

DAWAN Paris
DAWAN Nantes
DAWAN Lyon

11,rue Antoine Bourdelle, 75015 PARIS
32, Bd Vincent Gâche, 5e étage - 44200 NANTES
Bt Banque Rhône Alpes, 2ème étage - 235 cours Lafayette 69006 LYON



Intégration Continue: Exemple gitlab-ci & python

Plus d'info sur <http://www.dawan.fr> ou 0810.001.917

Formateur: Matthieu LAMAMRA

Rappels PYTHON (1/2)

- Environnement virtuel
 - Intérêt: contrôle et isolation des versions de l'interpréteur et des dépendances
 - Installation : `sudo apt install python3-venv`
 - Définir un répertoire de travail *workdir*
 - Activation / Désactivation :

```
$ python3 -m venv /path/to/workdir
```

```
$ source /path/to/workdir/bin/activate
```

```
> \path\to\workdir\Scripts\activate.bat
```

```
[>|$] deactivate
```

Rappels PYTHON (2/2)

- Gestion des dépendances
 - À l'intérieur de l'environnement virtuel, pip ne voit plus les dépendances extérieures
 - intérêt : on installe seulement les dépendances du(es) projet(s) de *workdir*
 - On fixe ces dépendances avec la commandes pip3 freeze
 - On installe une liste de dépendances avec la commande pip3 install -r *fichier*

```
(venv)$ pip3 freeze > requirements.txt
```

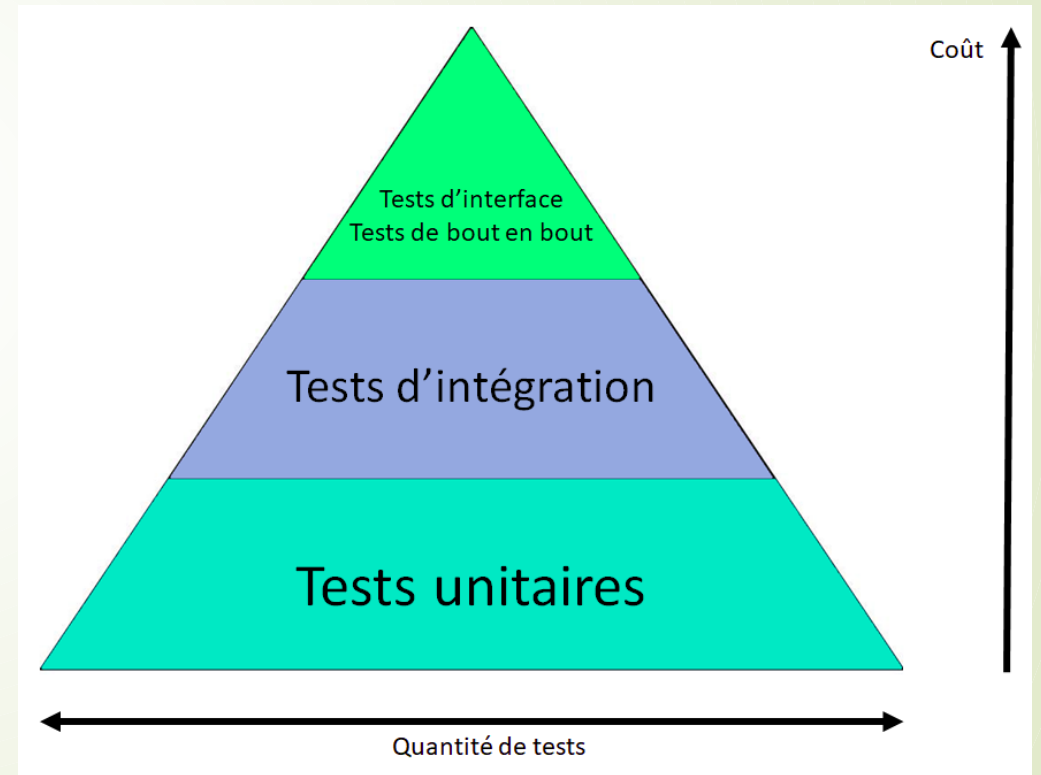
```
$ pip3 install -r /path/to/workdir/requirements.txt
```

Intégration Continue

- Tester le code

On prend en compte la granularité du code :

- Les tests unitaires valident les éléments logiques élémentaires du code : fonctions ou méthodes d'objet si POO
- Les test d'intégration valident et assurent la cohérence des structures intermédiaires du code : classes, composants, modules
- Les test d'interface ou tests de bout en bout (End to End ou e2e) valident le comportement global de l'application depuis ses points d'entrées (interfaces)



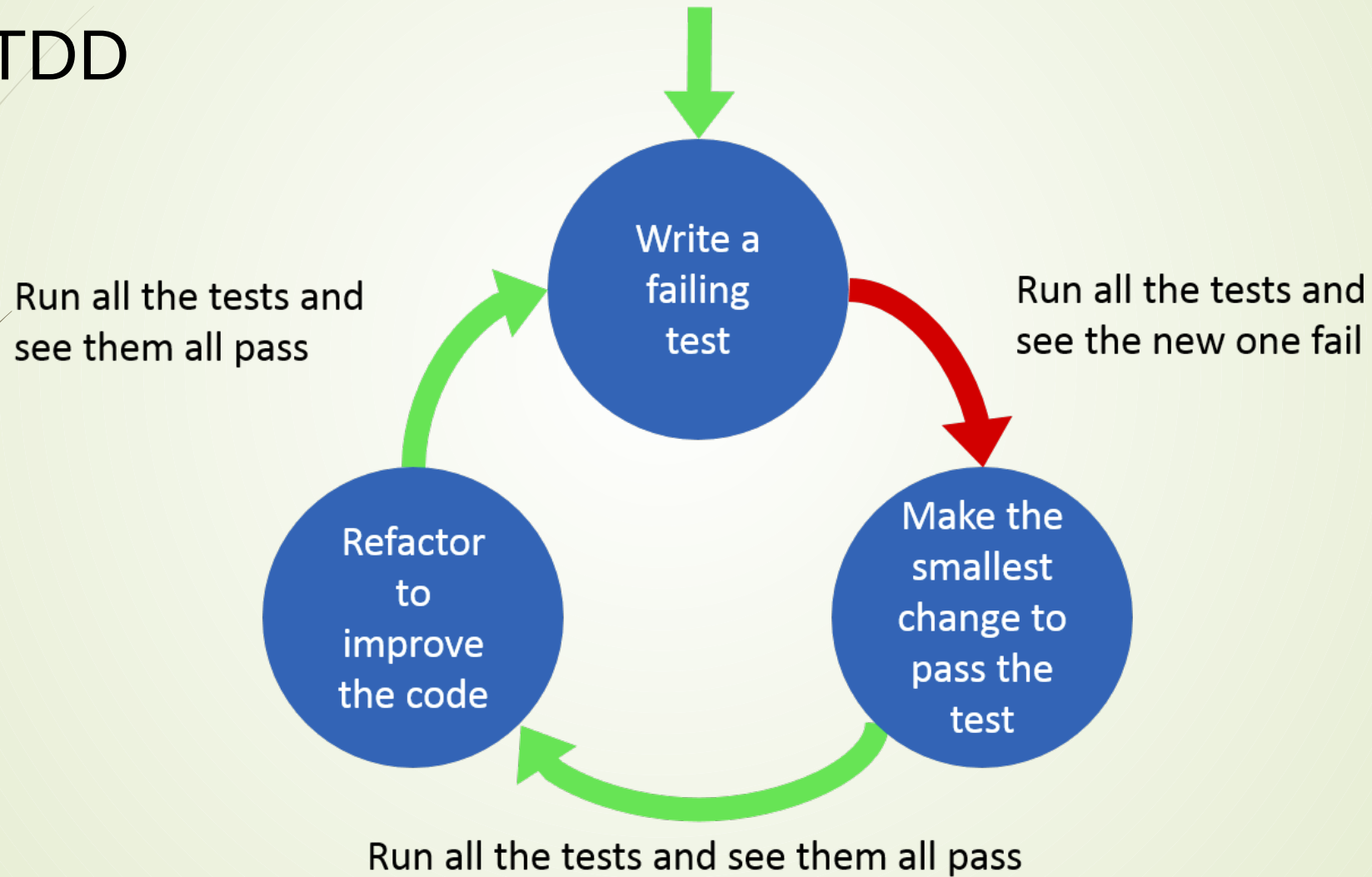
Intégration Continue

- TDD

- « Test Driven Development » ou développement piloté par les tests :
- Consiste à développer en commençant par coder les conditions d'échec du code : le test
- Ensuite on produit un code qui doit passer le test
- Enfin on remanie ce code à des fins d'optimisation et de non regression « Refactoring »
- On poursuit ces itérations sur l'ensemble des fonctionnalités à coder
- La production de tests systématique, alliée à la standardisation des ces tests par des outils dédiés sont une condition nécessaire de l'automatisation d'un processus d'intégration continue
- La TDD est emblématique d'une démarche « shift Left », car les tests sont avancés au niveau du développement

Intégration Continue

- TDD

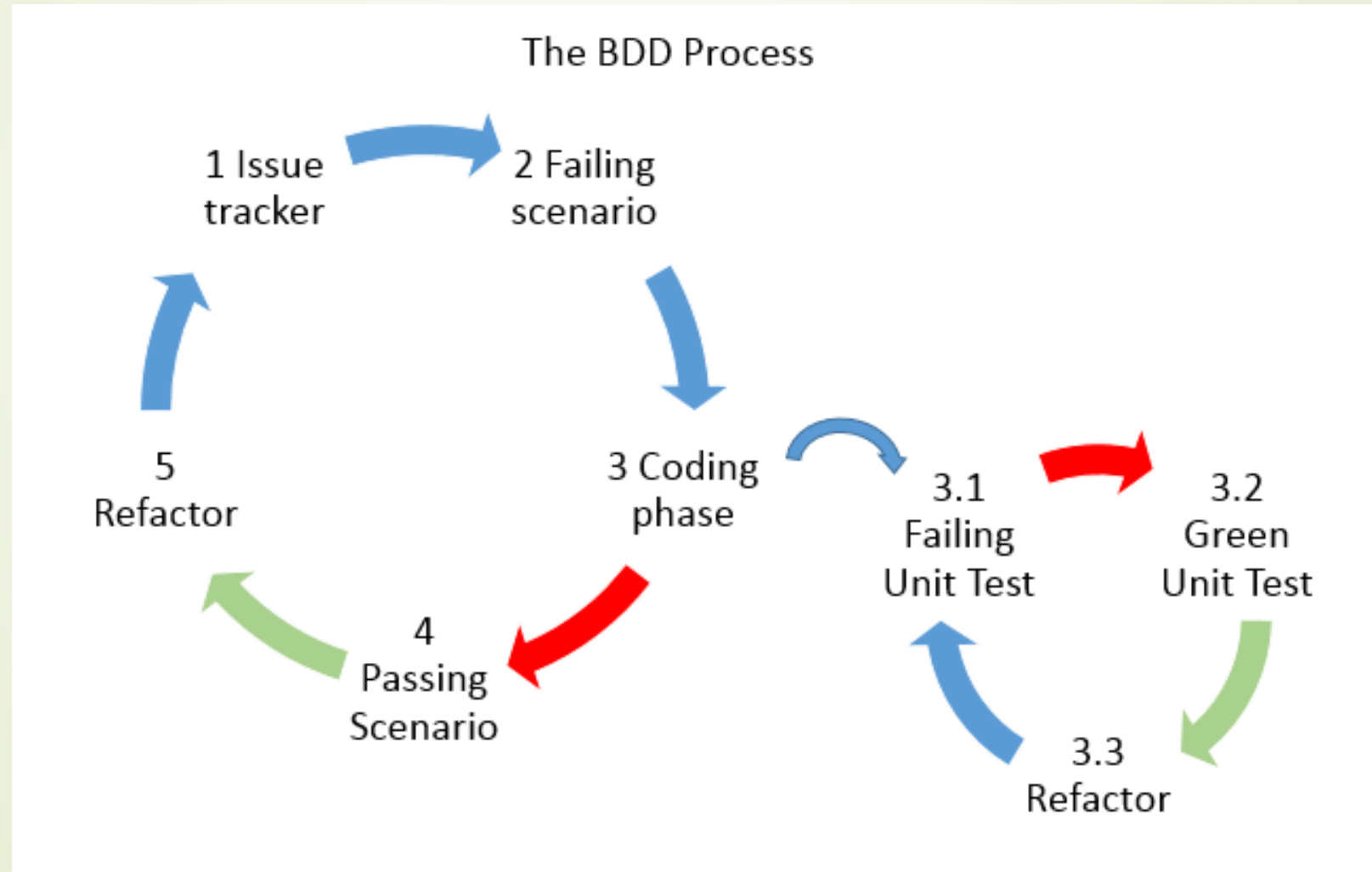


Intégration Continue

- BDD
 - « Behavior Driven Development » ou développement piloté par les comportements :
 - On parle ici des comportements attendus du produit, soit les spécifications fonctionnelles (ou « User Stories ») fruits des échanges entre le client et l'équipe technique
 - D'abord rédigées en langage naturel, ces spécifications sont réécrites dans un cadre formel tel que given / when / then
 - On peut agrémenter cette description d'exemples concrets, si possible automatisables
 - On applique La TDD sur cet ensemble
 - La BDD permet de rassurer le client sur le respect de son expression de besoin là où la TDD valide l'aspect technique du code, ou les spécifications détaillées

Intégration Continue

- BDD



Cl.yml : tests unitaires (1/3)

- Le module python unittest
 - 1 Classe de test héritant de **unittest.TestCase** par fonction / méthode / classe
 - Le nom de chaque méthode de test doit commencer par « **test** »
 - Chaque méthode de test doit exprimer au moins une fonction héritée self.assertXXXX...

Les méthodes setUp et tearDown sont exécutées respectivement avant et après chaque test, pour gérer son environnement

Ce sont des « fixtures »

```
import unittest
from bank import Person, Client, Account
from datetime import *

class TestPersonFullName(unittest.TestCase):

    def setUp(self):
        self.person = Person(1, "Matthieu", "LAMAMRA")

    def testFullName(self):
        self.assertEqual("Matthieu LAMAMRA", self.person.getFullName())
```

tester le module

```
if __name__ == '__main__':
    unittest.main()
```

Cl.yml : tests unitaires (2/3)

- Quelques méthodes « assert »

<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x) is True</code>
<code>assertFalse(x)</code>	<code>bool(x) is False</code>
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assertIsNot(a, b)</code>	<code>a is not b</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assertIsNotNone(x)</code>	<code>x is not None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertNotIn(a, b)</code>	<code>a not in b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>

CI.yml : tests unitaires (3/3)

- Déclenchement automatique des tests
 - Si les fichiers de tests python sont placés **à la racine du projet** :

```
units:  
  before_script:  
    - apt update  
    - apt install -y python3  
    - apt install -y python3-pip  
    - pip3 install -r $CI_PROJECT_DIR/requirements.txt  
  script:  
    - python3 -m unittest discover
```

Cl.yml : suites de test (1/2)

- Déclenchement automatique de suites de tests via unittest

```
# run_tests.py
def run_tests():
    suite = unittest.TestSuite()
    try:
        suite.addTest(TestClient)
        suite.addTest(TestAccount)
    except ValueError:
        print("unknown test, use: --help")
        sys.exit()

    runner = unittest.TextTestRunner()
    runner.run(suite)
```

```
# terminal
$ python3 -m unittest -v run_tests.py
```

Cl.yml : suites de test (2/2)

- Déclenchement automatique de suites de tests via python

- `unittest.TestLoader` gère les ajouts de tests
- `unittest.xxxTestRunner` gère le format d'affichage du rapport de test (txt, html, xml, json...)
- Le format `xml` est compatible avec `Junit`, format de tests unitaires géré par `gitlab CI`

```
# coverage_tests.py
def run_tests():
    """ lancement des tests sur client et account """
    suite, loader = TestSuite(), TestLoader()
    try:
        suite.addTest(loader.loadTestsFromModule(bank))
    return suite

if __name__ == "__main__":
    runner = XMLTestRunner(output="tests_artifacts")
    runner.run(run_tests())

# terminal
$ python3 coverage_tests.py
```


CI.yml : rapport junit (1/2)

- Remontée du rapport de test dans gitlab
 - La clé « **artifacts:reports** » permet de demander l'affichage d'un artefact sur l'interface de gitlab
 - La clé « **artifacts:reports:junit** » fait remonter le rapport de test au format JUnit dans le job

```
units:  
  stage: tests  
  before_script:  
    - pip3 install -r $CI_PROJECT_DIR/requirements.txt  
  script:  
    - python3 coverage_tests.py  
  artifacts:  
    reports:  
      junit: "$CI_PROJECT_DIR/tests_artifacts/TEST*.xml"
```

Cl.yml : rapport jUnit (2/2)

- Affichage des résultats de tests dans le job

[Pipeline](#) [Needs](#) [Jobs 1](#) **[Tests 2](#)**

[<](#) **units**

2 tests





0 failures

0 errors

100% success rate

16.00ms

Tests

Suite	Name	Filename	Status	Duration	Details
bank.test_account.TestAccount	testOverdraft	bank/test_account.py 		15.00ms	View details
bank.test_client.TestClient	testFullName	bank/test_client.py 		1.00ms	View details

Cl.yml : Couverture de code

- Analyse du taux de code testé

```
$ pip3 install coverage
```

```
$ coverage run coverage_tests.py
```

```
$ coverage report
```

Name	Stmts	Miss	Cover
-----	-----	-----	-----
bank/__init__.py	4	0	100%
bank/account.py	30	6	80%
bank/client.py	23	2	91%
bank/test_account.py	11	1	91%
bank/test_client.py	9	1	89%
run_tests.py	15	3	80%
-----	-----	-----	-----
TOTAL	92	13	86%

CI.yml : Couverture de code

- Mise en forme dans Gitlab CI
- Placer l'expression régulière du total de couverture dans **Settings** → **CI/CD** → **General Pipelines**

Test coverage parsing

/ ^TOTAL.+?(\\d+\\%)\$ /

- Générer le rapport de couverture dans gitlab-ci.yml (au moment des **merge requests**)

```
script:
  - coverage run coverage_tests.py
  - coverage xml -o
  $CI_PROJECT_DIR/tests_artifacts/coverage.xml
artifacts:
  reports:
    cobertura: "$CI_PROJECT_DIR/tests_artifacts/coverage.xml"
only:
  - merge_requests
```

Cl.yml : Couverture de code

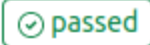

- Affichages de la couverture dans Gitlab CI

➤ Dans la page d'exécution du job

units_mr [Retry](#)

Duration: 25 seconds
Timeout: 1h (from project) ⓘ
Runner: matt-VirtualBox (#3)
Coverage: 89%

➤ Dans la table des jobs

Status	Job	Pipeline	Stage	Name	Timing	Coverage
	#214 ↕ refs/merge-... ↗ d3792281	#120 by 	tests	units_mr	⌚ 00:00:25 📅 5 minutes ago	89.0%

➤ Les lignes non couvertes sont indiquées dans le commit de la merge request

```
15 15
16 16     def testOverdraft2(self):
17 17         self.account.deposit(200)
18 18         self.assertFalse(self.account.overdraft)
19 19
20 20 +     def testDisplayBalance(self):
21 21 +         self.assertEqual("500.00 €", self.account.displayBalance())
22 22 +
20 23
21 24     if __name__ == "__main__":
22 25         unittest.main()
```


Cl.yml : tests end 2 end (1/5)

- Le serveur selenium
 - Solution java pour **piloter des navigateurs internet** au moyen de drivers ad hoc
 - Un module client selenium existe pour python3, permettant la **simulation de l'activité** des utilisateurs sur des sites
 - Le client pilote des drivers de navigateurs (Firefox, Chrome, Edge...) installés sur la machine ou sur un serveur selenium distant
 - **Installation** : `wget -q -O selenium.jar \`
<https://selenium-release.storage.googleapis.com/3.14/selenium-server-standalone-3.14.0.jar>
 - **Exécution** : `java -jar selenium.jar (port 4444)`

Cl.yml : tests end 2 end (2/5)

- Le client python selenium
 - Peut s'exécuter par le module « unittest »
 - Permet de simuler la navigation internet (HTTP, AJAX), les évènements (souris, clavier)
 - Permet de tester les navigateurs les plus courants (chrome, firefox...)
 - Permet de configurer les navigateurs (mode sans fenêtres, plugins...)

Ci.yml : tests end 2 end (3/5)

- Exemple de test e2e python avec selenium

```
import unittest
from selenium import webdriver
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities
from selenium.webdriver.firefox.options import Options
from selenium.webdriver.common.keys import Keys

class PythonOrgSearch(unittest.TestCase):

    def setUp(self):
        options = Options()
        options.headless = True
        self.driver = webdriver.Remote(
            options=options,
            command_executor="http://localhost:4444/wd/hub",
            desired_capabilities=DesiredCapabilities.FIREFOX)

    def test_search_in_python_org(self):
        driver = self.driver
        driver.get("http://www.python.org")
        elem = driver.find_element_by_name("q")
        elem.send_keys("pycon")
        elem.send_keys(Keys.RETURN)
        assert "No results found." not in driver.page_source

    def tearDown(self):
        self.driver.close()
```

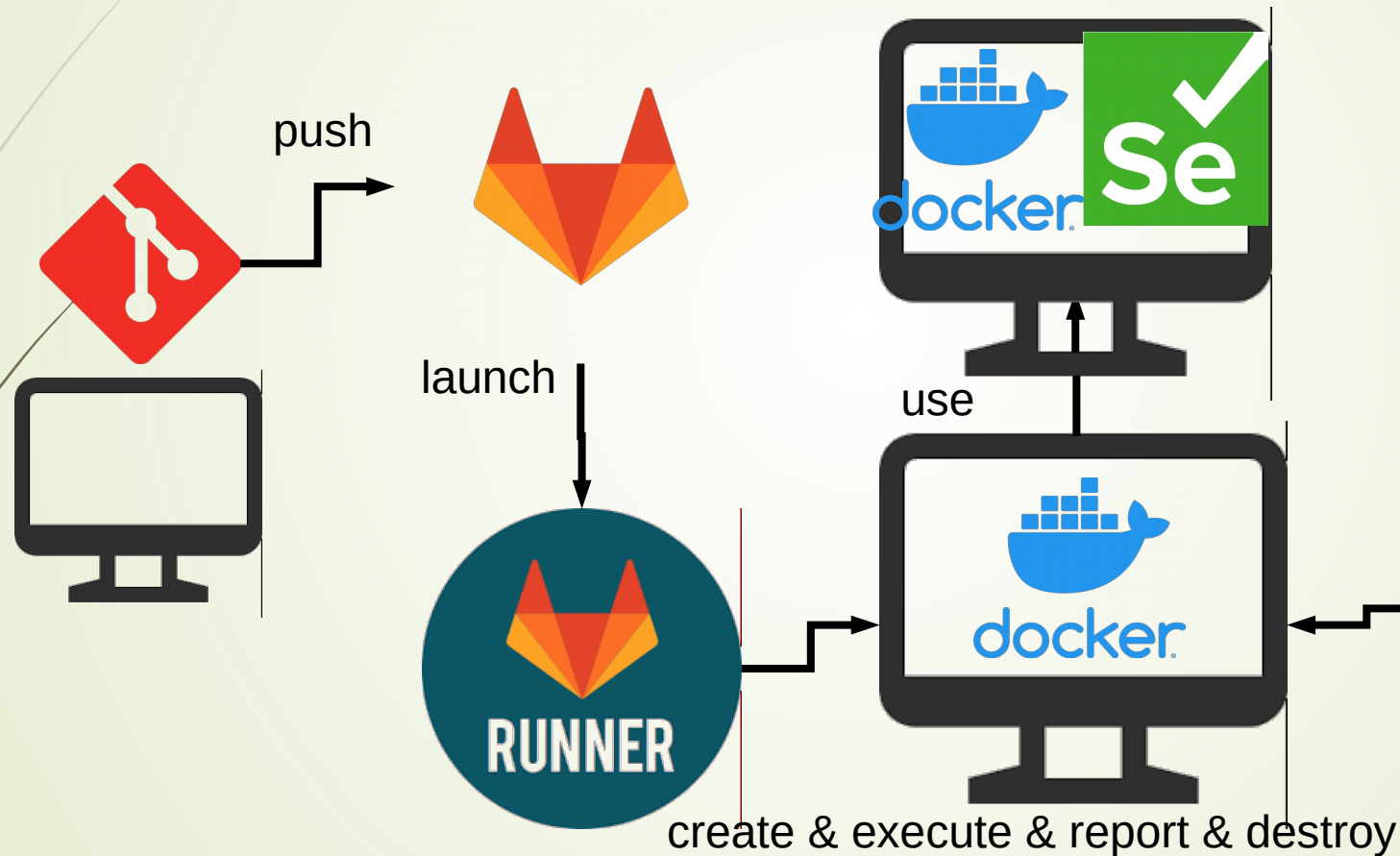
CI.yml : tests end 2 end (4/5)

- Intégration avec navigateur dans gitlab CI
 - Le conteneur en question doit contenir firefox ainsi que son driver pour que sélénium puisse les manipuler

```
before_script:  
  - apt update  
  - apt install -y firefox-esr  
  - mv $CI_PROJECT_DIR/geckodriver /usr/local/bin  
  - chmod +x /usr/local/bin/geckodriver  
  - pip3 install -r $CI_PROJECT_DIR/requirements.txt  
script:  
  - python3 e2e_firefox.py
```

SELENIUM: intégration

- Déclenchement d'un job e2e avec selenium



```
services:  
  - name: selenium...  
    alias: hostname  
before_script:  
  - ...  
  - pip3 install -r reqs.txt  
script:  
  - python3 e2e_remote.py
```


CI.yml : tests end 2 end (5/5)

- Intégration comme service dans gitlab CI
- l'étiquette **services** permet au runner d'adjoindre au conteneur exécutant le code des services réseau, (ici un conteneur selenium server pour firefox)

```
services:  
  - name: selenium/standalone-firefox:latest  
    alias: gitlab_selenium_server  
before_script:  
  - mv $CI_PROJECT_DIR/geckodriver /usr/local/bin  
  - chmod +x /usr/local/bin/geckodriver  
  - pip3 install -r $CI_PROJECT_DIR/requirements.txt  
script:  
  - python3 e2e_remote.py
```