







Atelier 1 : Tests Unitaires Backend & Frontend

Durée : 3-4 heures **Objectif** : Apprendre les tests unitaires avec Python (pytest) et TypeScript (Vitest)



Vue d'Ensemble

Dans cet atelier, vous allez :

-  Tester le **backend Python** avec pytest (FastAPI)
-  Tester le **frontend TypeScript** avec Vitest (React)
-  Comprendre le stockage en mémoire (préparation pour Atelier 3)
-  Configurer **GitHub Actions** pour l'intégration continue
-  Lancer l'application en local (frontend + backend)

Important : L'application est déjà construite. Vous allez apprendre à la tester et à garantir sa qualité !

Phase 1 : Installation & Configuration (30 min)

Étape 1.1 : Forker le Dépôt

1. Allez sur `https://github.com/umons/edl-starter`
2. Cliquez sur "**Fork**"
3. Clonez votre fork :

```
git clone https://github.com/VOTRE_NOM/edl-starter
cd edl-starter
```

Étape 1.2 : Installer UV

macOS/Linux :

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Windows :

```
powershell -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Vérifiez :

```
uv --version
```

Étape 1.3 : Installer les Dépendances

```
cd backend
uv venv
source .venv/bin/activate # Windows: .venv\Scripts\activate
uv sync
```

Qu'est-ce que ça fait ?

- `uv venv` → Crée un environnement virtuel isolé
- `source .venv/bin/activate` → Active l'environnement
- `uv sync` → Installe toutes les dépendances depuis `pyproject.toml`

Phase 2 : Explorer l'Application (15 min)

Étape 2.1 : Lancer le Serveur

```
uv run uvicorn src.app:app --reload
```

Étape 2.2 : Tester dans le Navigateur

Visitez :

- **API** : <http://localhost:8000>
- **Documentation interactive** : <http://localhost:8000/docs>
- **Santé** : <http://localhost:8000/health>

Étape 2.3 : Tester avec Swagger UI

1. Allez sur <http://localhost:8000/docs>
2. Cliquez sur **POST /tasks**
3. Cliquez sur **"Try it out"**
4. Entrez :

```
{  
  "title": "Ma première tâche",  
  "description": "Apprendre FastAPI"  
}
```

5. Cliquez sur **"Execute"**
6. Vous devriez voir un code `201 Created`

Étape 2.4 : Explorer le Code

Ouvrez `backend/src/app.py` dans votre éditeur :

- **Lignes 27-36** : Énumérations (TaskStatus, TaskPriority)
- **Lignes 39-68** : Modèles Pydantic
- **Lignes 76-77** : Stockage en mémoire (dictionnaire simple)
- **Lignes 180-205** : Endpoint pour créer une tâche
- **Lignes 144-160** : Endpoint pour lister les tâches

Important : Ce backend utilise un **stockage en mémoire** (un simple dictionnaire Python) pour Atelier 1 & 2. Vous apprendrez à utiliser PostgreSQL dans l'Atelier 3.

Phase 3 : Comprendre les Tests (20 min)

Étape 3.1 : Explorer les Fichiers de Test

Ouvrez ces fichiers :

- `backend/tests/conftest.py` → Fixtures de test
- `backend/tests/test_api.py` → Tests

Étape 3.2 : Qu'est-ce qu'une Fixture ?

Dans `conftest.py`, regardez :

```
@pytest.fixture(autouse=True)
def clean_tasks():
    """Nettoie les tâches avant et après chaque test"""
    clear_tasks()
    yield
    clear_tasks()

@pytest.fixture
def client():
    """Fournit un client HTTP de test"""
    with TestClient(app) as test_client:
        yield test_client
```

Pourquoi c'est utile ?

- `clean_tasks` : Nettoie automatiquement le stockage en mémoire avant chaque test
- `client` : Vous n'avez pas à créer un client dans chaque test
- pytest les injecte automatiquement quand vous écrivez `def test_xxx(client):`

Étape 3.3 : Lancer les Tests Existants

```
uv run pytest -v
```

Vous devriez voir :

```
tests/test_api.py::test_root_endpoint PASSED
tests/test_api.py::test_health_check PASSED
tests/test_api.py::test_create_task PASSED
tests/test_api.py::test_list_tasks PASSED
tests/test_api.py::test_get_task_by_id PASSED
... (19 tests au total)

===== 19 passed in 0.45s =====
```

Étape 3.4 : Comprendre un Test

Regardez `test_create_task` dans `test_api.py` :

```
def test_create_task(client):
    # ARRANGE : Préparer les données
    new_task = {
        "title": "Acheter des courses",
        "description": "Lait, œufs, pain"
    }

    # ACT : Faire la requête
    response = client.post("/tasks", json=new_task)

    # ASSERT : Vérifier
    assert response.status_code == 201
    assert response.json()["title"] == "Acheter des courses"
```

Pattern Arrange-Act-Assert :

1. **Arrange** → Préparer
 2. **Act** → Agir
 3. **Assert** → Vérifier
-

Phase 4 : Écrire Vos Tests (45 min)

🎯 Exercice 1 : Test DELETE (15 min - À faire ensemble)

Objectif : Écrire un test qui supprime une tâche

Étapes :

1. Créer une tâche
2. Obtenir son ID
3. La supprimer avec `client.delete()`
4. Vérifier qu'elle a disparu (404)

Travaillons ensemble :

```
def test_delete_task(client):  
    # 1. Créer une tâche  
    create_response = client.post("/tasks", json={"title": "À supprimer"})  
    task_id = create_response.json()["id"]  
  
    # 2. Supprimer la tâche  
    delete_response = client.delete(f"/tasks/{task_id}")  
    assert delete_response.status_code == 204  
  
    # 3. Vérifier qu'elle a disparu  
    get_response = client.get(f"/tasks/{task_id}")  
    assert get_response.status_code == 404
```

Points clés :

- ⚠ N'oubliez pas le `f` dans `f"/tasks/{task_id}"`
- ⚠ DELETE retourne 204, pas 200
- ⚠ Il faut VÉRIFIER que la tâche a bien disparu

👉 Exercice 2 : Test UPDATE (10 min - À faire seul)

Complétez `test_update_task` dans `test_api.py` :

Objectif : Mettre à jour le titre d'une tâche

Astuce : C'est similaire au test DELETE, mais avec `client.put()`

👉 Exercice 3 : Test Validation Titre Vide (5 min)

Complétez `test_create_task_empty_title` :

Objectif : Vérifier qu'un titre vide est refusé

```
def test_create_task_empty_title(client):  
    response = client.post("/tasks", json={"title": ""})  
    assert response.status_code == 422 # Erreur de validation
```

👉 Exercice 4 : Test Titre Manquant (5 min)

Complétez `test_create_task_no_title` :

Objectif : Vérifier qu'une tâche sans titre est refusée

👉 Exercice 5 : Test 404 (5 min)

Complétez `test_get_nonexistent_task` :

Objectif : Vérifier qu'obtenir une tâche inexistante retourne 404

🎁 Exercices Bonus (Si vous avez le temps)

- **Bonus 1** : Tester le filtrage par statut
- **Bonus 2** : Tester la mise à jour partielle
- **Bonus 3** : Tester le cycle de vie complet

Phase 5 : Couverture de Code (15 min)

Étape 5.1 : Lancer les Tests avec Couverture

```
uv run pytest --cov
```

Résultat :

```
----- coverage: platform darwin, python 3.12.7 -----
Name                               Stmts  Miss  Cover
-----
src/app.py                          156     6    96%
-----
TOTAL                              156     6    96%
```

Note : La couverture est très élevée (96%) car le backend est simple avec stockage en mémoire. Dans l'Atelier 3, vous ajouterez une base de données PostgreSQL.

Étape 5.2 : Générer un Rapport HTML

```
uv run pytest --cov --cov-report=html
```

Ouvrir le rapport :

```
open htmlcov/index.html # macOS
start htmlcov/index.html # Windows
```

Questions à se poser :

- Quelles lignes ne sont pas testées ?
- Est-ce important de les tester ?
- Le backend utilise un stockage en mémoire - simple et parfait pour l'apprentissage !
- Dans l'Atelier 3, vous migrerez vers PostgreSQL pour la persistance des données

Phase 6 : Tests Frontend (30 min)

Étape 6.1 : Comprendre le Frontend

Le frontend est une application **React + TypeScript** simple qui communique avec le backend.

Structure :


```

frontend/
├─ src/
|   ├─ App.tsx           # Composant principal
|   ├─ App.css           # Styles simples
|   ├─ api/
|   |   ├─ api.ts        # Client API
|   |   └─ api.test.ts   # Tests API ← ON TESTE ÇA
|   └─ components/
|       ├─ SimpleTaskList.tsx
|       └─ TaskForm.tsx
└─ package.json

```

Important : On teste **uniquement l'API** (pas les composants React) pour rester simple.

Étape 6.2 : Lancer les Tests Frontend

```

cd frontend
npm test

```

Vous devriez voir :

```

✓ src/api/api.test.ts (3 tests) 4ms
  ✓ fetches tasks from the backend
  ✓ creates a new task
  ✓ throws error when API fails

```

```

Test Files  1 passed (1)
Tests      3 passed (3)

```

Étape 6.3 : Analyser les Tests

Ouvrez `frontend/src/api/api.test.ts` :

```

describe('API Module', () => {
  it('fetches tasks from the backend', async () => {
    // Mock fetch pour simuler la réponse
    (globalThis as any).fetch = vi.fn(() =>
      Promise.resolve({

```

```

    ok: true,
    json: () => Promise.resolve([
      { id: 1, title: 'Test Task', status: 'todo' }
    ]),
  })
);

const tasks = await api.getTasks();
expect(tasks).toHaveLength(1);
expect(tasks[0].title).toBe('Test Task');
});
});

```

Concepts clés :

- **Mocking** : On simule `fetch()` pour ne pas appeler le vrai backend
- **async/await** : Tests asynchrones
- **expect()** : Assertions Vitest (similaire à pytest)

Étape 6.4 : Couverture Frontend

```
npm run test:coverage
```

Résultat :

File		% Stmts		% Branch		% Funcs		% Lines	
-----		-----		-----		-----		-----	
api.ts		68.42		55.55		50		68.42	

Note : On teste uniquement l'API (pas les composants React) pour Atelier 1. C'est suffisant !

Étape 6.5 : Lancer l'Application Complète

Terminal 1 - Backend :

```

cd backend
uv run uvicorn src.app:app --reload

```

Terminal 2 - Frontend :

```
cd frontend  
npm run dev
```

Ouvrir : <http://localhost:5173>

Vous pouvez créer/modifier/supprimer des tâches ! 🎨

⚠ **Important** : Les données sont en mémoire. Si vous redémarrez le backend, tout est perdu (c'est normal pour Atelier 1-2).

Phase 7 : GitHub Actions (40 min)

Étape 7.1 : Créer le Fichier Workflow

```
touch .github/workflows/test.yml
```

Étape 7.2 : Écrire le Workflow

Ouvrez `.github/workflows/test.yml` et ajoutez :

```
name: Tests Backend  
  
on:  
  push:  
    branches: [ main ]  
  pull_request:  
    branches: [ main ]  
  
jobs:  
  test:  
    runs-on: ubuntu-latest  
  
    steps:  
    - name: Récupérer le code  
      uses: actions/checkout@v4
```

```
- name: Installer Python
  uses: actions/setup-python@v5
  with:
    python-version: '3.11'

- name: Installer UV
  run: |
    curl -LsSf https://astral.sh/uv/install.sh | sh
    echo "$HOME/.local/bin" >> $GITHUB_PATH

- name: Installer les dépendances
  run: |
    cd backend
    uv venv
    uv sync

- name: Lancer les tests
  run: |
    cd backend
    uv run pytest -v --cov

- name: Vérifier la couverture
  run: |
    cd backend
    uv run pytest --cov --cov-fail-under=90
```

Étape 7.3 : Comprendre le Workflow

Déclencheurs (`on`) :

- Se lance quand vous poussez sur `main`
- Se lance sur chaque pull request

Étapes (`steps`) :


1. Récupérer le code
2. Installer Python 3.11
3. Installer UV
4. Installer les dépendances

5. Lancer les tests
6. Vérifier que la couverture est $\geq 90\%$

Étape 7.4 : Pousser sur GitHub

```
git add .  
git commit -m "Ajout des tests et du workflow CI/CD"  
git push origin main
```

Étape 7.5 : Vérifier sur GitHub

1. Allez sur votre dépôt GitHub
2. Cliquez sur l'onglet **"Actions"**
3. Vous verrez votre workflow en cours d'exécution
4. Attendez la coche verte 

Si ça échoue :

- Cliquez sur le workflow rouge
- Regardez quelle étape a échoué
- Lisez le message d'erreur
- Corrigez et poussez à nouveau

Phase 8 : Vérification Finale (15 min)

Liste de Contrôle

Vérifiez que vous avez :

Backend :


- ☐ UV installé (`uv --version` fonctionne)
- ☐ Backend qui tourne localement (<http://localhost:8000>)
- ☐ Tous les tests backend qui passent (19 tests)
- ☐ Compréhension du stockage en mémoire (dictionnaire Python)

- ☐ Couverture backend > 90% (actuellement 96%)

Frontend :




- ☐ Frontend qui tourne localement (<http://localhost:5173>)
- ☐ Tous les tests frontend qui passent (3 tests API)
- ☐ Compréhension du mocking avec Vitest
- ☐ Application complète fonctionnelle (créer/modifier/supprimer des tâches)

CI/CD :





- ☐ Fichier `.github/workflows/test.yml` créé
- ☐ Tests qui passent sur GitHub 

Ce que Vous Avez Appris

UV :

-  Installation et configuration
-  `uv venv` et `uv sync`
-  Gestion moderne des dépendances



pytest :

-  Structure d'un test (Arrange-Act-Assert)
-  Fixtures (`client` , `reset_storage`)
-  Lancer des tests
-  Couverture de code

HTTP Testing :

-  GET, POST, PUT, DELETE
-  Codes de statut (200, 201, 204, 404, 422)
-  Validation des données

GitHub Actions :

-  Créer un workflow
-  Tests automatisés

-  Intégration continue (CI)
-

Problèmes Courants

"Module not found"

→ Activez l'environnement virtuel : `source .venv/bin/activate`

"No module named 'src'"

→ Vous devez être dans `backend/` : `cd backend`

Tests qui échouent

→ Lancez un seul test : `uv run pytest tests/test_api.py::test_create_task -v -s`

Workflow GitHub qui échoue

→ Vérifiez que vous avez bien `cd backend` avant chaque commande

Ressources

- [Documentation FastAPI](#)
- [Documentation pytest](#)
- [Documentation UV](#)