



« Votre passeport pour l'emploi numérique »

Utilisation de `httpx` et `TestClient` avec Pytest

FastAPI fournit un client de test intégré basé sur `httpx`, permettant de simuler des requêtes HTTP **sans démarrer un vrai serveur**. Cela facilite l'écriture de tests efficaces, simples et rapides.

Sommaire

- [TestClient : principe de base](#)
- [Tester une route simple](#)
- [Tester les erreurs de validation](#)
- [Test d'intégration avec SQLModel \(sans fixture\)](#)
- [Override d'une dépendance avec une fausse base](#)
- [Résumé à retenir](#)

TestClient : principe de base

```
from fastapi.testclient import TestClient
from main import app # ou votre fichier d'application

client = TestClient(app)

def test_homepage():
    response = client.get("/")
    assert response.status_code == 200
```

 **TestClient** fonctionne comme un navigateur ou Postman, mais dans vos tests automatisés.

Tester une route simple

```
def test_hello():
    response = client.get("/hello")
    assert response.status_code == 200
    assert response.json() == {"message": "Hello, World!"}
```

 Ce test vérifie que la route répond et que le format de la réponse est correct.

Tester les erreurs de validation

FastAPI utilise Pydantic pour valider les entrées. En cas de données incorrectes, une erreur 422 est automatiquement renvoyée.

```
def test_create_user_invalid():
    payload = {"email": "pas-un-email"} # Format invalide
    response = client.post("/users", json=payload)
    assert response.status_code == 422
```

 Très utile pour tester la robustesse de votre API face à des erreurs de saisie.

Test d'intégration avec SQLModel (sans fixture)

On peut tester une vraie interaction API avec une base de test locale, sans utiliser de fixture `pytest`.

Code à tester

```
from fastapi.testclient import TestClient
from sqlmodel import SQLModel, Session, create_engine, select
from fastapi import FastAPI, Depends

# ◇ Définition du modèle
class Client(SQLModel, table=True):
    id: int | None = None
    nom: str
    email: str

# ◇ Création de l'application FastAPI
app = FastAPI()

# ◇ Dépendance DB
DATABASE_URL = "sqlite:///memory:"
engine = create_engine(DATABASE_URL, connect_args={"check_same_thread": False})

def get_db():
    with Session(engine) as session:
        yield session

# ◇ Route de création
@app.post("/clients", status_code=201)
def create_client(client: Client, db: Session = Depends(get_db)):
    db.add(client)
    db.commit()
    db.refresh(client)
    return client

# ◇ Route de lecture
@app.get("/clients/{client_id}")
def get_client(client_id: int, db: Session = Depends(get_db)):
    return db.exec(select(Client).where(Client.id == client_id)).first()
```

Test de l'API avec `TestClient`

```
# ◇ Le test complet
def test_create_and_read_client():
    SQLModel.metadata.create_all(engine)

    # 🖌 Surcharge de la dépendance
    def override_get_db():
        with Session(engine) as session:
            yield session

    app.dependency_overrides[get_db] = override_get_db
    client = TestClient(app)

    # ✅ Création d'un client
    data = {"nom": "Alice", "email": "alice@example.com"}
    res = client.post("/clients", json=data)
    assert res.status_code == 201
    assert res.json()["nom"] == "Alice"

    # 🔎 Lecture du client
    res = client.get("/clients/1")
    assert res.status_code == 200
    assert res.json()["email"] == "alice@example.com"

    app.dependency_overrides.clear()
```

💡 Ce test crée une base temporaire et effectue un scénario "POST → GET" sur la route.

Résumé à retenir

- `TestClient` permet de tester vos routes FastAPI sans lancer de serveur.
- `pytest` est le framework recommandé pour ces tests.
- Utilisez `app.dependency_overrides` pour remplacer vos dépendances pendant les tests.
- `SQLModel` rend les tests avec base SQLite en mémoire plus lisibles et faciles à maintenir.
- Les tests simples n'ont pas besoin de fixture ; on verra plus tard comment les utiliser pour mutualiser du code.