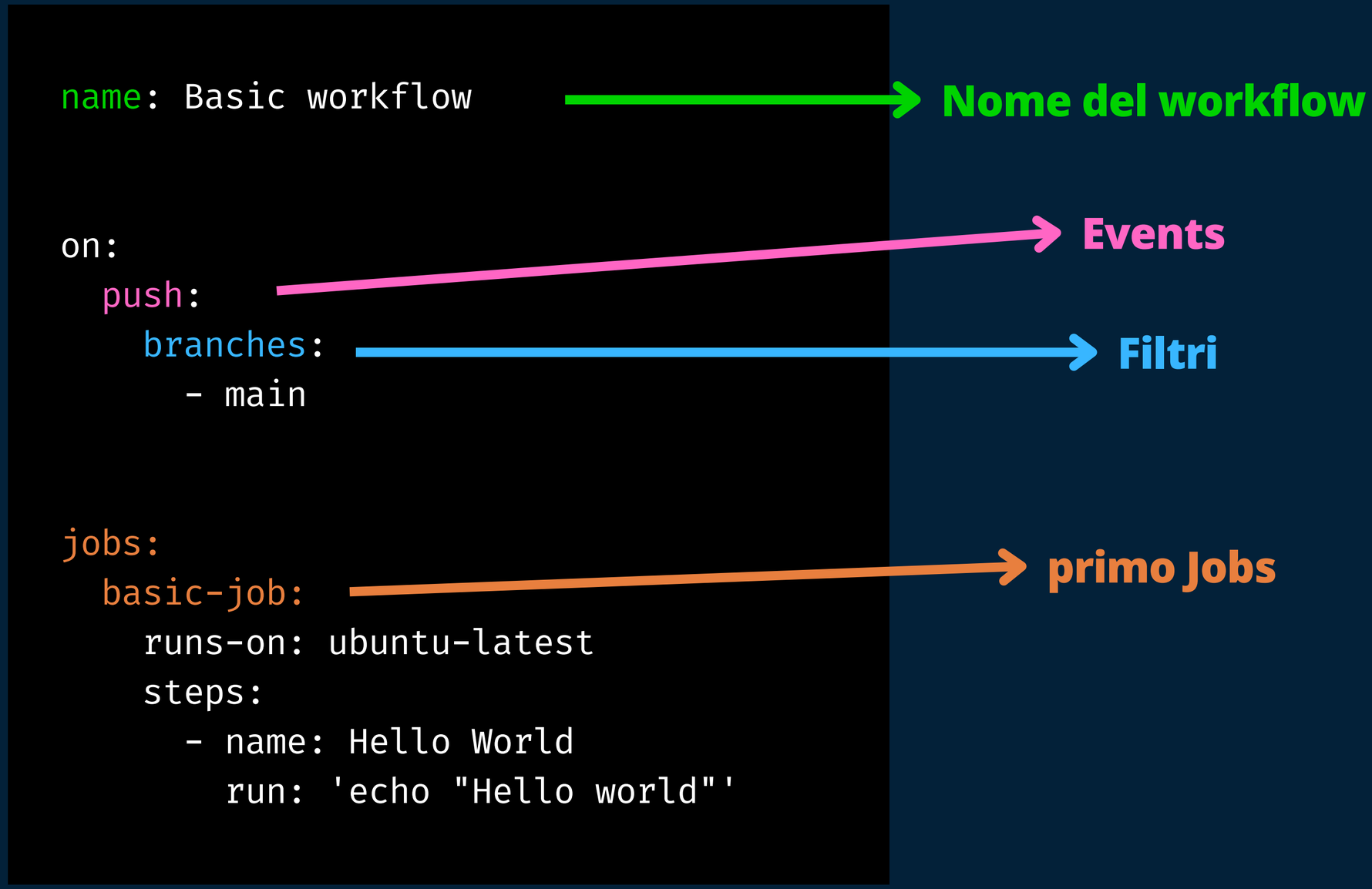
yet another markup language

È un linguaggio di markup che viene usato per i file di configurazione.

```
foo: bar
pleh: help
stuff:
  foo: bar
  bar: foo
```

I workflow di GitHub vengono scritti in YAML

Per info vedi la documentazione GitHub:
https://docs.github.com/en/actions/using-jobs/using-jobs-in-a-workflow


name: Basic workflow
Nome del workflow
on:
push:
Events
branches:
Filtri
- main
jobs:
basic-job:
primo Jobs
runs-on: ubuntu-latest
steps:
- name: Hello World
run: 'echo "Hello world"'

.github/workflows

cartella da creare per contenere tutti i workflow della repository

Contiene tutti i file *.yaml* dei vari workflow.

È importante creare files separati per ogni workflow.

Eventuali workflow esterni alla directory non verranno considerati da GitHub come Actions.

Runs-on

immagine su cui esegue il job

GitHub quando esegue i workflow alloca una Virtual Machine dedicata. Possiamo decidere il suo sistema operativo con l'opzione ***runs-on***

- *ubuntu-latest*
- *windows-latest*
- *macos-latest*
- *...*

Per info vedi la documentazione GitHub:
https://docs.github.com/en/actions/using-jobs/using-jobs-in-a-workflow

Steps

sono le azioni che compongo i jobs

In questo caso è presente un unico step che eegue il comando:

echo "Hello world"

Ci sono anche altri tipi di step oltre a run, il più comune è **uses** che esegue un workflow già definito.

```
...

jobs:
  basic-job:
    runs-on: ubuntu-latest
    steps:
      - name: Hello World
        run: 'echo "Hello world"'
```

Steps | uses

sono le azioni che compongo i jobs

È molto comune voler eseguire dei comandi/azioni sulla propria repository, dovremmo quindi fare un clone sul runner.

Con **uses** possiamo farlo usando un'action preesistente (p.e. checkout)

```
...

jobs:
  basic-job:
    runs-on: ubuntu-latest
    steps:
      - name: Clone repo
        uses: actions/checkout@v4
```

Steps | uses

le più popolari

sono le azioni che compongo i jobs

Jamesives/github-pages-deploy-action@4.1.4	deploy di una cartella nella gh-pages branch
actions/setup-python@v2	setup un environment python
github/super-linter@v4	eseguire una combinazione di linter generando un’unico report

Qui ne trovi tante altre:

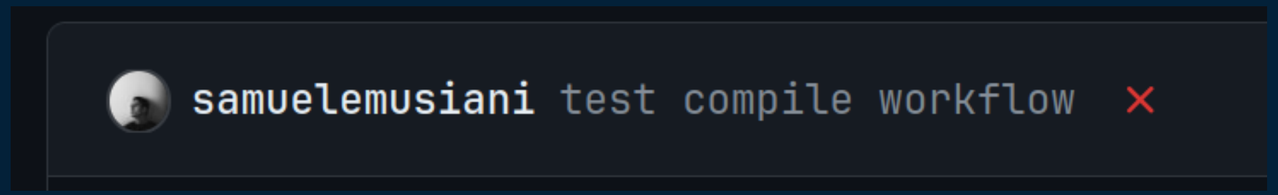
https://github.com/sdras/awesome-actions

Action failed

Le actions possono riscontrare degli errori

Quando un comando del workflow produce un errore, l'actions viene bloccata da GitHub e le viene assegnata lo status di **"failed"**

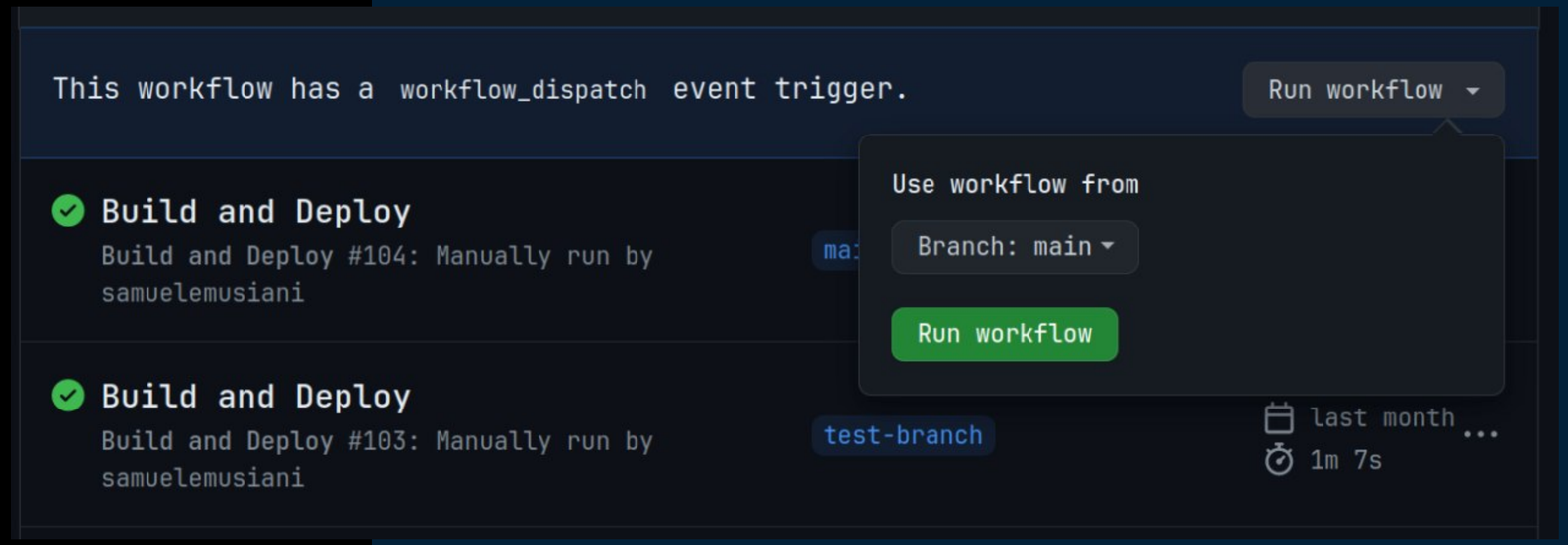
Tutti gli steps successivi non verranno runnati.



Events | Workflow dispatch

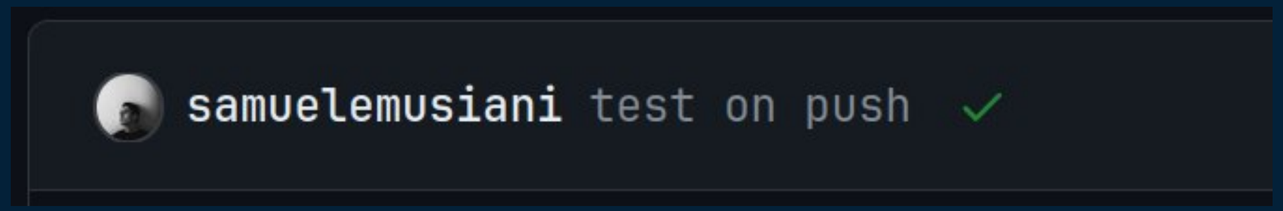
ci permette di eseguire l'actions manualmente

```yaml
name: Basic workflow

on:
 push:
   branches:
   - main
 workflow_dispatch:

...
```



Action successful

Quando tutti i comandi del workflow vengono eseguiti senza errori, la action termina correttamente.



Espressioni ${{ ... }}

Le action possono avere variabili e segreti

Per usarli si usa una sintassi speciale formata da ${{ <nome_variabile> }}

Esempio:
pippo = [1, 2, 3]

Per usarla: ${{ pippo }}

https://docs.github.com/en/actions/learn-github-actions/expressions

Strategy | matrix

un job può essere eseguito su specifici range

Se si vuole testare il progetto su più versioni diverse, invece di creare tante actions è possibile usare le **strategy**.

Il set di passaggi del processo verrà eseguito su ognuna di queste configurazioni.

```
...
jobs:
  node-test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node: [18.x, 19.x, 20.x]
    steps:
      - name: Setup node ${{ matrix.node }}
        uses: actions/setup-node@v2
        with:
          node-version: ${{ matrix.node }}
```

If

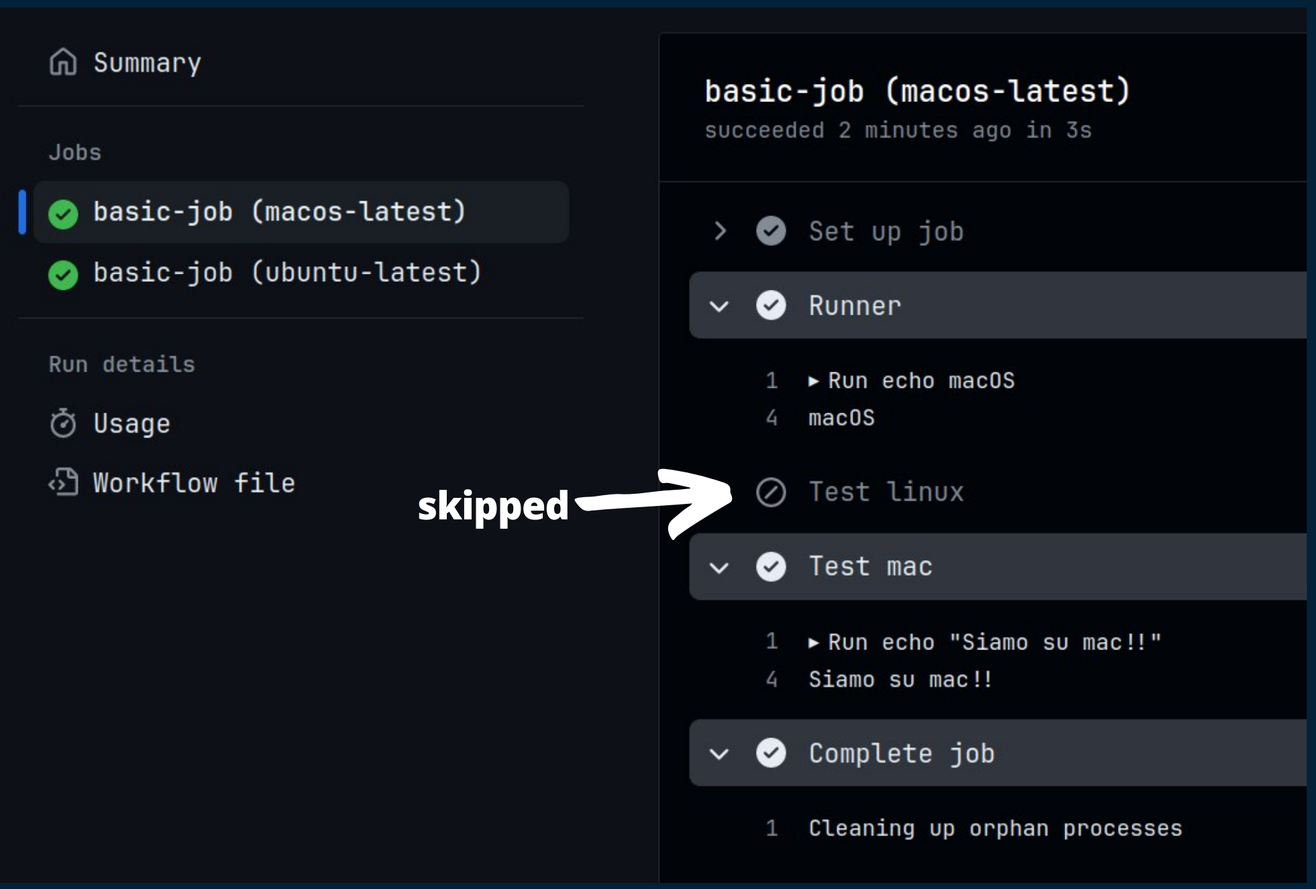
lo step a cui si riferisce viene eseguito se la condizione è soddisfatta

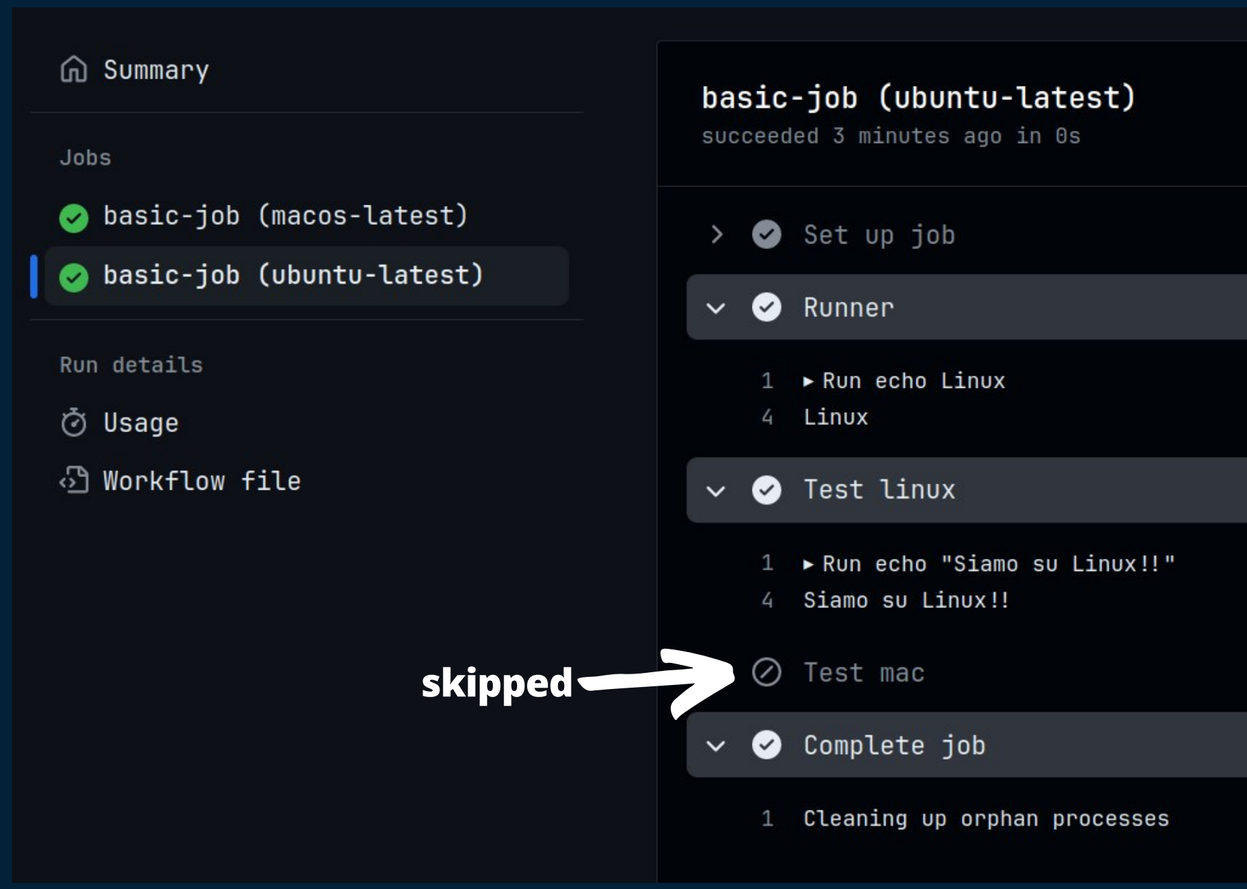
Impedisce l'esecuzione di uno step a meno che non venga soddisfatta una condizione.

if usa un'espressione senza il blocco ${{ ...}}.

```
...

jobs:
  basic-job:
    strategy:
      matrix:
        os: [macos-latest, ubuntu-latest]
    runs-on: ${{ matrix.os }}
    steps:
      - name: Runner
        run: 'echo ${{ runner.os }}'

      - name: Test linux
        if: runner.os == 'Linux'
        run: 'echo "Siamo su Linux!!"'

      - name: Test mac
        if: runner.os == 'macOS'
        run: 'echo "Siamo su mac!!"'
```

If


Summary
Jobs
basic-job (macos-latest)
basic-job (ubuntu-latest)
Run details
Usage
Workflow file
basic-job (macos-latest)
succeeded 2 minutes ago in 3s
Set up job
Runner
1 ▸ Run echo macOS
4 macOS
skipped
Test linux
Test mac
1 ▸ Run echo "Siamo su mac!!"
4 Siamo su mac!!
Complete job
1 Cleaning up orphan processes


Summary
Jobs
basic-job (macos-latest)
basic-job (ubuntu-latest)
Run details
Usage
Workflow file
basic-job (ubuntu-latest)
succeeded 3 minutes ago in 0s
Set up job
Runner
1 ▸ Run echo Linux
4 Linux
Test linux
1 ▸ Run echo "Siamo su Linux!!"
4 Siamo su Linux!!
skipped
Test mac
Complete job
1 Cleaning up orphan processes

Secrets

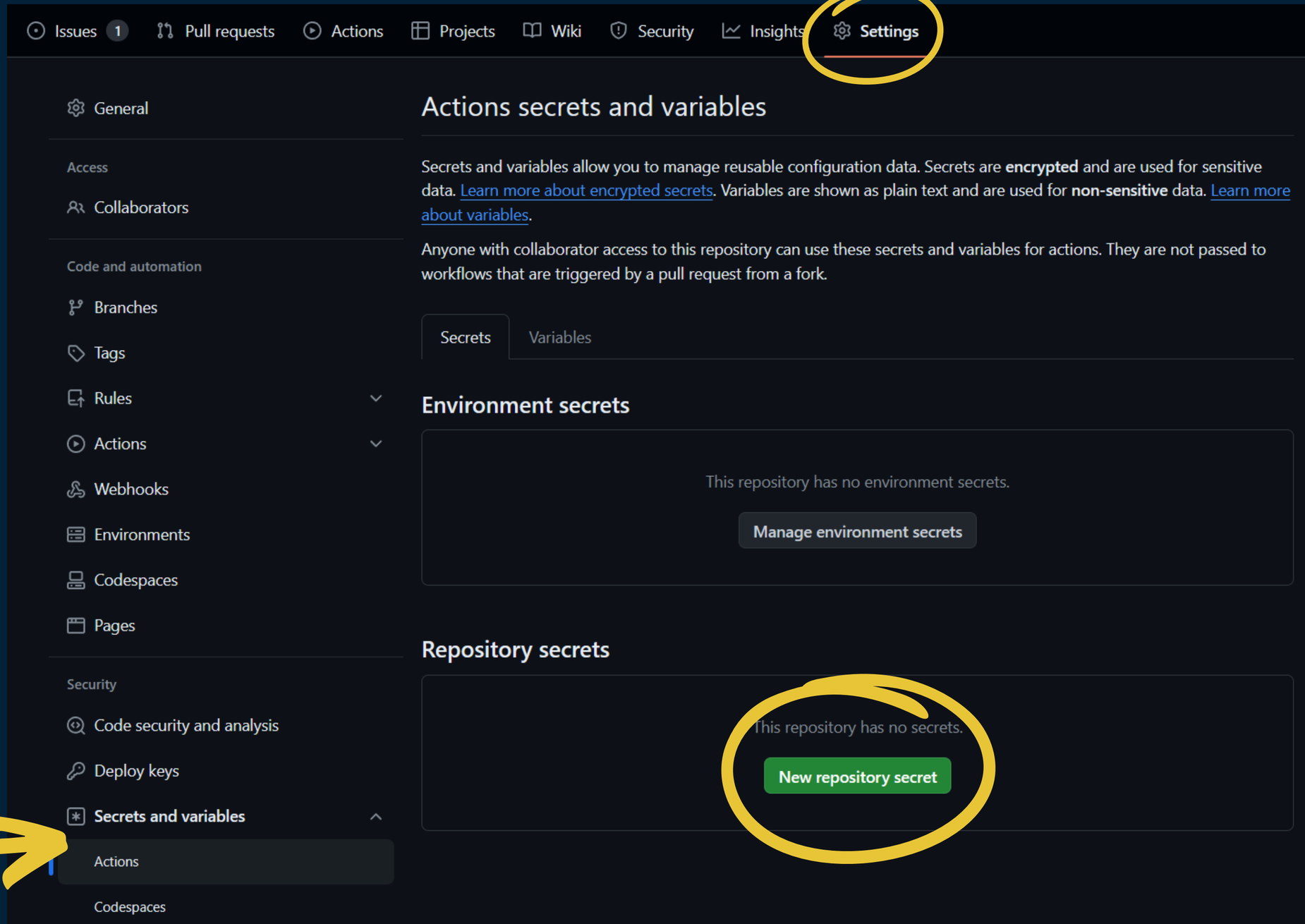
variabili/token/password che possiamo usare in modo sicuro nei jobs

Può essere necessario a volte usare token, password per usare dei comandi come ssh, API, chiave di deploy...

```
...
jobs:
  basic-job:
    runs-on: ubuntu-latest
    steps:
      - name: Clone repo
        run: ssh debian@cs.unibo.it
        with:
          password: ${{ secrets.NOME_SECRETS }}
```

Secrets

Per usarli sono da inserire all'interno delle "Settings" della repository.



Proviamo a creare delle actions per la repository csunibo/lab-git2

creiamo una *fork* della repo

Cloniamo la fork sul nostro pc in locale

```
$  git clone git@github.com:<nome_utente>/lab-git2
```

Se avete già clonato la fork in locale nel precedente laboratorio:

1. aprite la fork su github
2. premere su “sync fork”
3. premere su “update fork”
4. Aprite la repository in locale e mandate un *git pull*

Nella nostra repository vediamo che la cartella *workflow* è già stata creata da noi,
se la apriamo troviamo il nostro *basic_workflow.yaml*

Creiamo il nostro primo workflow

Vogliamo realizzare un'action per compilare *file.cpp*

1. creo il file *compile.yaml*
2. diamo un **nome** all'action
3. vogliamo che **per ogni commit** file.cpp compili
    a. **clonare la repo,**
       usando **un'action già pronta**

```yaml
name: Compile c++

on:
  push

jobs:
  basic-job:
    runs-on: ubuntu-latest
    steps:
      - name: Clone the repo
        uses: actions/checkout@v4
```

Creiamo il nostro primo workflow

Vogliamo realizzare un'action per compilare *file.cpp*

1. creo il file *compile.yaml*
2. diamo un **nome** all'action
3. vogliamo che **per ogni commit** file.cpp compili
   a. **clonare la repo,**
      usando **un'action già pronta**

   b. **installiamo il compilatore g++**

```yaml
name: Compile c++

on:
  push

jobs:
  basic-job:
    runs-on: ubuntu-latest
    steps:
      - name: Clone the repo
        uses: actions/checkout@v4

      - name: Install g++
        run: sudo apt install gcc
```

Creiamo il nostro primo workflow

Vogliamo realizzare un'action per compilare *file.cpp*

1. creo il file *compile.yaml*
2. diamo un **nome** all'action
3. vogliamo che **per ogni commit** file.cpp compili
   a. **clonare la repo,**
      usando **un'action già pronta**
   b. **installiamo il compilatore g++**
   c. usiamo il classico comando per **compilare il file**

```yaml
name: Compile c++

on:
  push

jobs:
  basic-job:
    runs-on: ubuntu-latest
    steps:
      - name: Clone the repo
        uses: actions/checkout@v4

      - name: Install g++
        run: sudo apt install gcc

      - name: Compile the file
        run: g++ file.cpp
```

Needs

un workflow per essere eseguito ha bisogno che prima ne sia eseguito un altro

- Identifica tutti i processi che devono essere completati correttamente prima di essere eseguiti.
- Può essere una stringa o una matrice di stringhe.
- Se un processo ha esito negativo, tutti i processi che lo richiedono vengono ignorati, a meno che non utilizzino un'istruzione condizionale che determina la continuazione del processo.

```
...
jobs:
  basic-job:
    runs-on: ubuntu-latest
    steps:
      - name: Clone repo
        run: ssh debian@cs.unibo.it
  ...
  build:
    name: Build the project
    runs-on: ubuntu-latest
    needs: basic-job
    steps:
      ...
```

GitHub Action Limits

Product	Storage	Minutes per month
GitHub	500 MB	2,000
GitHub Pro	1 GB	3,000
GitHub Free for organizations	500 MB	2,000
GitHub Team	2 GB	3,000
GitHub Enterprise Cloud	50 GB	50,000

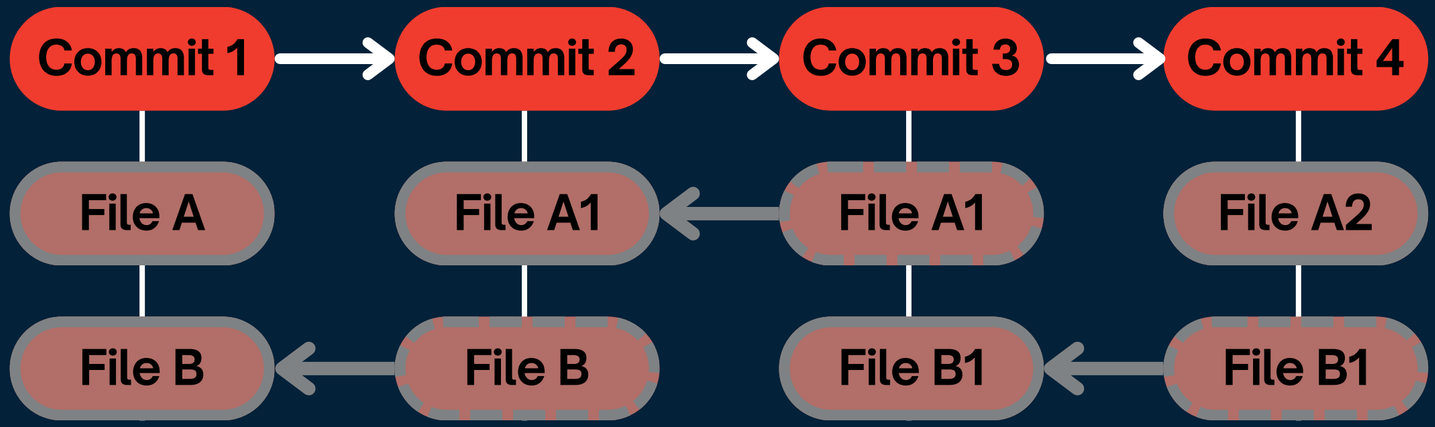
git lfs

git lfs

Git è pensato per lavorare solo con file di testo.

Tutti gli algoritmi di diff sono per file di testo.

Con file binari (come pdf, immagini, audio, video...) non riesce a fare le differenze con la versione precedente: se viene modificato il file, git crea una nuova versione del file da capo.



git lfs large file storage

Se si volesse sviluppare un videogioco, i file audio, le immagini, ecc... appesantirebbero la history di git.
Usando *git lfs* risolviamo questo problema.

Invece di salvare direttamente il file nella history di git, viene salvato un “puntatore” in forma testuale.
I file binari vengono salvati su un server a parte e vengono inclusi nella repository tramite questi “puntatori”.

A livello utente non cambia quasi nulla (infatti continueremo a vedere i binari nella repository) ma git internamente sarà molto più veloce perché nella history non avrà binari.

Installazione

Se non avete già installato git lfs, andate sul sito per scaricarlo: **https://git-lfs.com/**

Per verificare se lo avete aprite un terminale e provate

```
$ git-lfs
```

Una volta installato eseguite il comando per inizializzare git lfs

```
$ git lfs install
```

[questo comando va eseguito solo una volta, dura “per sempre”]

Creiamo la prima repository con lfs

Creiamo una repository da GitHub e cloniamola in locale.

Per usare git lfs su tutti i file .png usiamo

```
$ git lfs track "*.png"
```

Ci creerà un file *.gitattributes* con dentro:

```
*.png filter=lfs diff=lfs merge=lfs -text
```

In questo modo git sa che tutti i file con pattern *.png devono essere gestiti con lfs.

Ora possiamo usare tutti i soliti comandi git come prima, in automatico sarà git a gestire il tutto.

git lfs & GitHub

GitHub ci limita nell’uso di git lfs: limiti di storage e di band_width

possiamo vedere nelle impostazioni > Billing and Plans