diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Fixtures de test avec Pytest (FastAPI / SQLModel)

Les **fixtures** sont des fonctions spéciales de Pytest qui permettent de partager une configuration entre plusieurs tests : base de données, client HTTP, données initiales, etc. Elles rendent les tests plus **lisibles**, **isolés**, et **faciles à maintenir**.

Sommaire

Pourquoi utiliser des fixtures ?

- ☑ **Réutilisation** : configuration commune utilisée dans plusieurs tests
- ☑ **Isolation** : chaque test utilise une base vide ou initialisée à l'identique
- ☑ **Nettoyage** : la base ou la session sont automatiquement libérées à la fin
- ☑ **Clarté** : les tests ne s'encombrent pas de logique d'initialisation

Workflow typique d'un test FastAPI avec fixtures

Étape	Description
Création base test	Une base SQLite en mémoire est créée
Création tables	Les modèles SQLModel sont appliqués
Insertion d'un objet par défaut	Un ou plusieurs objets sont créés dans la base
Création session SQLAlchemy	Fournie aux tests via la dépendance `get_session()`
Client HTTP `TestClient`	Crée avec `app.dependency_overrides` pour injecter la session de test
Exécution du test	Chaque test utilise cette configuration

Code : création des fixtures de base avec données par défaut

```
import pytest
from sqlmodel import create_engine, SQLModel, Session
from fastapi.testclient import TestClient
from main import app, get_session
from models import Item  # Remplace par le bon chemin si nécessaire

@pytest.fixture
def test_session():
    """
    Crée une base SQLite en mémoire, initialise les tables,
    et insère un item de test par défaut.
    """
    engine = create_engine("sqlite:///:memory:", connect_args=
{"check_same_thread": False})
    SQLModel.metadata.create_all(engine)

    with Session(engine) as session:
        # ➕ OPTIONNEL : Insertion d'un objet par défaut
        default_item = Item(name="item test", description="créé via fixture")
        session.add(default_item)
        session.commit()

        yield session  # Fournit la session au test

@pytest.fixture
def client(test_session):
    """
    Crée un client FastAPI qui utilise la session de test en override.
    """
    def override_get_session():
        yield test_session

    # Ecrase la connexion à l'ancienne base de données par la nouvelle
    app.dependency_overrides[get_session] = override_get_session

    with TestClient(app) as test_client:
        yield test_client

    app.dependency_overrides.clear()
```

Exemples d'utilisation des fixtures dans les tests

Test de la page d'accueil

```
def test_home(client):
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Bienvenue"}
```

Test de création d'un objet en base

```
def test_create_item(client):
    payload = {"name": "item POST", "description": "créé par le test"}
    response = client.post("/items/", json=payload)

    assert response.status_code == 201
    data = response.json()
    assert data["name"] == payload["name"]
    assert data["description"] == payload["description"]
```

Test de lecture de l'objet créé automatiquement

```
def test_read_default_item(client):
    response = client.get("/items/1")
    assert response.status_code == 200
    assert response.json()["name"] == "item test"
    assert response.json()["description"] == "créé via fixture"
```

> 💡 Ce test **ne crée rien**, il profite simplement de l'objet pré-inséré dans la fixture.

À retenir

- Les **fixtures permettent de factoriser** la configuration des tests (base, client, données).
- La **base est en mémoire**, donc rapide et jetable.
- Les **overrides de dépendances** permettent d'utiliser `TestClient` avec une session injectée.
- On peut **pré-remplir la base** avec une ou plusieurs entrées par défaut pour simplifier les tests de lecture.
- Chaque test est **rapide, isolé et prévisible**, ce qui facilite la montée en complexité.