





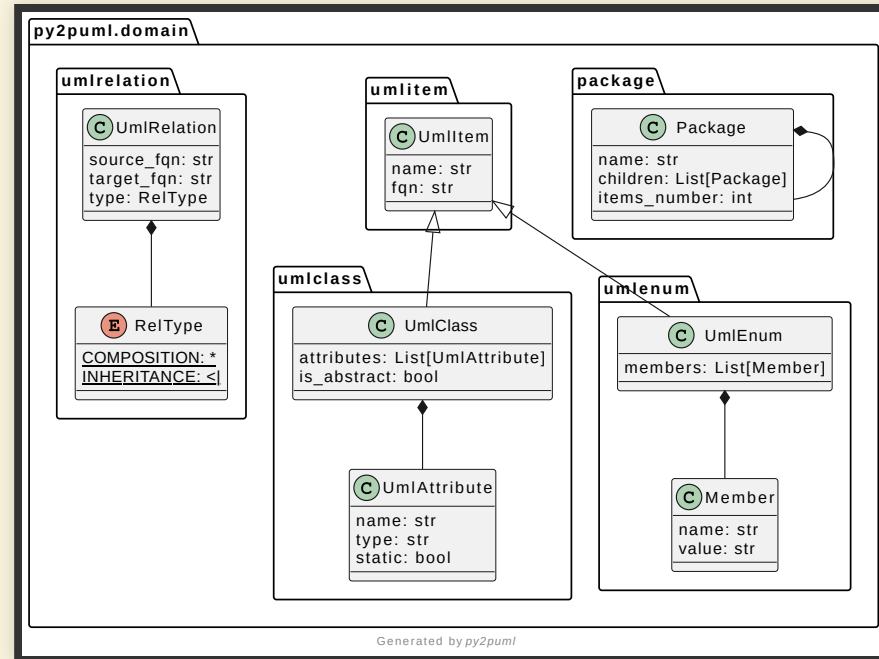
Outiller son projet open-source Python avec les Github actions

Qui suis-je ?

-   dev Python @ Purecontrol
-  github.com/lucsorel (outils code-to-doc)
-  @lucsorelgiffo@floss.social

Le projet open-source Python

<https://github.com/lucsorel/py2puml> : génère un diagramme de classes en scannant les structures de données de la base de code.



Syndrôme de Spiderman

Spiderman

- 1 piqure : 🦂
- grimpe aux murs : 💪
- anime des soirées mousse avec les poignets : 🎉



Syndrôme de Spiderman

Spiderman

- 1 piqure : 🦋
- grimpe aux murs : 💪
- anime des soirées mousse avec les poignets : 🎉



py2puml

- 140+ ★ : 🎉
- 37 issues (24 fermées) : 🦋
- 25 PR (21 fermées) : 😓

*"De nouvelles
responsabilités
demandent de
nouveaux pouvoirs."*

Luc (du 35) 🕸

Accueillir et cadrer les contributions

Accueillir et cadrer les contributions

- non-régression des fonctionnalités

Accueillir et cadrer les contributions

- non-régression des fonctionnalités
- homogénéité des pratiques
 - formatage de code
 - qualité de code



CONTRIBUTING.md dans le dépôt ?



"bla bla bla" → ❌

Des conventions non outillées finissent dans l'oubli.



Intégration continue

- non-régression → **tests automatisés** (+ couverture de code testé)
- homogénéité de la base de code
 - formatage de code → **formateur**
 - qualité de code → **linter**

Mise en place des Github actions

Syntaxe **déclarative** permettant d'exécuter des commandes ou des outils au fil du cycle de vie *git* du projet.

```
super-projet/  
├── .github/  
│   └── workflows/  
│       ├── {nom_de_workflow}.yaml # ci.yaml, par exemple  
│       └── ...  
└── ...
```



Mark Hamill



Mark Yaml

Anatomie progressive d'un workflow d'intégration continue Github

Départ : système d'exploitation + checkout du code source

```
name: Python CI # facultatif

on: push          # évènement(s) git déclencheur(s)

jobs:
  build:          # le nom de l'opération
    runs-on: ubuntu-latest # système d'exploitation utilisé
    steps:        # étapes de l'opération
      - name: Récupération du code # facultatif
        uses: actions/checkout@v4 # utilisation d'une recette existante (▲ @version)
```

Documentation officielle des Github actions.

Actions : les recettes de base

Exemples : github.com/actions/checkout, github.com/abatilo/actions-poetry

Actions : les recettes de base

- action = brique réutilisable et adaptable à votre besoin
 - chacune a son dépôt
 - configuration décrite dans `action.yml`
 - parfois accompagnée de scripts référencés dans `action.yml`

Exemples : github.com/actions/checkout, github.com/abatilo/actions-poetry

Actions : les recettes de base

- action = brique réutilisable et adaptable à votre besoin
 - chacune a son dépôt
 - configuration décrite dans `action.yml`
 - parfois accompagnée de scripts référencés dans `action.yml`
- 62 actions "officielles" proposées par Github (septembre 2023)

Exemples : github.com/actions/checkout, github.com/abatilo/actions-poetry

Actions : les recettes de base

- action = brique réutilisable et adaptable à votre besoin
 - chacune a son dépôt
 - configuration décrite dans `action.yml`
 - parfois accompagnée de scripts référencés dans `action.yml`
- 62 actions "officielles" proposées par Github (septembre 2023)
- nombreuses actions communautaires


Exemples : github.com/actions/checkout, github.com/abatilo/actions-poetry

Étapes de build du projet


Étapes de build du projet

1. récupérer les sources du projet 


Étapes de build du projet

1. récupérer les sources du projet 
2. installer python


Étapes de build du projet

1. récupérer les sources du projet 
2. installer python
3. installer poetry


Étapes de build du projet

1. récupérer les sources du projet 
2. installer python
3. installer poetry
4. installer les dépendances

Étapes de build du projet

1. récupérer les sources du projet 
2. installer python
3. installer poetry
4. installer les dépendances
5. lancer les tests automatisés + couverture

Étapes de build du projet

1. récupérer les sources du projet 
2. installer python
3. installer poetry
4. installer les dépendances
5. lancer les tests automatisés + couverture
6. auditer la qualité du code

Installation de Python - pyenv

Projet utilisant **pyenv** pour définir la version de python.

```
name: Python CI
on: push

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Installation de Python
        uses: actions/setup-python@v4           # action officielle
        with:                                   # configuration pour pyenv
          python-version-file: '.python-version' # màj transparente de l'IC 👍
```


Installation de Python - alternatives

Configuration alternatives de l'action :

```
steps:
  # dernière version disponible (Δ varie en fonction du runner)
  - uses: actions/setup-python@v4

  # version spécifiée en dur 🙅
  - uses: actions/setup-python@v4
    with:
      python-version: '3.9'

  # utilisation de PyPy
  - uses: actions/setup-python@v4
    with:
      python-version: 'pypy3.9'
```

Voir d'autres cas d'usage avancés (intervalle ou matrice de versions, caches pour outils spécifiques, etc.).

Installation des dépendances - poetry

- utilisation de **poetry** pour la gestion des dépendances
- dossier `.venv/` local (à mettre en cache 💡)

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      # [...]
      - name: Installation de poetry
        uses: abatilo/actions-poetry@v2 # action communautaire 🔍
        with:
          poetry-version: 1.5.1 # optionnel

      - name: Mise en cache de l'environnement virtuel
        uses: actions/cache@v3 # action officielle
        with:
          path: ./venv
          key: venv-${{ hashFiles('poetry.lock') }} # clé d'éviction sur le lock-file

      - name: Installation des dépendances du projet
        run: poetry install # exécution explicite d'une commande 🔍
```

Installation des dépendances - requirements.txt

Alternative avec pip + requirements.txt directement 🙋 :

```
steps:
- uses: actions/setup-python@v4
  with:
    python-version: '3.9'
    cache: 'pip' # utilisation d'un cache pour pip

- run: pip install -r requirements.txt # pas besoin d'environnement virtuel
      # au sein d'un runner
```

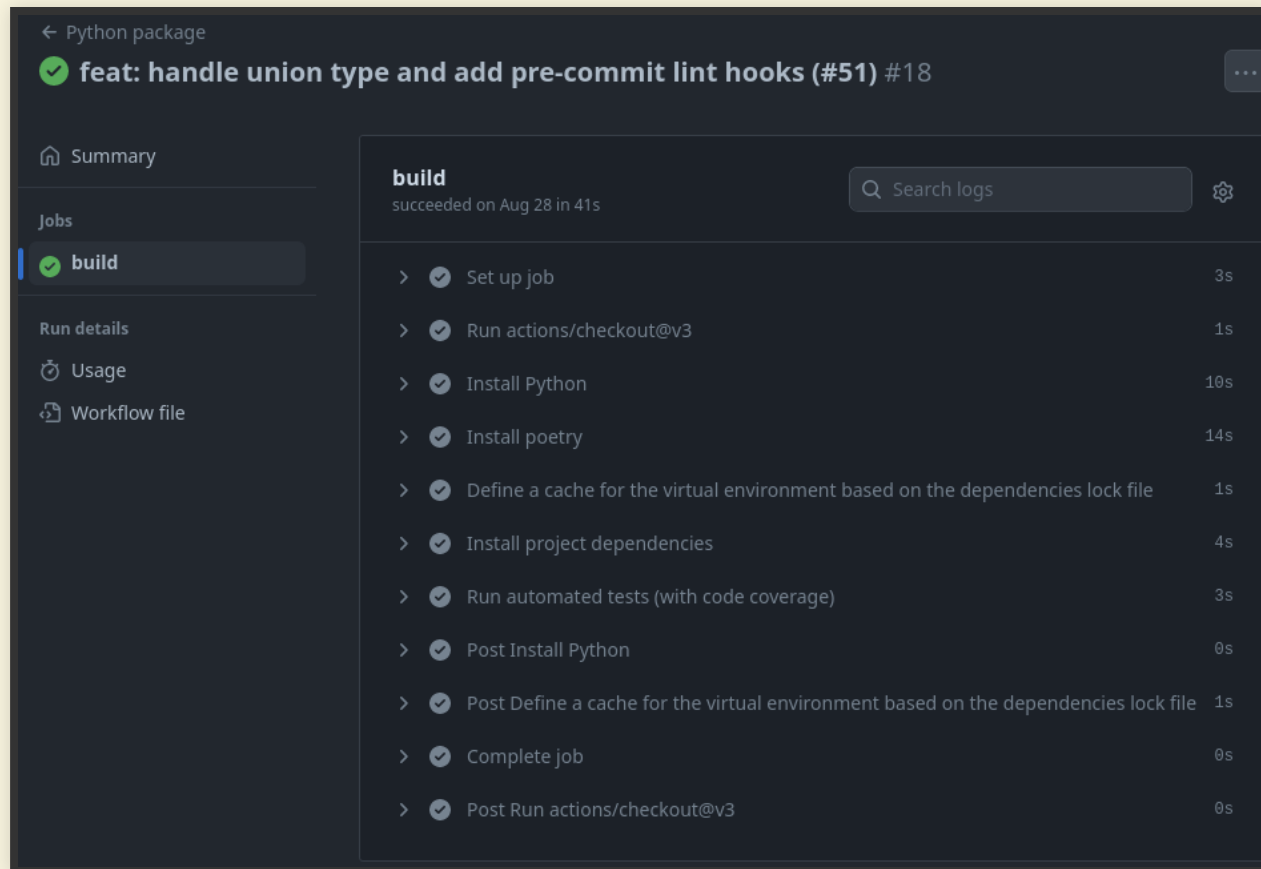
Exécution des tests

```
env:
  MIN_CODE_COVERAGE_PERCENT: 93 # 👉 définition d'une variable

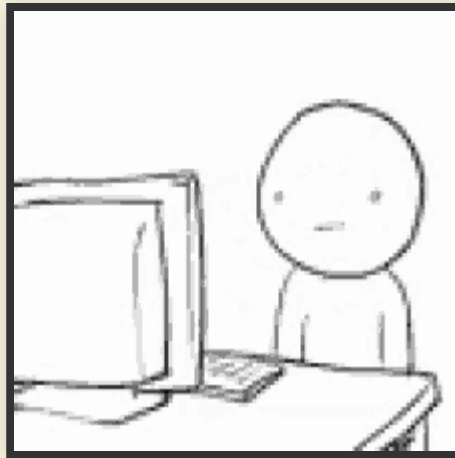
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      # [...]
      - name: Lancement des tests automatisés (avec couverture de code)
        run: > # '>' permet de séparer une commande sur plusieurs lignes
          poetry run pytest -v --cov=py2puml --cov-branch
          --cov-report term-missing
          --cov-fail-under $MIN_CODE_COVERAGE_PERCENT # utilisation 👉
          # l'opération sera en échec si le taux de couverture descend sous le seuil
```

Consultation des exécutions de workflow dans Github

1. cliquer sur le menu **Actions** de votre dépôt sur Github
2. cliquer sur une des exécutions de workflow



Démo !



github.com/lucsorel/math-cli

Et les hooks de pre-commit ?

L'outil **pre-commit** utilise les mécanismes de hook de `git` pour lancer des outils à différentes étapes du cycle de vie `git` du projet.



Figure 1. Rediffusion : <https://www.youtube.com/watch?v=IOHrTE45RVM>

Exemple de configuration pre-commit (.pre-commit-config.yaml)

```
repos:
-   repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.5.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: double-quote-string-fixer
-   repo: https://github.com/google/yapf
    rev: v0.40.2
    hooks:
      - id: yapf
        additional_dependencies: [toml]
-   repo: https://github.com/astal-sh/ruff-pre-commit
    rev: v0.0.292
    hooks:
      - id: ruff
```

Voir pre-commit.com/hooks.html.

Utiliser pre-commit en intégration continue

À la dure dans le runner Github d'intégration continue

- installer git
- lancer pre-commit

Ou via une intégration avec le service pre-commit.ci

Intégrer pre-commit.ci dans son projet Github 1/5

1. aller sur <https://pre-commit.ci/>
2. s'identifier avec Github



pre-commit ci

a continuous integration service for the pre-commit framework

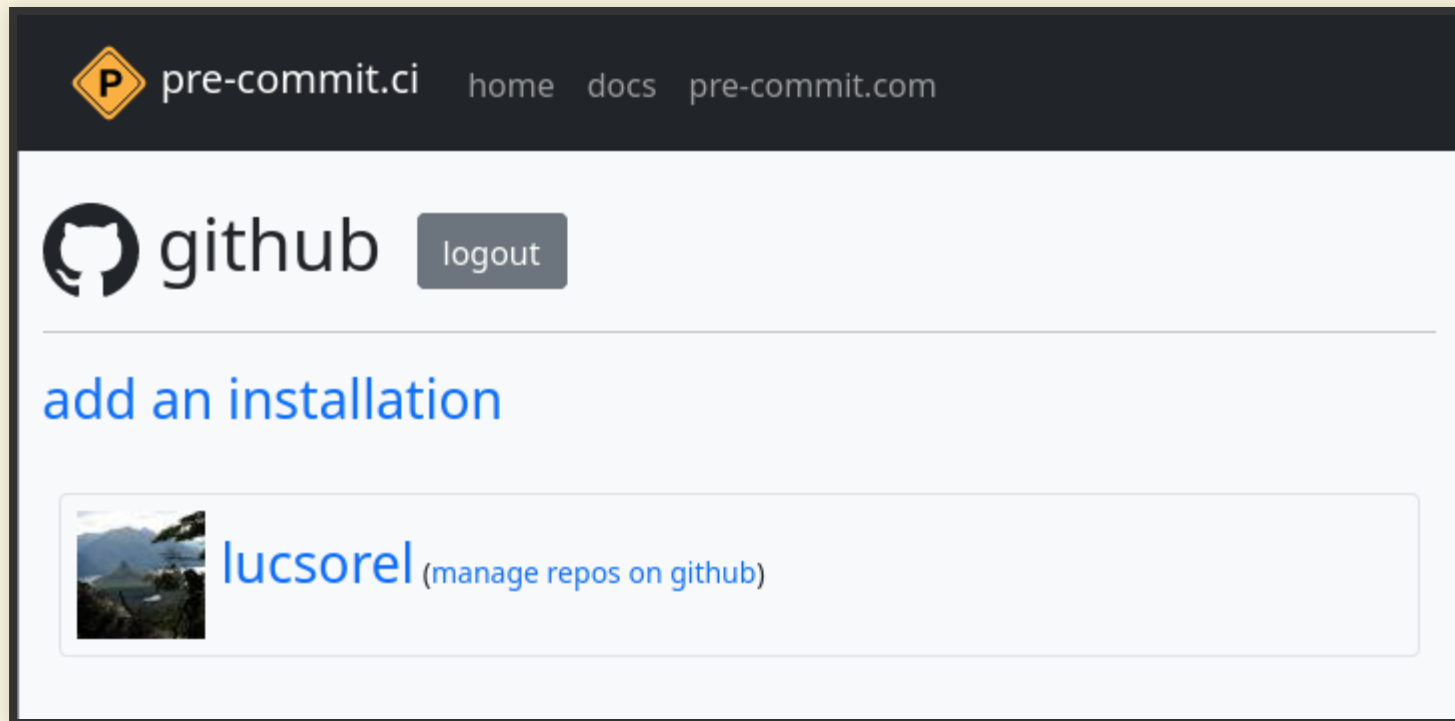
Developers spend a fair chunk of time during their development flow on fixing relatively trivial problems in their code. **pre-commit.ci** both enforces that these issues are discovered (which is opt-in for each developer's workflow via **pre-commit**) but also fixes the issues automatically, letting developers focus their time on more valuable problems.



Sign In With GitHub

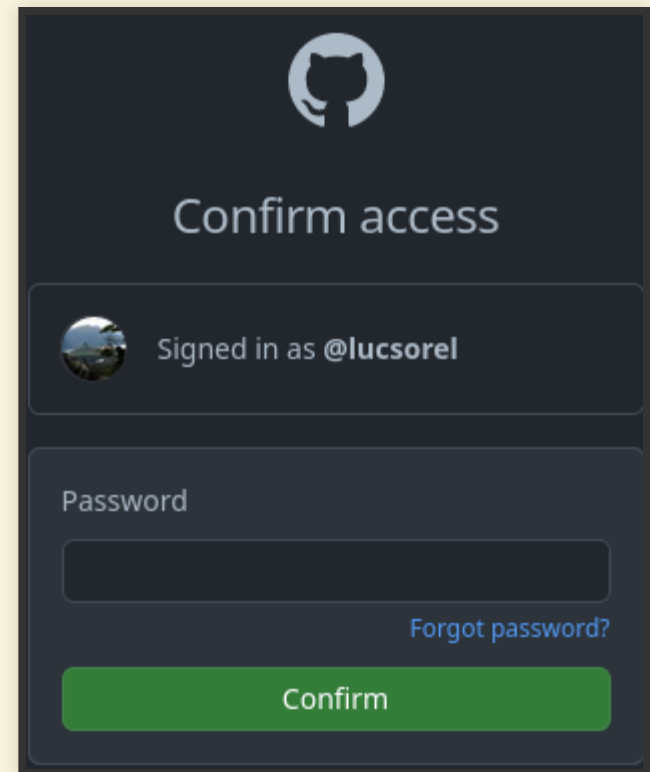
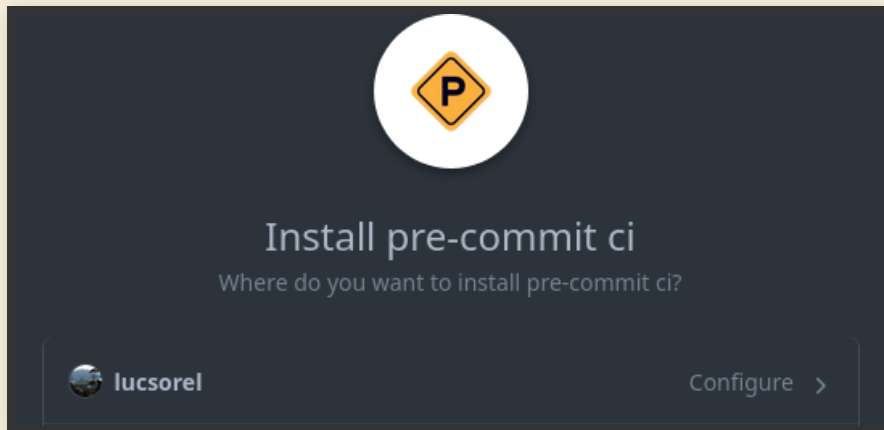
Intégrer pre-commit.ci dans son projet Github 2/5

- ajouter une installation



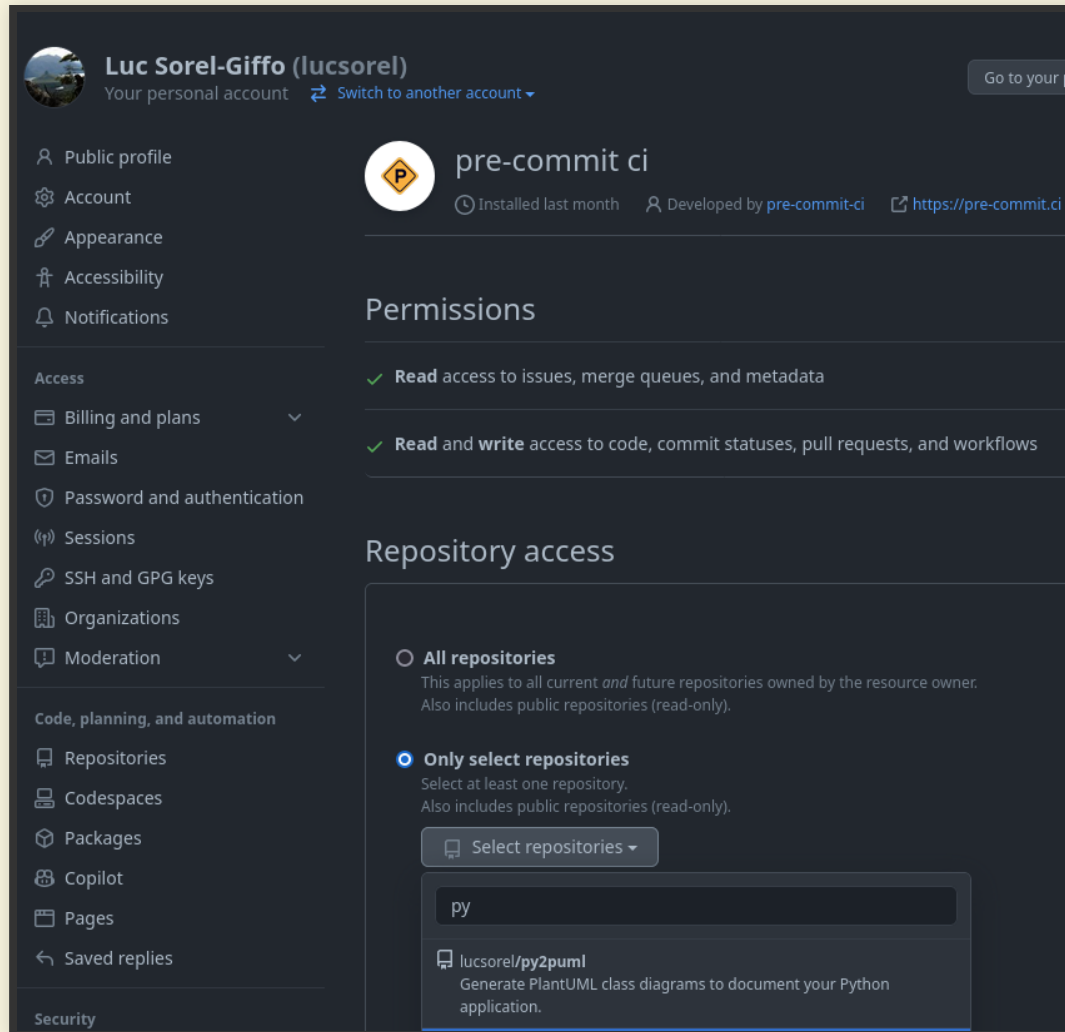
Intégrer pre-commit.ci dans son projet Github 3/5

- configurer l'intégration
- confirmer l'accès au compte



Intégrer pre-commit.ci dans son projet Github 4/5

- sélectionner le dépôt de code à intégrer



Intégrer pre-commit.ci dans son projet Github 5/5

👉 Ne se déclenche que dans le cadre d'une pull-request.

The screenshot displays the pre-commit.ci web interface. At the top, there's a navigation bar with the pre-commit.ci logo, links for home, docs, and pre-commit.com, and the username lucstorel. Below the navigation bar is a green header with a large white checkmark icon. The main content area shows the repository name `lucstorel / py2puml · #62` and the commit hash `87`. It indicates the commit was queued at `2023-10-02 19:41:48.610909` and took a total time of `17.8s` to complete. A list of steps is shown with their respective durations: `queue (79ms)`, `mergeable check (985ms)`, `clone (306ms)`, `ci config (1ms)`, `pull image (303µs)`, `build (13.4s)`, `download (752ms)`, and `run (2.3s)`. The `run` step is highlighted in green, indicating it passed. Below the list, a detailed log of the `run` step is shown, listing various checks and their results, all of which are `Passed`.

pre-commit.ci home docs pre-commit.com lucstorel

✓

lucstorel / py2puml · #62 87

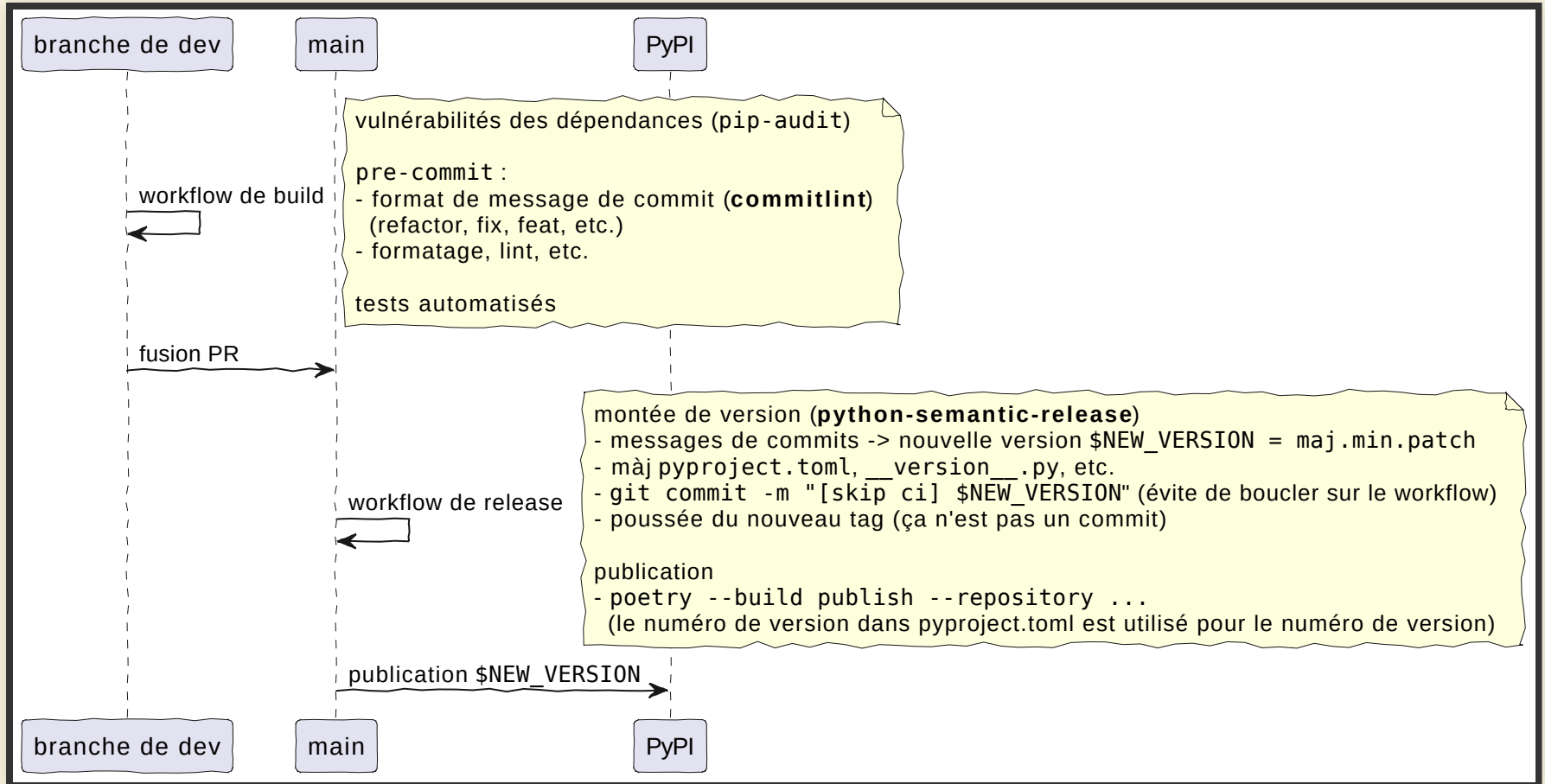
queued at: 2023-10-02 19:41:48.610909

total time to completion: 17.8s

- queue (79ms)
- mergeable check (985ms)
- clone (306ms)
- ci config (1ms)
- pull image (303µs)
- build (13.4s)
- download (752ms)
- run (2.3s)

```
check yaml.....Passed
check toml.....Passed
trim trailing whitespace.....Passed
fix end of files.....Passed
fix double quoted strings.....Passed
check for added large files.....Passed
python tests naming.....Passed
isort.....Passed
Yapf.....Passed
ruff.....Passed
```

Prochaines étapes



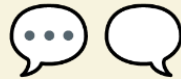
Voir : `pip-audit`, utiliser `skip ci`, hook `commitlint`, outil `Python Semantic Release`

Les Github actions !



- système gratuit de CI / CD
- YAML déclaratif
- originalité des actions : briques composables
- s'adapte à une diversité de projets

Merci !



Des questions ?

Présentation à retrouver sur github.com/lucsorel/conferences/tree/main/python-rennes-2023.10.10-cicd-projets-python 