TESTCONTAINERS

TESTCONTAINERS

TU NE DOUTERAS PLUS DE TES FONCTIONNALITÉS, JEUNE PYDAWAN·E

Luc Sorel-Giffo — jeudi 27 juin 2024 - 10h30 amphi A — BreizhCamp (Rennes)

@UI SUIS-JE ?

- tech lead Python chez Purecontrol `#techForGood`

- (OSS) outils doc-as-code :
  - py2puml
  - pydoctrace

@UI SUIS-JE ?

- tech lead Python chez Purecontrol `#techForGood`

- (OSS) outils doc-as-code :
  - py2puml
  - pydoctrace

- co-animateur Python Rennes

Figure 1. Meetup : www.meetup.com/fr-FR/python-rennes

Figure 2. Pour rejoindre le slack : join.slack.com/t/pythonrennes/shared_invite/zt-1yd4yioap-lBAngm3Q0jxAKLP6fYJR8w

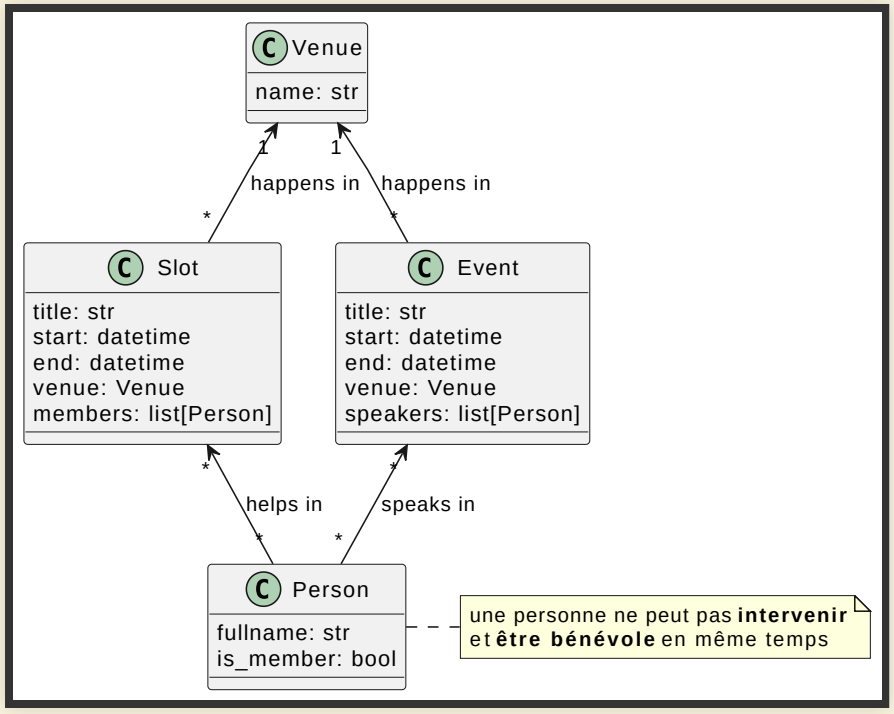
- @lucsorelgiffo@floss.social

PRÉAMBULE ⚠️

- testcontainers
- pytest : fixtures, markers ; FastAPI : TestClient
- Python : gestionnaire de contexte (with…), générateur (yield…), architecture

Démos : members-agenda (planning de bénévoles gérant les indisponibilités)

MEMBERS-AGENDA

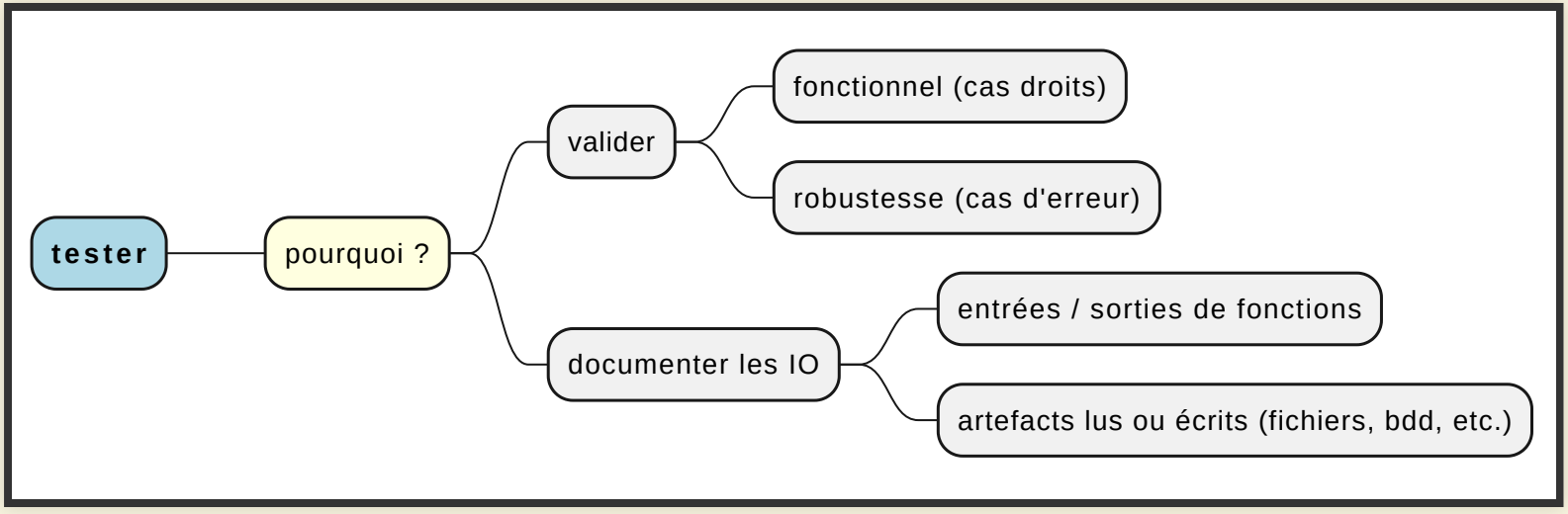


Environnement technique :

- server web : FastAPI
- base de données : MySQL (avec pymysql)
- framework de test : pytest

POURQUOI TESTER ?

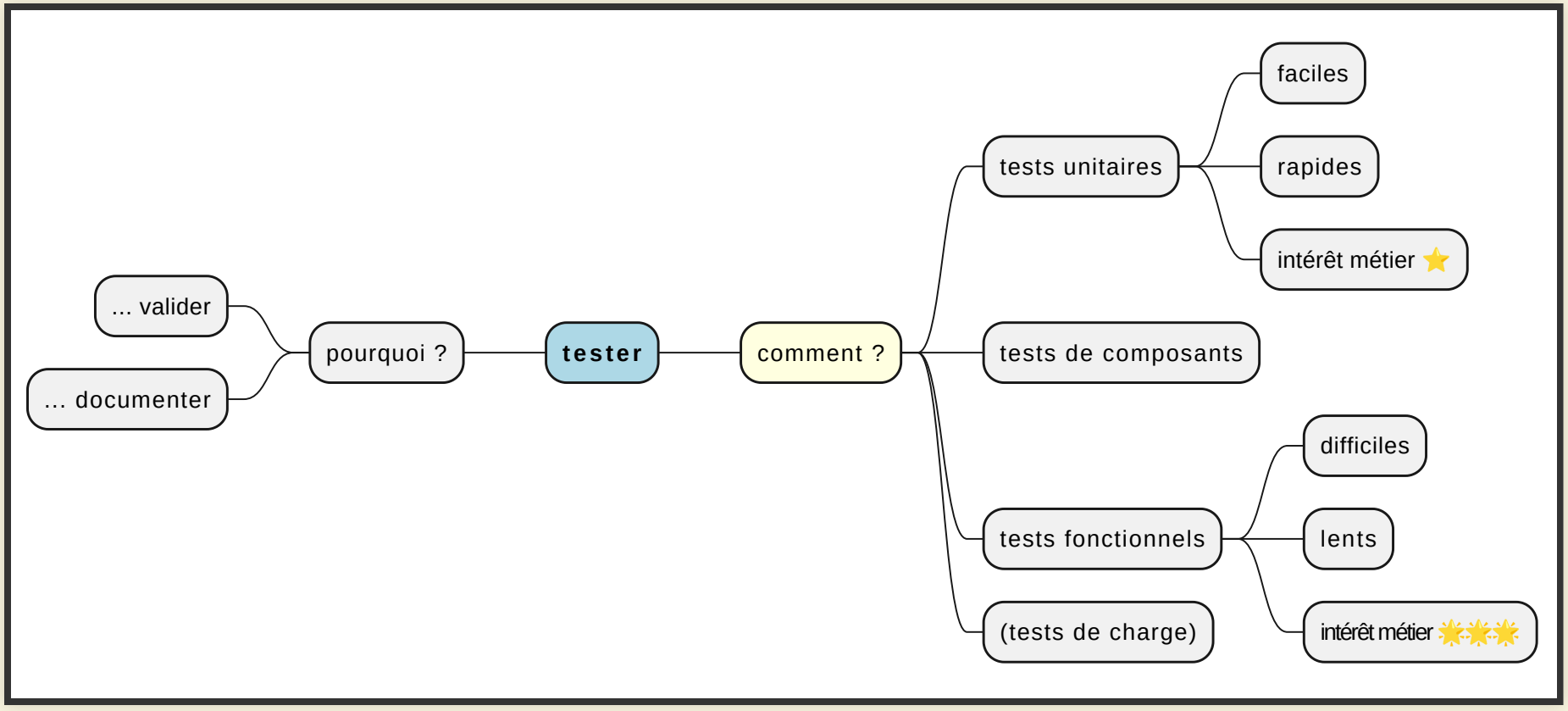
POURQUOI TESTER ?



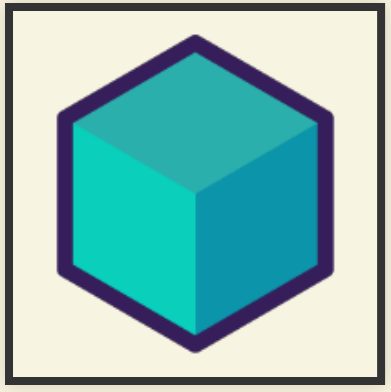
QUELS TYPES DE TEST ?

🤷 ?

@QUELS TYPES DE TEST ?



TESTCONTAINERS



A framework for providing throwaway, lightweight instances of databases, message brokers, web browsers, or just about anything that can run in a Docker container.

- multi-clients : python, java, go, etc.
- +50 services (bdd, brokers)

- github.com/testcontainers/testcontainers-python
- 1.4k ⭐, 27 releases (juin 2024)
- open-source (Apache 2.0)
- 112 contributeur·ices

INSTALLATION

```
pip install "testcontainers-python[mysql]"

poetry add --group dev "testcontainers-python[mysql]"
```

PRINCIPE DE FONCTIONNEMENT

1. démarrage du container "vide"
2. création du contexte initial du test
3. déroulé du test
4. assertions sur l’état final
5. arrêt et suppression du container

Services conteneurisés : testcontainers-python.readthedocs.io/en/latest/modules/index.html.

🧪 1 : LA "PROD"

```python
from os import getenv
from fastapi import FastAPI
from pymysql.connections import Connection, DictCursor

def get_connection() -> Connection:
    M_HOST = getenv('MYSQL_HOST')
    M_PORT = int(getenv('MYSQL_PORT'))
    M_USER = getenv('MEMBERS_AGENDA_USER')
    M_PWD = getenv('MEMBERS_AGENDA_PASSWORD')
    M_DB = getenv('MEMBERS_AGENDA_DATABASE')

    return Connection(
        host=M_HOST, port=M_PORT, user=M_USER, password=M_PWD, database=M_DB
    )

app = FastAPI()

@app.get('/venues')
def get_venues() -> list[dict]:
    get_connection() as connection:
        with connection.cursor(DictCursor) as cursor:
            cursor.execute('SELECT * FROM venues;')
            return cursor.fetchall()
        # -> clôture du curseur
    # -> clôture de la connexion
```

🧪 1 : LE TEST

```python
from os import environ
from fastapi.testclient import TestClient
from testcontainers.mysql import MySqlContainer

from members_agenda_api.__main__ import app, get_connection

def test_get_venues():
    with MySqlContainer() as container:
        environ["MYSQL_HOST"] = container.get_container_host_ip()
        environ["MYSQL_PORT"] = container.get_exposed_port(3306)
        environ["MEMBERS_AGENDA_USER"] = container.username
        environ["MEMBERS_AGENDA_PASSWORD"] = container.password
        environ["MEMBERS_AGENDA_DATABASE"] = container.dbname

        create_2_test_venues(get_connection())

        client = TestClient(app)
        response = client.get('/venues')

        assert response.status_code == 200
        venues = response.json()
        assert len(venues) == 2
        assert venues[1] == {
          'id': 2, 'name': 'Goodies', 'rank': 2, 'bg_color_hex': '2D8289'
        }
```

🧪 1 : LE TEST

🧪 1 : LE BILAN

🤷 ?

🧪 1 : LE BILAN

- ça marche
- c’est lent
- code d’initialisation du contexte
- surcharger des variables d’environnement
  - tests fragiles
  - les lire à chaque connexion est contre-intuitif
- code de test indenté dans le "with MySqlContainer()…"

MONKEYPATCH

- `fixture` : fonctionnalité ou données de test injectées par pytest
- `monkeypatch` : fixture permettant de modifier toute propriété d'un objet (le temps du cas de test)
- "*In Python, everything is an object*"

→ les définitions d'un module peuvent être modifiées à chaud

Doc & tutoriels : docs.pytest.org/en/latest/reference/reference.html#monkeypatch

MONKEYPATCH

Dans `test_get_venues.py`, quelle définition de module faut-il modifier ?

```
members_agenda_api/
 ├─ services/
 │   ├─ connection.py  # 🍌 def get_connection()
 │   ├─ dataservice.py # class DataService
 │   └─ __init__.py    # 🍌 import get_connection, DataService ; def get_data_service()
 ├─ api.py             # import get_data_service ; API_ROUTER = ...
 ├─ __main__.py        # import API_ROUTER ; app = ...
 ├─ ...
 ├─ tests/
 │   ├─ test_get_venues.py  # TestClient(app) ; 🐒 ?
 ...
```

2 MONKEYPATCH

```python
from fastapi.testclient import TestClient
from pymysql.connections import Connection
from testcontainers.mysql import MySqlContainer

from members_agenda_api.__main__ import app

from tests.members_agenda_api.test_1_get_venues_envvars import create_2_test_venues

def test_get_2_venues_mkp(monkeypatch):
    with MySqlContainer() as container:
        connection = Connection(
            host=container.get_container_host_ip(),
            port=int(container.get_exposed_port(3306)),
            user=container.username, password=container.password,
            database=container.dbname,
        )
        create_2_test_venues(connection)

        monkeypatch.setattr(
            'members_agenda_api.services.get_connection', lambda: connection
        )

        client = TestClient(app)
        response = client.get('/api/venues')
        assert response.status_code == 200
        ...
```

🧪 2 : MONKEYPATCH

🧪 2 : LE BILAN

🤷 ?

🧪 2 : LE BILAN

- ce qui est monkeypatché n'est pas testé
- les modifications faites par monkeypatch durent le temps du cas de test
- cumul des temps de démarrage des conteneurs 🐌

Outiller ses tests d'intégration

Besoins :

- une fixture injectable dans les cas de test
- propose une connexion au service conteneurisé
- propose des fonctionnalités d'initialisation
- **un** conteneur pour **tous** les tests

```python
from pathlib import Path
from typing import NamedTuple

from pymysql.connections import Connection

from tests.containers.sql_queries_parser import (
    execute_sql_queries
)

class SqlTestHelper(NamedTuple):
    connection: Connection

    def setup_with_sql_filepath(self, sql_filepath: Path):
        """
        Executes the queries in the given sql file
        against the database in the connection
        """
        with open(
            sql_filepath, encoding='utf8'
        ) as sql_file:
            execute_sql_queries(sql_file, self.connection)
```

TESTS/CONFTEST.PY

```python
from pymysql.connections import Connection
from pytest import fixture
from testcontainers.mysql import MySqlContainer
from tests.containers.sql_helper import SqlTestHelper

@fixture(scope="session")
def sql_test_helper() -> SqlTestHelper:
    db_name = 'members_agenda'

    # Docker container creation
    with MySqlContainer(
        image="mysql:8.0",
        dbname=db_name,
    ) as sql_container:
        # database connection (for direct use or monkeypatching in tests)
        connection = Connection(
            user = sql_container.username,
            password = sql_container.password,
            host = sql_container.get_container_host_ip(),
            port = int(sql_container.get_exposed_port(sql_container.port)),
            database = db_name,
        )

        yield SqlTestHelper(connection)

        # you could write post-tests code here
```

UTILISER LA FIXTURE

```python
from fastapi.testclient import TestClient

from members_agenda_api.__main__ import app
from members_agenda_api.services.dataservice import DataService

from tests.containers.sql_helper import SqlTestHelper
from tests.containers.sql_files import SQL_FILES_FOLDER


def test_api_get_venues_fixture(monkeypatch, sql_test_helper: SqlTestHelper):
    sql_test_helper.setup_with_sql_filepath(SQL_FILES_FOLDER / 'venues_samples.sql')

    monkeypatch.setattr(
        'members_agenda_api.api.get_data_service',
        lambda: DataService(sql_test_helper.connection)
    )

    client = TestClient(app)
    response = client.get('/api/venues')

    assert response.status_code == 200
    assert len(response.json()) == 2
```

3 FIXTURE

🧪 3 : LE BILAN

🤷 ?

🧪 3 : LE BILAN

- code de test plus court, désindenté
- contexte initialisé via un fichier .sql

🧪 4 TESTS PARAMÉTRÉS

"Qui affecter en bénévole en amphi A, de 10h15 et 12h30 ❓"

horaires	accueil$_{id:1}$	amphi A$_{id:4}$	amphi C$_{id:6}$	amphi D$_{id:7}$
10h15	slot$_{id:43}$: Alex$_{id:7}$	slot$_{id:44}$: ❓		slot$_{id:47}$:
12h30		"Testcontainers..."$_{id:22}$: Luc$_{id:79}$		"Manifeste..."$_{id:45}$: Cécilia$_{id:25}$
12h30			slot$_{id:51}$: Johanna$_{id:70}$	
13h30			"Virus..."$_{id:39}$: Nailya$_{id:95}$	

"Et que se passe-t-il si on affecte une personne indisponible ?" 🤔

🧪 4 TESTS PARAMÉTRÉS

🧪 4 : LE BILAN

🤷 ?

🧪 4 : LE BILAN

- temps de création d'un seul conteneur
- un seul code de test → plein de cas
- documentation des entrées-sorties
- documentation cas droits / cas d'erreurs

LES BONUS

TRAÇAGE DOCUMENTAIRE

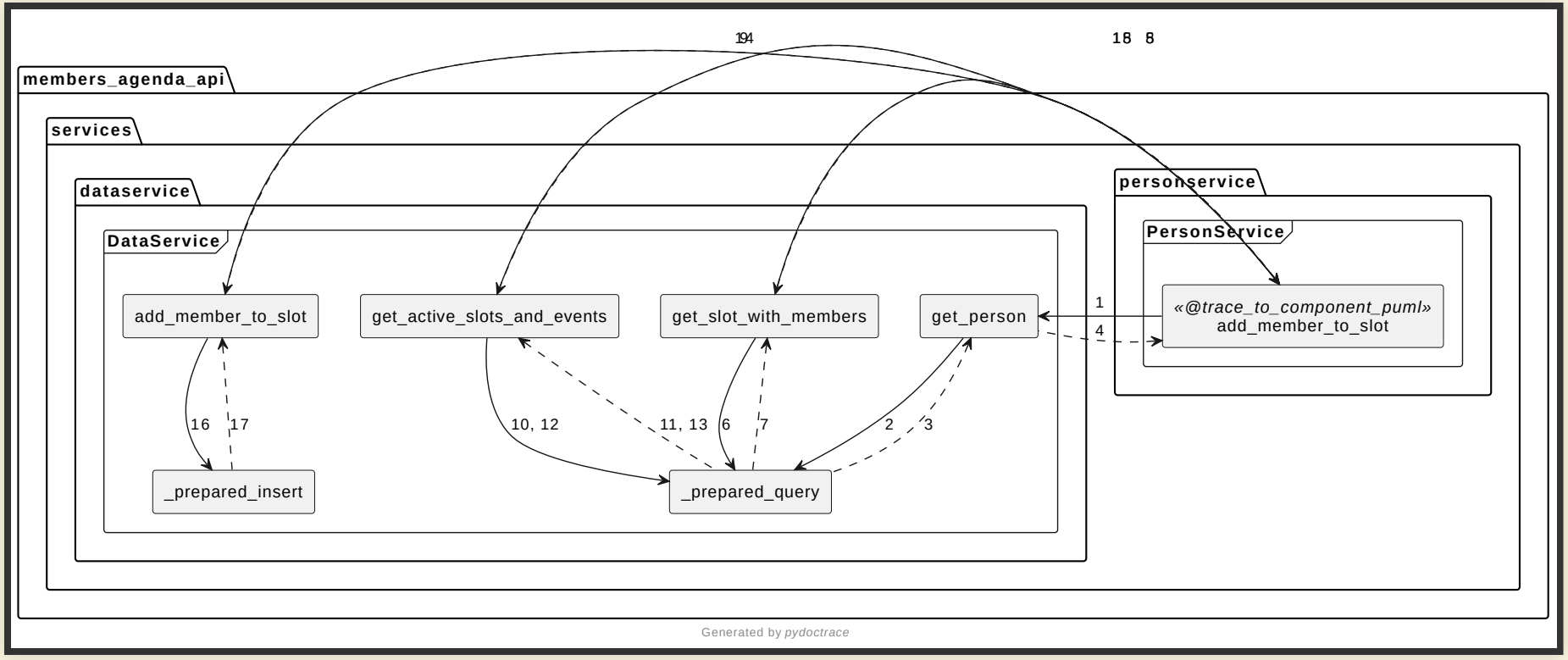
pypi.org/project/pydoctrace/ : créer des diagrammes (séquence, composants) d'exécution d'une fonction via un décorateur.

Voir youtu.be/iRtr9NJJ6Cw : Doc-tracing : fouiller une base de code fossile grâce au traçage d'exécution (BreizhCamp 2023)

UTILISATION CLASSIQUE

```python
from pydoctrace.doctrace import trace_to_component_puml
from pydoctrace.callfilter.presets import EXCLUDE_STDLIB_PRESET, Preset
EXCLUDE_LIBS_PRESET = Preset(...)

class PersonService:
    @trace_to_component_puml(filter_presets=[EXCLUDE_STDLIB_PRESET, EXCLUDE_LIBS_PRESET]
    def add_member_to_slot(self, member_id: int, slot_id: int) -> int:
        ...
```



🧪 5 : DOC-TRACING

🧪 5 : LE BILAN

🤷?

🧪 5 : LE BILAN

- documentation autogénérée
- discuter fonctionnalités avec votre (c6)PO
- discuter architecture & implémentation avec l'équipe

ÇA VOUS MARK 1

Pour labelliser des tests à dé·sélectionner :

1. déclarer le label dans `pyproject.toml`
2. dé·sélectionner les tests avec `pytest -m …`

```
[tool.pytest.ini_options]
addopts = "--strict-markers"
markers = [
    "containers: integration tests requiring docker test containers",
]
```

```
pytest -m "containers"
pytest -m "not containers"
```

ÇA VOUS MARK 2

Désélection conditionnelle (dans `tests/conftest.py`)

```python
from subprocess import run
from pytest import mark

def _is_docker_available() -> bool:
    is_docker_installed_process = run(("which", "docker"), capture_output=True)
    if is_docker_installed_process.returncode != 0:
        return False

    is_docker_running_process = run(("docker", "ps"), capture_output=True)
    return is_docker_running_process.returncode == 0

# déclaration programmatique du marker
mark.skipifnodocker = mark.skipif(
    not _is_docker_available(), reason="Requires docker to spin a container"
)
```

```python
@mark.containers
@mark.skipifnodocker
def test_dataservice_get_venues(monkeypatch, sql_test_helper: SqlTestHelper):
    sql_test_helper.setup_with_sql_filepath(SQL_FILES_FOLDER / 'venues_samples.sql')
    ...
```

CONTRIBUER

Voir github.com/testcontainers/testcontainers-python/pull/413 :

Ajouter un module :

- image Docker par défaut
- méthodes :
  - `start()`
  - `_health_check()` : informe du démarrage et de la disponibilité du conteneur
- écrire des tests impliquant le conteneur

💙💛 BONNES PRATIQUES

Utilisation astucieuse de concepts avancés de Python : générateurs, gestionnaire de contexte d’exécution

💙💛 BONNES PRATIQUES

Utilisation astucieuse de concepts avancés de Python : générateurs, gestionnaire de contexte d'exécution

- localiser l'endroit où la connexion à la base est faite → facile à monkeypatcher / mocker

💙💛 BONNES PRATIQUES

Utilisation astucieuse de concepts avancés de Python : générateurs, gestionnaire de contexte d'exécution

- localiser l'endroit où la connexion à la base est faite → facile à monkeypatcher / mocker
- regrouper les interactions "natives" au service dans une classe (ou dans un module)
  - tester la classe avec testcontainers
  - mocker la classe dans les tests qui l'utilisent indirectement

💙💛 BONNES PRATIQUES

Utilisation astucieuse de concepts avancés de Python : générateurs, gestionnaire de contexte d'exécution

- localiser l'endroit où la connexion à la base est faite → facile à monkeypatcher / mocker
- regrouper les interactions "natives" au service dans une classe (ou dans un module)
  - tester la classe avec testcontainers
  - mocker la classe dans les tests qui l'utilisent indirectement
- nettoyer le conteneur avant la création du contexte

💙💛 BONNES PRATIQUES

Utilisation astucieuse de concepts avancés de Python : générateurs, gestionnaire de contexte d'exécution

- localiser l'endroit où la connexion à la base est faite → facile à monkeypatcher / mocker
- regrouper les interactions "natives" au service dans une classe (ou dans un module)
  - tester la classe avec testcontainers
  - mocker la classe dans les tests qui l'utilisent indirectement
- nettoyer le conteneur avant la création du contexte
- TestClient pour tester une API sans lancer le serveur web (voir testing FastAPI)

💙💛 BONNES PRATIQUES

Utilisation astucieuse de concepts avancés de Python : générateurs, gestionnaire de contexte d'exécution

- localiser l'endroit où la connexion à la base est faite → facile à monkeypatcher / mocker
- regrouper les interactions "natives" au service dans une classe (ou dans un module)
  - tester la classe avec testcontainers
  - mocker la classe dans les tests qui l'utilisent indirectement
- nettoyer le conteneur avant la création du contexte
- TestClient pour tester une API sans lancer le serveur web (voir testing FastAPI)
- tester les cas droits et d'erreur

💙💛 BONNES PRATIQUES

Utilisation astucieuse de concepts avancés de Python : générateurs, gestionnaire de contexte d'exécution

- localiser l'endroit où la connexion à la base est faite → facile à monkeypatcher / mocker
- regrouper les interactions "natives" au service dans une classe (ou dans un module)
  - tester la classe avec testcontainers
  - mocker la classe dans les tests qui l'utilisent indirectement
- nettoyer le conteneur avant la création du contexte
- TestClient pour tester une API sans lancer le serveur web (voir testing FastAPI)
- tester les cas droits et d'erreur
- rappeler la valeur documentaire des tests

MERCI 🙏

DES QUESTIONS ?

Présentation à retrouver sur github.com/lucsorel/conferences/{...}/breizhcamp-2024.06.27-testcontainers-pytest 📑

Figure 3. Vos retours sur openfeedback.io/LyIREj0UbxmZ6vcFmxmN/2024-06-27/670894