Guide de Qualité pour le Projet Fédérateur

Introduction

Ce document présente les bonnes pratiques à suivre pour développer l'application de gouvernance de données dans le cadre du projet fédérateur.

1 Organisation des Branches Git

Pour garantir un dépôt GitHub propre et contrôlé, il faut respecter la structure suivante :

- **main** : contient uniquement du code stable et prêt à être déployé.
- **develop** : branche d'intégration, reçoit les fonctionnalités validées.
- **feature/<nom>** : branche pour travailler une nouvelle fonctionnalité.

2 Workflow de Développement

Chaque contribution doit suivre les étapes suivantes :

1. Créer une branche `feature/` depuis `develop`.
2. Développer et tester localement la fonctionnalité.
3. Respecter les conventions de codage (PEP8, typage, style).
4. Pousser la branche sur GitHub.
5. Ouvrir une Pull Request vers `develop`.
6. Vérifier que tous les tests CI passent.

7. Faire relire le code (code review).
8. Fusionner uniquement après validation.

Aucun commit direct ne doit être réalisé sur `main` ou `develop`.

3 Bibliothèques Python Recommandées

Pour structurer correctement un projet FastAPI, les bibliothèques suivantes sont obligatoires :

Back-end FastAPI

- `fastapi` : framework principal.
- `uvicorn` : serveur ASGI pour exécuter FastAPI.
- `pydantic` : validation des données et modèles.

Sécurité et Authentification

- `python-jose` : génération et validation JWT.
- `passlib` : hashage des mots de passe.
- `bcrypt` : hashing sécurisé.

Tests

- `pytest` : framework de tests.
- `pytest-asyncio` : tests asynchrones.
- `httpx` : client HTTP pour tester les endpoints FastAPI.

Qualité du Code

- `black` : formateur de code.
- `flake8` : vérification stylistique.
- `isort` : organisation des imports.
- `mypy` : vérification des types.

4 Structure Recommandée d'un Projet FastAPI

```
app/
 main.py
 config.py
 database.py

 routers/
    users.py
    auth.py

 models/
    user.py

 schemas/
    user_schema.py

 core/
    security.py
    auth.py

 tests/
     test_users.py
     test_auth.py
```

5 Tests Automatisés

Chaque fonctionnalité doit être testée :

- Tests unitaires (pytest)
- Tests d'intégration avec FastAPI TestClient
- Tests asynchrones avec `pytest-asyncio`

Les tests doivent s'exécuter automatiquement via GitHub Actions.

6 Intégration Continue (CI)

GitHub Actions doit vérifier :

- Lancement des tests PyTest
- Analyse du code (flake8, black –check, isort –check)
- Vérification du typage (mypy)
- Build du projet

Toute Pull Request doit afficher un statut **vert** avant d'être fusionnée.

7 Bonnes Pratiques de Développement

- Respecter la norme PEP8.
- Utiliser des noms de variables explicites.
- Documenter chaque fonction (docstring).
- Utiliser le typage Python (`typing`) obligatoirement.
- Garder les fichiers courts et lisibles.
- Un commit doit être petit et clair.
- Un commit = une idée, une modification cohérente.
- Ne pas versionner : fichiers temporaires, venv, données brutes.
- Utiliser un `.gitignore` complet.

8 Bonnes Pratiques Git / GitHub

- Rédiger des messages de commit lisibles :
  - `feat: ajout du système d'authentification`
  - `fix: correction du hashage des mots de passe`
- Toujours créer une Pull Request pour fusionner.
- Ne jamais travailler directement dans `main`.
- Écrire une description claire de la PR.
- Utiliser les issues GitHub pour organiser le travail.