**Documentation sur la Mise en Place, le Développement et la Gestion du Code avec GitHub**

**1. Introduction**  
Ce document récapitule les différentes étapes et bonnes pratiques mises en œuvre lors de la configuration et du développement de l’application DevOPTIQ. Il couvre :

* La mise en place de l’environnement de développement (Python, Flask, SQLAlchemy).
* La structure du projet.
* L’installation et l’utilisation de l’environnement virtuel.
* La configuration, le lancement et les tests basiques de l’application.
* La mise en place du versionnement du code source avec Git.
* L’intégration avec GitHub pour la sauvegarde, la collaboration et le suivi d’évolution.

**2. Structure du Projet**  
Le projet est organisé comme suit :

Projet OPTIQ - DevOPTIQ/

├─ Archives/

├─ Bases de données/

├─ Code/

│ ├─ app.py

│ ├─ extensions.py

│ ├─ models/

│ │ ├─ models.py

│ ├─ routes/

│ │ ├─ activities.py

│ ├─ instance/ (Contiendra la base optiq.db)

│ ├─ \_\_init\_\_.py (facultatif)

├─ requirements.txt

└─ ...

Points importants :

* app.py est le point d’entrée de l’application Flask.
* extensions.py contient l’initialisation de l’instance db pour SQLAlchemy.
* models/ et routes/ structurent le code (modèles de données, endpoints).
* Le dossier instance/ stocke la base de données optiq.db.
* requirements.txt liste les dépendances.

**3. Environnement Virtuel et Installation des Dépendances**

**3.1 Création et activation de l’environnement virtuel (Windows)**

Je ne suis pas certain de cette ligne et à quoi elle sert :

python -m venv Venv

cd C:\Users\Hubert.AFDEC\A.F.D.E.C\Projet OPTIQ – DevOPTIQ  
dans Visual Strudio  
.\venv\Scripts\Activate.ps1

Dans la fenêtre de Command Windows

venv\Scripts\activate

Pour désactiver l’environnement virtuel :

deactivate

**3.2 Installation des dépendances**  
À la racine du projet :

pip install -r requirements.txt

**4. Configuration de l’Application**

**4.1 extensions.py**

from flask\_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

**4.2 models/models.py**

from extensions import db

class Activity(db.Model):

\_\_tablename\_\_ = 'activities'

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(100), nullable=False)

**4.3 routes/activities.py**

from flask import Blueprint, jsonify, request

from extensions import db

from models.models import Activity

activities\_bp = Blueprint('activities', \_\_name\_\_)

@activities\_bp.route('/activities', methods=['GET'])

def get\_activities():

activities = Activity.query.all()

return jsonify([{"id": a.id, "name": a.name} for a in activities])

@activities\_bp.route('/activities', methods=['POST'])

def create\_activity():

data = request.get\_json()

new\_activity = Activity(name=data['name'])

db.session.add(new\_activity)

db.session.commit()

return jsonify({"id": new\_activity.id, "name": new\_activity.name}), 201

**4.4 app.py**

import os

from flask import Flask

from extensions import db

from models.models import Activity

from routes.activities import activities\_bp

from sqlalchemy import text

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] = f"sqlite:///{os.path.join(os.path.dirname(\_\_file\_\_), 'instance', 'optiq.db')}"

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

db.init\_app(app)

app.register\_blueprint(activities\_bp)

@app.route('/')

def hello():

return 'Hello, Flask!'

if \_\_name\_\_ == '\_\_main\_\_':

instance\_path = os.path.join(os.path.dirname(\_\_file\_\_), 'instance')

if not os.path.exists(instance\_path):

os.makedirs(instance\_path)

with app.app\_context():

db.create\_all()

try:

db.session.execute(text('SELECT 1'))

print("Connexion à la base de données réussie.")

except Exception as e:

print(f"Erreur lors de la connexion à la base de données : {e}")

app.run(debug=True)

**5. Lancement et Test de l’Application**

1. Se placer dans le répertoire Code :
2. cd C:\Users\Hubert.AFDEC\A.F.D.E.C\Projet OPTIQ - DevOPTIQ\code
3. Activer l’environnement virtuel :
4. .\Venv\Scripts\Activate
5. Lancer l’application :
6. python app.py
7. Tester dans le navigateur :
   * [http://127.0.0.1:5000](http://127.0.0.1:5000/) affiche "Hello, Flask!"
   * <http://127.0.0.1:5000/activities> affiche [] si aucune activité n’est en base.
8. Ajouter une activité (dans un second terminal, sans arrêter l’application) :
9. Invoke-RestMethod -Uri "http://127.0.0.1:5000/activities" -Method POST -ContentType "application/json" -Body '{"name":"Mon Activite"}'

Cette commande crée une nouvelle activité et renvoie un statut 201 CREATED.

1. Rafraîchir <http://127.0.0.1:5000/activities> pour voir l’activité.

**6. Gestion du Code Source avec Git et GitHub**

**6.1 Installation et Vérification de Git**  
Vérifier l’installation de Git :

git --version

Si Git n’est pas reconnu, installez-le depuis <https://git-scm.com/downloads>, puis relancez un terminal.

**6.2 Initialisation du Dépôt Git**  
À la racine du projet :

git init

git status

Créez ou modifiez .gitignore pour ignorer les fichiers indésirables (environnement virtuel, base de données locale, etc.) :

Venv/

Code/instance/

\_\_pycache\_\_/

\*.pyc

**6.3 Ajout des fichiers et Commit**

git add .

git commit -m "Initialisation du dépôt, ajout du code source et des documents"

**6.4 Connexion à GitHub**  
Créez un dépôt sur GitHub (par exemple DevOPTIQ). Vous obtiendrez une URL du type https://github.com/votre-nom-utilisateur/DevOPTIQ.git.

Liez le dépôt local au dépôt distant :

git remote add origin https://github.com/votre-nom-utilisateur/DevOPTIQ.git

Si origin existe déjà, ignorez cette étape.

**6.5 Pousser le Code sur GitHub**

git push -u origin main

Saisissez vos identifiants GitHub si nécessaire.  
Rendez-vous sur la page du dépôt GitHub pour vérifier que les fichiers sont bien présents.

**7. Bonnes Pratiques Git**

* **Commits fréquents et signifiants** : Faites des commits réguliers, avec des messages clairs.
* **Utilisation des branches** : Pour développer une nouvelle fonctionnalité, créez une branche dédiée.
* **.gitignore** : Maintenez à jour le .gitignore pour éviter d’ajouter des fichiers inutiles.
* **Tests et Intégration Continue** : Intégrez éventuellement des tests unitaires et un pipeline CI (GitHub Actions) pour vérifier la qualité du code à chaque commit.

**Remarques :**

Actuellement, le contrôle de version avec Git et l’hébergement sur GitHub que vous avez mis en place n’est pas automatique. Par défaut, vous devez régulièrement :

1. **Ajouter les modifications** :
2. git add .
3. **Créer un commit** décrivant les changements :
4. git commit -m "Description des modifications"
5. **Pousser ces changements vers GitHub** :
6. git push

Ces étapes ne se font pas toutes seules. C’est à vous de penser régulièrement à versionner votre travail. Cependant, il existe plusieurs moyens de faciliter ou d’automatiser ce processus :

**a. Intégration Git dans Visual Studio Code (ou Visual Studio)**

Si vous utilisez Visual Studio Code, l’intégration Git est native. Vous pouvez effectuer un commit, un push, un pull ou voir l’historique des modifications directement depuis l’interface graphique, sans taper de commande. Des icônes et des raccourcis permettent de gérer facilement le versionnement.

Avec Visual Studio (pas Code, mais l’IDE .NET), vous disposez aussi d’une intégration Git interne (selon les versions) qui permet de faire des commits et des push directement depuis l’interface. Cela reste manuel, mais plus accessible que la ligne de commande.

**b. Scripts simplifiés**

Vous pouvez créer un script (batch, comme pour l’initialisation de l’environnement) qui exécute automatiquement git add ., git commit -m "..." (avec un message générique) et git push. Ce script pourrait être lancé manuellement chaque fois que vous souhaitez sauvegarder. C’est simple à exécuter, mais vous perdez la personnalisation du message de commit, et il est moins recommandé d’automatiser complètement les commits (car le message doit décrire les changements).

**c. Automatisation partielle via des hooks**

Git dispose de mécanismes appelés « hooks » qui s’exécutent à certains moments (par exemple avant un commit, après un push, etc.). Vous pourriez théoriquement utiliser des hooks pour automatiser certaines étapes, mais cela demande un peu de configuration et ne dispense pas complètement de l’action manuelle.

**d. Intégration Continue (CI/CD) sur GitHub**

L’utilisation de GitHub Actions permet d’automatiser certaines opérations (tests, déploiements) après chaque push, mais elle n’automatise pas le commit et le push initial depuis votre machine. Il s’agit plutôt d’automatiser ce qui se passe après que vous ayez poussé votre code (tests, builds, déploiement).

**e. Synchronisation automatique (non recommandée pour le code)**

Il existe des outils pour synchroniser des répertoires avec un dépôt GitHub, mais ce n’est pas conseillé pour un projet de développement, car le concept du contrôle de version est justement de commiter au bon moment, avec des messages utiles, et non pas de tout enregistrer en continu.

**Conclusion :**

* Le processus n’est pas automatique par défaut : il faut penser à faire des commits et à pousser le code sur GitHub.
* Vous pouvez considérablement simplifier le processus en utilisant l’interface intégrée à Visual Studio ou Visual Studio Code, ce qui évite la saisie de plusieurs lignes de commande.
* Vous pouvez créer un petit script ou un raccourci, mais il est important de conserver un minimum de contrôle pour garder des messages de commits clairs et un historique cohérent.
* L’automatisation complète du versionnement est généralement déconseillée, car les opérations de commit et de push doivent rester conscientes et organisées.

**8. Étapes Suivantes**

* Ajouter des endpoints (PUT, DELETE) pour mettre à jour ou supprimer des activités.
* Mettre en place des tests unitaires (Pytest).
* Utiliser Flask-Migrate pour gérer les migrations de la base de données.
* Améliorer la documentation et la validation des données (Marshmallow, Pydantic).
* Continuer à itérer, versionner et améliorer le code.

**9. Redémarrage après arrêt de l’ordinateur :**

Ci-dessous, un exemple de script batch (.bat) qui permettra, après un redémarrage de l’ordinateur, de remettre en place votre environnement de développement. Ce script :

1. Vérifie la présence de Git et Python.
2. Active l’environnement virtuel Python.
3. Installe/Met à jour les dépendances à partir du requirements.txt.
4. Lance l’application Flask.

Ce script suppose que :

* Le projet se trouve dans C:\Users\Hubert.AFDEC\A.F.D.E.C\Projet OPTIQ - DevOPTIQ.
* Le répertoire Venv (environnement virtuel) est dans ce même répertoire.
* Le fichier requirements.txt est présent à la racine du projet.
* Le code de l’application (dont app.py) est dans le répertoire Code.

**Script batch (start\_environment.bat)**

@echo off

REM Ce script réinitialise l'environnement de développement et lance l'application Flask.

REM Emplacement du projet (adapté à votre situation)

set PROJECT\_DIR=C:\Users\Hubert.AFDEC\A.F.D.E.C\Projet OPTIQ - DevOPTIQ

REM Se placer dans le répertoire du projet

cd "%PROJECT\_DIR%"

echo Vérification de la version de Git...

git --version

if errorlevel 1 (

echo Git n'est pas trouvé. Veuillez installer Git ou vérifier votre variable PATH.

pause

exit /b 1

)

echo Vérification de la version de Python...

python --version

if errorlevel 1 (

echo Python n'est pas trouvé. Veuillez installer Python ou vérifier votre variable PATH.

pause

exit /b 1

)

echo Activation de l'environnement virtuel Python...

call "%PROJECT\_DIR%\Venv\Scripts\activate.bat"

if errorlevel 1 (

echo Impossible d'activer l'environnement virtuel. Vérifiez que le dossier Venv existe et est un environnement Python valide.

pause

exit /b 1

)

echo Installation/mise à jour des dépendances...

pip install -r "%PROJECT\_DIR%\requirements.txt"

if errorlevel 1 (

echo Echec de l'installation des dépendances. Vérifiez votre connexion internet ou le fichier requirements.txt.

pause

exit /b 1

)

echo Tout est prêt. Nous allons lancer l'application Flask.

cd "%PROJECT\_DIR%\Code"

echo Lancement de l'application (appuyez sur CTRL+C pour arrêter)...

python app.py

pause

**Instructions pour utiliser ce script :**

1. **Enregistrer le script :**  
   Copiez le code ci-dessus dans un fichier texte et enregistrez-le sous le nom start\_environment.bat à la racine de votre projet, c’est-à-dire dans C:\Users\Hubert.AFDEC\A.F.D.E.C\Projet OPTIQ - DevOPTIQ.
2. **Lancer le script :**  
   Après un redémarrage de votre ordinateur, ouvrez l’Explorateur de fichiers, allez dans le dossier C:\Users\Hubert.AFDEC\A.F.D.E.C\Projet OPTIQ - DevOPTIQ, et double-cliquez sur le fichier start\_environment.bat.  
   Vous pouvez également ouvrir une fenêtre de commande (CMD) ou PowerShell, naviguer jusqu’au répertoire du projet et taper :
3. .\start\_environment.bat
4. **Ce que fait le script :**
   * Il vérifie la disponibilité de Git et Python.
   * Il active l’environnement virtuel.
   * Il installe les dépendances requises.
   * Il se positionne dans le dossier Code et lance votre application Flask.

Si tous les prérequis sont en place, vous verrez des messages confirmant chaque étape, puis l’application Flask démarrera. Si un problème survient, un message d’erreur s’affichera et le script s’arrêtera.

1. **Arrêter l’application :**  
   L’application Flask restera active tant que la fenêtre de commande est ouverte. Pour l’arrêter, appuyez sur CTRL + C dans la fenêtre où elle s’exécute.

Ce script vous assure une remise en place rapide de votre environnement sans avoir à vous souvenir de chaque commande. Il vous suffira de le relancer après un redémarrage pour être opérationnel.

**Conclusion**  
Ce document fournit une base solide pour développer l’application DevOPTIQ, du paramétrage de l’environnement à la mise en place du dépôt GitHub. En suivant ces bonnes pratiques, vous pourrez faire évoluer votre application de manière structurée et sécurisée, en gardant une trace de l’historique de vos modifications et en facilitant la collaboration.