Projet DevOps : MyTube - Gestionnaire de Playlist Vidéos

🎯 Objectif général du projet

Créer une application **Flask** permettant de gérer vos vidéos YouTube favorites (ajout, consultation, etc.) et implémenter un workflow DevOps professionnel : tests automatisés et intégration continue avec GitHub Actions.

À la fin du projet, chaque commit sur votre repository GitHub déclenchera automatiquement une série d'actions de tests.

📔 Cahier des charges de votre projet

Vous allez créer une mini application Flask (en python) qui permettra de gérer vos vidéos youtube préférées. Vous pourrez ajouter de nouvelles vidéos youtube, les modifier, les supprimer, etc. Vous ajouterez les informations qui vous paraissent pertinentes, selon le temps et vos envies (nombre de fois que la vidéo a été visionnée, tags, ajouter/ modifier des notes personnelles, etc.).

Étape 1 : Configuration initiale du projet Flask

🎯 Objectif de l'étape [~ 4h]

À la fin de cette étape, vous devez avoir une application Flask fonctionnelle, le projet correctement configuré, et votre code mis sur Github.

📔 Tâches à réaliser :

T1.1 Créer un dossier pour le projet nommé **flask-devops-project**. Rangez vos dossiers !

T1.2 Créer (et activer !) un environnement virtuel Python isolé pour ce projet avec le nom **venv**.

Ressource sur les environnements virtuels Python : https://docs.python.org/fr/3/tutorial/venv.html

Conseil préliminaire pour la **gestion des données** de votre application :

- Utiliser un fichier **`videos.json`** pour la persistance
- Initialiser ce fichier avec 2-3 vidéos d'exemple pour faciliter les tests

```
Exemple de structure pour videos.json :
[
  {
    "id": 1,
    "url": "https://youtube.com/watch?v=abc123",
    "title": "Titre de la vidéo",
    "views": 0
  }
]
```

T1.3 Créer une application Flask minimaliste contenant les routes suivantes :

- Route GET **`/`** : Affiche la page d'accueil de votre site.
- Route GET **`/videos`** :
  - Affiche l'ensemble des vidéos disponibles
  - **Conseil** : Utilisez une boucle **`{% for video in videos %}`** dans le template
- Route GET/PUT/DELETE **`/videos/<int:id>`** :
  - GET : Affiche une vidéo ainsi que toutes les métadonnées sur cette vidéo (nombre de visionnages, …)
  - PUT : Modifier les informations d'une vidéo
  - DELETE : Supprimer la vidéo
- Route GET **`/videos/search`** :
  - Rechercher des vidéos par titre
  - Affiche un formulaire de recherche et les résultats de recherche.
- Route POST **`/videos/add`** :
  - route qui permet d'ajouter une nouvelle vidéo à la playlist
- Route GET **`/videos/add`** :
  - Affiche le formulaire pour ajouter une nouvelle video.

Ressource : Flask Get Started : https://flask.palletsprojects.com/en/stable/quickstart/

Conseil pour une **application plus professionnelle** :

- **Valider les données d'entrée** de votre application :
  - Vérifiez toujours que les données reçues sont au bon format
  - Retournez des messages d'erreur clairs et explicites si nécessaires

- Utiliser des **codes HTTP**, comme par exemple :
  - 200 : Succès (GET)
  - 201 : Création réussie (POST)
  - 400 : Requête invalide (données manquantes
  - 404 : Ressource non trouvée

T1.4 Créer et remplir un fichier requirements.txt. Installer les dépendances nécessaires.

T1.5 Lancer l'application Flask localement et tester "à la main" les routes dans votre navigateur.

T1.6 Mettez votre projet sur github

- Mettre le projet en **visibilité publique**
- Ne pas oublier de mettre un **.gitignore**
- Ne pas oublier de mettre (et remplir !) un fichier **README.md**

Étape 2 : Écrire des tests unitaires pour votre projet Flask

🎯 Objectif de l'étape [~ 5h]

À la fin de cette étape, vous devez avoir écrit des tests unitaires permettant de tester votre code python de manière automatique et de valider le bon fonctionnement de vos routes Flask.

📔 Tâches à réaliser :

T2.1 Suivre le tutoriel suivant les tests en python.
https://blog.ippon.fr/2025/02/10/maitriser-pytest-guide-complet-du-developpeur-python-1-3/

T2.2 Créer la structure de tests

- Créer un dossier **`tests/`** à la racine du projet ainsi que le fichier **`test_app.py`** pour y regrouper les tests.
- Créer un fichier **`conftest.py`** vide à la racine de votre projet (important pour éviter les erreurs d'import !)
- N'oubliez pas de rajouter pytest à votre requirements !

Conseil pour l'écriture des tests :

- Commencez par les tests les plus simples (vérifier qu'une page se charge) avant d'écrire des tests plus complexes
- Pour chaque route, testez au moins un cas "normal" et un cas "d'erreur"

Pour vous aider, voici un exemple de test pour la route **`"/"`** :

```
def test_home_page(client):
     """Test que la page d'accueil fonctionne"""
     client = app.test_client()
     response = client.get('/')
     assert response.status_code == 200
```

T2.3 Ecrire des tests adaptés à votre application Flask. Par exemple :

- La page d'accueil (`/`) se charge correctement et retourne un code HTTP 200
- La page liste des vidéos (`/videos`) se charge correctement
- Une vidéo spécifique peut être affichée (`/videos/1`)
- Une vidéo inexistante retourne bien une erreur 404
- La recherche de vidéos fonctionne correctement
- L'ajout d'une nouvelle vidéo via le formulaire fonctionne

> **Note supplémentaire :**
>
> Dans un contexte professionnel DevOps, on écrit souvent les tests **avant** le code (approche TDD - Test-Driven Development). Pour ce projet, nous écrivons les tests après pour mieux comprendre, mais vous pouvez expérimenter cette approche si vous intégrez de nouvelles fonctionnalités !

Étape 3 : Automatiser les tests avec GitHub Actions

🎯 Objectif de l'étape [~ 5h]

À la fin de cette étape, vos tests s'exécuteront automatiquement à chaque modification de code (push) sur GitHub.

> **Qu'est-ce qu'un Workflow GitHub Actions et pourquoi les utiliser ?**
>
> Un **Workflow GitHub Actions** est une série de tâches (ou *jobs*) automatisées et configurables, et qui s'exécutent directement dans votre dépôt GitHub.
>
> Ces workflows - écrits dans un fichier YAML - sont conçus pour se déclencher automatiquement en réponse à des événements spécifiques, comme un `push` de code ou des **`pull_request`** sur une branche donnée (à définir).
>
> Dans notre cas nous allons nous servir des **GitHub Actions** pour lancer les tests à chaque push sur la branche main.

📔 Tâches à réaliser :

T3.1 Comprendre les bases de GitHub Actions

- Lire cet article de blog pour comprendre les github actions (lisez jusqu'à la partie : "*Utilisation d'actions de la marketplace* ") : https://blog.stephane-robert.info/docs/pipeline-cicd/github/

T3.2 Créer votre premier workflow de tests

- Créer un fichier **`.github/workflows/tests.yml`** (pensez à créer les dossiers parents .github et workflows ;)
- Configurer le workflow pour qu'il se déclenche sur les événements **`push`** et **`pull_request`** de la branche **`main`**
- Définir un job qui s'exécute sur **`ubuntu-latest`**
- Récupérer le code du repository (utiliser l'action **`actions/checkout@v5`** : https://github.com/marketplace/actions/checkout)

Pour vous aider, voici un starter kit de démarrage pour le workflow github. Pour la suite, vous allez devoir chercher sur internet !

```yaml
name: Tests CI
on:
 push:
  branches: [ main ]

jobs:
 test:
  runs-on: ubuntu-latest

  steps:
   - name: Checkout code
     uses: actions/checkout@v4

   # etc .....
```

- Configurer python à la bonne version que vous utilisez (exemple : 3.11, 3.12, …) en utilisant l'action **`actions/setup-python@v6`** : https://github.com/marketplace/actions/setup-python
- En utilisant **`run`** Installer les dépendances depuis **`requirements.txt`**
- Exécuter les **`tests`** avec pytest en mode verbose
- Aller sur GitHub → onglet "Actions" de votre repository

- Observer le workflow s'exécuter automatiquement
- Vérifier que le workflow fonctionne correctement

Étape 4 : Protéger la branche main !

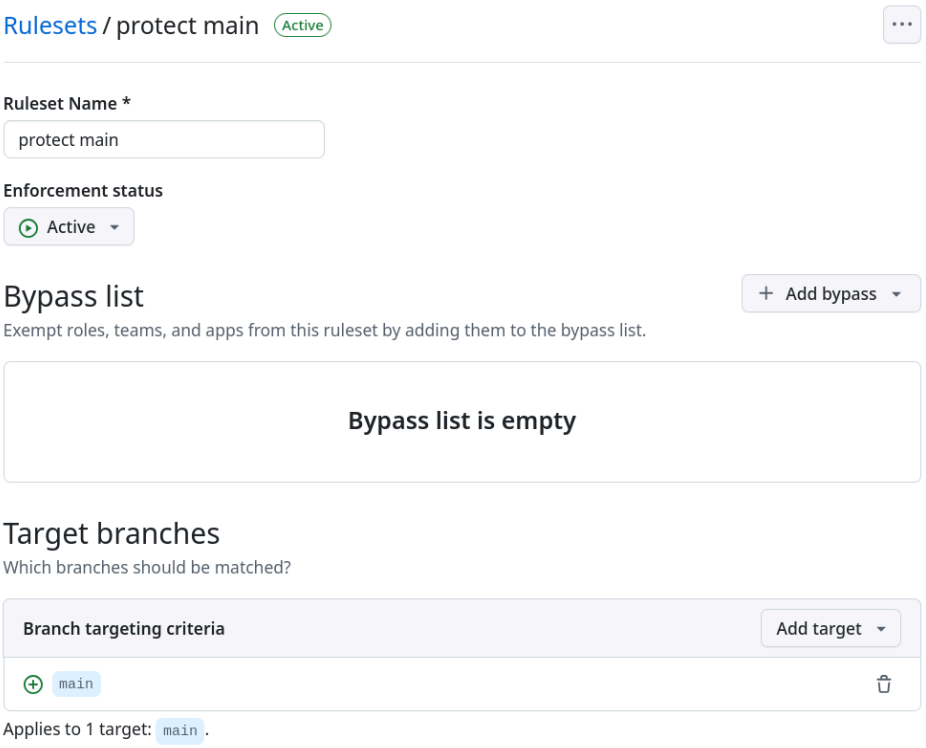
🎯 Objectif de l'étape [~ 5h]

À l'issue de cette étape, votre branche main sera protégée, c'est à dire que vous ne pourrez plus la mettre à jour par simple push. Toute mise à jour se fera obligatoirement par **pull request et merge**. Une pull request ne passant pas les tests unitaires ne pourront pas être fusionnée avec la branche **main.**

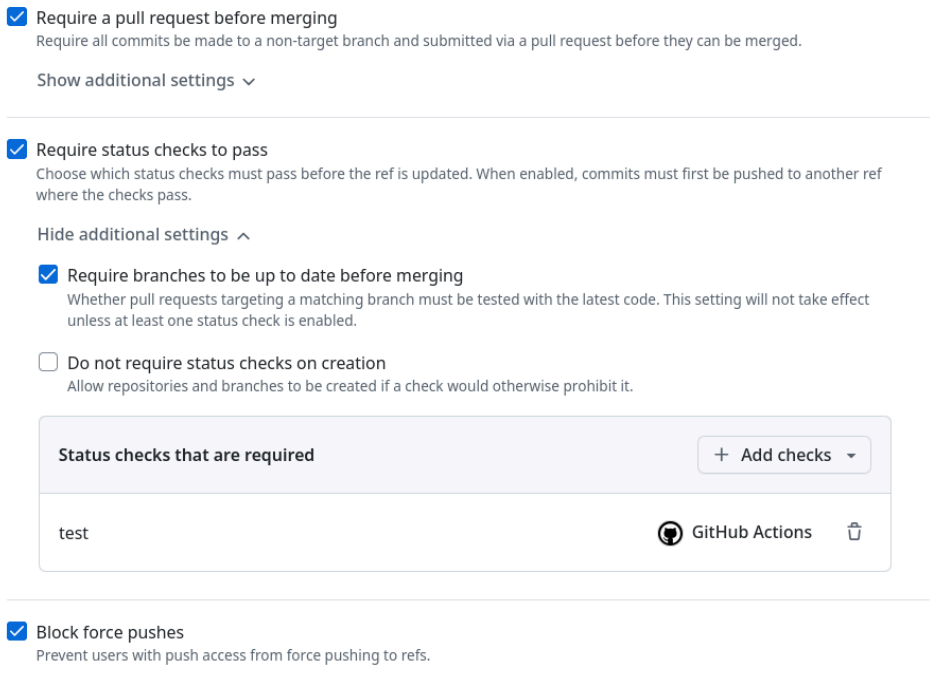
📔Tâches à réaliser :

T4.1 Protéger la branche **`main`** pour garantir la qualité du code

- Aller dans les paramètres de votre repository GitHub : **`Settings`** → **`Branches`** et configurer la protection :
    - **Add branch ruleset**
    - Donnez un nom à votre ruleset, par exemple **`"main protection"`**
    - Mettez le **enforcement status** à **`Active`**
    - Target branch : **`main`**
    - **`Require status check to pass`**
    - Ajoutez dans les paramètres optionnels et ajoutez les **status check that are required** : vous devez sélectionner dans la liste des status checks le nom du workflow à passer : le nom que vous avez donné dans `.github/workflows/tests.yml` (il devrait apparaître après avoir été exécuté au moins une fois)
- Cocher également "**Require branches to be up to date before merging"**
- Cocher **"Require pull request before merging**". Laisser les options par défaut
- Cliquer sur "Create" ou "Save changes" pour activer la protection



Capture d'écran #1 : Configuration du ruleset et choix de la branche à protéger



Capture d'écran #2 : ajout des status check

T4.2 Tester la protection de branche

- Créer une nouvelle branche : **`git checkout -b test-protection`**
- Faire une modification dans votre code qui fait échouer un test
- Commiter et pousser cette branche sur GitHub puis créer une Pull Request vers **`main (compare & pull request)`**
- Observer que GitHub bloque automatiquement la fusion tant que les tests n'ont pas tous réussi.
- Corriger le test, pousser à nouveau, et vérifier que la fusion est maintenant possible

Remarques importantes : Pourquoi protéger la branche main ?

Avec cette protection, vous ne pourrez plus faire **`git push origin main`** directement. Vous devrez toujours passer par une **Pull Request**.

Cette protection de branche est une pratique DevOps essentielle qui garantit que :

- Aucun code défectueux n'arrive en production
- Tous les tests doivent passer avant fusion
- Le code est toujours dans un état fonctionnel
- Les erreurs sont détectées avant la fusion, pas après

Résultat : La branche **`main`** est garantie fonctionnelle à tout moment. C'est exactement comme cela que travaillent les équipes de développement en entreprise !