Ejercicio Práctico: Aplicación de Estrategias de Ramificación, Colaboración y Automatización con Git y GitHub en un Proyecto Python

Enunciado:

En este ejercicio, se simulará el flujo de trabajo de un equipo de desarrollo que aplica buenas prácticas en gestión de contribuciones, estrategias de ramificación y uso de herramientas de integración continua y gestión de proyectos en GitHub, utilizando un proyecto en **Python**. Incorporarán una rama en el flujo de trabajo siguiendo un modelo adaptado de **GitFlow**.

Objetivos del ejercicio:

1. **Crear y configurar un repositorio en GitHub** llamado `proyecto-python`.
2. **Implementar la estrategia de ramificación GitFlow** con ramas adicionales `qa` y `release`, estableciendo las ramas `main`, `develop` y `qa`.
3. **Desarrollar una nueva funcionalidad** en una *feature branch* y aplicar buenas prácticas de commits.
4. **Realizar un Pull Request** para integrar la funcionalidad en `develop`, simulando el proceso de revisión de código.
5. **Fusionar `develop` en `qa` para pruebas** y corregir errores si es necesario.
6. **Preparar una versión para lanzamiento** creando una *release branch* desde `qa` y fusionándola en `main` y `develop`.
7. **Configurar GitHub Actions** para automatizar pruebas y despliegue, incluyendo al menos **3 pruebas distintas**:
    - **Pruebas unitarias con Pytest**.
    - **Análisis de código estático con Flake8**.
    - **Análisis de seguridad con Bandit**.
8. **Configurar una GitHub Action que realice un rollback** en caso de fallo, incluyendo el código detallado.
9. **Utilizar GitHub Projects** para gestionar las tareas del proyecto, creando un tablero Kanban y automatizando el flujo de trabajo.

Pasos y Código para Realizar el Ejercicio:

Paso 1: Crear y Configurar el Repositorio en GitHub

1. **Crear el repositorio:**
    - Inicia sesión en tu cuenta de GitHub.
    - Haz clic en **"New"** para crear un nuevo repositorio.
    - Nombra el repositorio `proyecto-python`.
    - Agrega una descripción opcional.
    - Marca la casilla **"Initialize this repository with a README"**.
    - Añade un archivo `.gitignore` apropiado (selecciona **"Python"**).
    - Selecciona una licencia (por ejemplo, **MIT**).
    - Haz clic en **"Create repository"**.

Clonar el repositorio en tu máquina local:
bash

```
git clone https://github.com/tu_usuario/proyecto-python.git
cd proyecto-python
```

Configurar las ramas `main`, `develop` y `qa`:
bash

```
# Asegúrate de estar en 'main'
git checkout main

# Crear y cambiar a la rama 'develop'
git checkout -b develop

# Subir la rama 'develop' al repositorio remoto
git push origin develop

# Crear y cambiar a la rama 'qa' desde 'develop'
git checkout -b qa

# Subir la rama 'qa' al repositorio remoto
git push origin qa
```

Paso 2: Implementar la Estrategia de Ramificación GitFlow con Rama `qa`

1. **Inicializar Git Flow (opcional):**

Instala la extensión `git-flow` si aún no la tienes:
bash

```
# En macOS
brew install git-flow-avh

# En Ubuntu/Debian
sudo apt-get install git-flow
```

Para instalar Git Flow en Windows de la manera más corta, hay dos métodos principales:

1. **Usando Git for Windows (GitBash)** - El método más rápido:

bash

```
git flow init
```

Si ya tienes Git instalado, este comando inicializará Git Flow. Si no está disponible, usa el segundo método.

2. **Usando Chocolatey** - Método alternativo en una sola línea:

bash

```
choco install gitflow-winget -y
```

El primer método es el más directo si ya tienes Git instalado. Si necesitas instalar Chocolatey primero, puedes hacerlo con este comando en PowerShell (como administrador):

powershell

```
Set-ExecutionPolicy Bypass -Scope Process -Force;
[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex
((New-Object
System.Net.WebClient).DownloadString('https://community.chocolatey.org/ins
tall.ps1'))
```

Te recomiendo el primer método (`git flow init`) si ya tienes Git instalado, ya que es la forma más rápida y directa.

Inicializa Git Flow en tu repositorio:
bash

```
git flow init
```

- Acepta los nombres por defecto para las ramas (`main`, `develop`, etc.).
- **Nota:** La rama `qa` no es parte del GitFlow original, por lo que la gestionaremos manualmente.

Paso 3: Desarrollar una Nueva Funcionalidad en una Feature Branch
Crear una *feature branch* para la nueva funcionalidad:

bash

```
# Usando git-flow
git flow feature start agregar-login

# Si no usas git-flow
# git checkout -b feature/agregar-login develop
```

1. **Desarrollar la funcionalidad:**

Crea un archivo `app.py` y agrega el siguiente código:
python

```
# app.py
def login(username, password):
    # Lógica ficticia de autenticación
    if username == "admin" and password == "1234":
        return "Autenticación exitosa"
    else:
        return "Credenciales incorrectas"

if __name__ == "__main__":
    # Simulación de login
    print(login("admin", "1234"))
```

Actualiza el `README.md` con instrucciones sobre cómo ejecutar la aplicación:
markdown

````
# Proyecto Python

Este es un proyecto de ejemplo en Python que incluye una función
de login básica.

## Ejecución

Requisitos: Python 3.x

Ejecuta el siguiente comando para probar la aplicación:

```bash
python app.py
````

Realizar commits con mensajes claros:
bash

```
git add .
git commit -m "Agregar función de login básica"
```

Subir la *feature branch* al repositorio remoto:
bash

```
git push origin feature/agregar-login
```

Paso 4: Realizar un Pull Request y Simular la Revisión de Código

1. **Crear un Pull Request en GitHub:**
    - Navega al repositorio en GitHub.
    - Haz clic en **"Compare & pull request"**.
    - Asegúrate de que la rama base sea `develop` y la de comparación sea `feature/agregar-login`.
    - Proporciona un título descriptivo y una descripción detallada del PR.
    - Asigna el PR y agrega etiquetas si lo deseas.
    - Haz clic en **"Create pull request"**.
2. **Simular la revisión de código:**
    - Revisa el código en el PR.
```

- Agrega comentarios en líneas específicas si consideras que hay mejoras.

Si es necesario, realizar cambios adicionales:
bash

```
# Realiza cambios en el código si se solicitaron ajustes
git add .
git commit -m "Corregir comentarios de revisión: mejorar mensajes
de salida"
git push origin feature/agregar-login
```

- Los nuevos commits aparecerán en el PR.

3. **Aprobar y fusionar el Pull Request:**
   - Una vez satisfecho con los cambios, aprueba el PR.
   - Haz clic en **"Merge pull request"** en GitHub.
   - Elige la opción de fusión adecuada.
   - Confirma la fusión.

4. Eliminar la rama de funcionalidad:
bash

```
git branch -d feature/agregar-login
git push origin --delete feature/agregar-login
```

Paso 5: Fusionar `develop` en `qa` para Pruebas y Corregir Errores

1. **Fusionar `develop` en `qa`:**
   bash

```
git checkout qa
git pull origin qa
git merge develop
git push origin qa
```

2. **Realizar pruebas en la rama `qa`:**
   - Despliega la rama `qa` en un entorno de pruebas.
   - Ejecuta pruebas manuales y/o automáticas.
3. **Corregir errores encontrados en `develop`:**

Si se encuentran errores, cambia a `develop`:
bash

```
git checkout develop
# Realiza las correcciones necesarias
git add .
git commit -m "Corregir error X encontrado en QA"
git push origin develop
```

Fusiona nuevamente `develop` en `qa` para verificar las correcciones:
bash

```
git checkout qa
git merge develop
git push origin qa
```

2. **Repetir el proceso hasta que `qa` esté estable.**

Paso 6: Preparar una Release Branch y Fusionarla en `main` y `develop`

Crear una *release branch* desde `qa`:
bash

```
git checkout qa
git checkout -b release/v1.0.0
git push origin release/v1.0.0
```

1. **Realizar ajustes finales en la rama `release/v1.0.0`:**

Actualiza la versión en un archivo `version.txt` o en el `README.md`:
markdown
Copiar código

```
## Versión 1.0.0
```

- Asegúrate de que todo funciona correctamente.

bash

```
git add .
git commit -m "Preparar versión 1.0.0 para lanzamiento"
```

Fusionar la *release branch* en `main`:
bash

```
git checkout main
git pull origin main
git merge --no-ff release/v1.0.0
git push origin main
```

Etiquetar la versión:
bash

```
git tag -a v1.0.0 -m "Lanzamiento de versión 1.0.0"
git push origin main --tags
```

Fusionar los cambios de la *release branch* en `develop`:
bash

```
git checkout develop
git merge --no-ff release/v1.0.0
git push origin develop
```

Eliminar la *release branch*:
bash

```
git branch -d release/v1.0.0
git push origin --delete release/v1.0.0
```

Paso 7: Configurar GitHub Actions para Automatizar Pruebas y Despliegue

Configuraremos **tres GitHub Actions** para pruebas y una acción adicional para realizar un **rollback** en caso de fallo.

Action 1: Pruebas Unitarias con Pytest

1. **Crear pruebas unitarias usando Pytest:**

Instala Pytest:
bash

```
pip install pytest
```

Crea el archivo `test_app.py`:
python

```
# test_app.py
from app import login

def test_login_correct_credentials():
    assert login("admin", "1234") == "Autenticación exitosa"

def test_login_incorrect_credentials():
    assert login("user", "wrongpassword") == "Credenciales
incorrectas"
```

2. **Agregar el flujo de trabajo para ejecutar pruebas unitarias:**

Crea el archivo `.github/workflows/unit-tests.yml`:
yaml

```
name: Unit Tests

on:
  push:
    branches: [ qa ]
  pull_request:
    branches: [ qa ]

jobs:
  test:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.x'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
```

```
        pip install pytest

    - name: Run tests
      run: |
        pytest
```

Action 2: Análisis de Código con Flake8

1. **Configurar Flake8 para análisis estático:**

Instala Flake8:
bash

```
pip install flake8
```

2. **Agregar el flujo de trabajo para análisis de código:**

Crea el archivo `.github/workflows/code-analysis.yml`:
yaml

```
name: Code Analysis

on:
  push:
    branches: [ qa ]
  pull_request:
    branches: [ qa ]

jobs:
  lint:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
```

```
        python-version: '3.x'

    - name: Install Flake8
      run: |
        python -m pip install --upgrade pip
        pip install flake8

    - name: Run Flake8
      run: |
        flake8 app.py test_app.py
```

Action 3: Pruebas de Seguridad con Bandit

1. **Configurar Bandit para análisis de seguridad:**

Instala Bandit:
bash

```
pip install bandit
```

2. **Agregar el flujo de trabajo para pruebas de seguridad:**

Crea el archivo `.github/workflows/security-scan.yml`:
yaml

```
name: Security Scan

on:
  push:
    branches: [ qa ]
  pull_request:
    branches: [ qa ]

jobs:
  security:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python
```

```
      uses: actions/setup-python@v4
      with:
        python-version: '3.x'

    - name: Install Bandit
      run: |
        python -m pip install --upgrade pip
        pip install bandit

    - name: Run Bandit
      run: |
        bandit -r .
```

Action 4: Realizar un Rollback en Caso de Fallo

1. **Crear scripts de despliegue y rollback:**

deploy.sh:
bash

```
#!/bin/bash
echo "Desplegando la aplicación..."

# Simular un despliegue
DEPLOY_SUCCESS=${DEPLOY_SUCCESS:-true}

if [ "$DEPLOY_SUCCESS" != "true" ]; then
    echo "Error en el despliegue"
    exit 1
fi

echo "Despliegue completado con éxito"
```

rollback.sh:
bash

```
#!/bin/bash
echo "Realizando rollback a la versión anterior..."
# Simular rollback
echo "Rollback completado"
```

Hacer ejecutables los scripts:
bash

```
chmod +x deploy.sh rollback.sh
```

2. **Agregar el flujo de trabajo para despliegue y rollback:**

Crea el archivo `.github/workflows/deploy.yml`:
yaml

```
name: Deploy and Rollback

on:
  push:
    branches: [ main ]

jobs:
  deploy:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Ejecutar despliegue
      env:
        DEPLOY_SUCCESS: false  # Cambiar a 'true' para simular
éxito
      run: |
        ./deploy.sh

  rollback:

    needs: deploy
    if: failure()
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Ejecutar rollback
```

```
      run: |
        ./rollback.sh
```

- **Explicación:**
  - El job `deploy` intenta desplegar la aplicación.
  - Si `DEPLOY_SUCCESS` es `false`, el despliegue falla y el job `rollback` se ejecuta.
  - El job `rollback` realiza el rollback a la versión anterior.

3. **Probar el flujo de trabajo:**

Realiza un commit y push a `main` para desencadenar el flujo:
bash

```
git add .
git commit -m "Agregar scripts de despliegue y rollback"
git push origin main
```

Paso 8: Utilizar GitHub Projects para Gestionar las Tareas del Proyecto

1. **Crear un proyecto en GitHub:**
   - En el repositorio, ve a la pestaña **"Projects"**.
   - Haz clic en **"New project"**.
   - Selecciona **"Board"** para crear un tablero Kanban.
   - Nombra el proyecto, por ejemplo, **"Gestión del Proyecto"**.
   - Configura las columnas:
     - **To Do**
     - **In Progress**
     - **QA Testing**
     - **Review**
     - **Done**
2. **Agregar issues al proyecto:**
   - Crea algunos issues representando tareas:
     - Implementar función de registro de usuarios.
     - Mejorar la interfaz de usuario.
     - Documentar la API.
   - Al crear cada issue, asignarlo al proyecto en la sección **"Projects"**.
3. **Automatizar el flujo de trabajo:**
   - Configura automatizaciones para mover tarjetas automáticamente entre columnas según el estado de los issues o pull requests.
   - Por ejemplo:

        - Cuando se abre un **pull request** hacia `develop`, la tarjeta se mueve a **"In Progress"**.
        - Cuando se fusiona en `develop`, la tarjeta se mueve a **"QA Testing"**.
        - Cuando se fusiona en `main`, la tarjeta se mueve a **"Done"**.

Resumen y Conclusión

Al completar este ejercicio, habrás aplicado los conceptos de:

- **Gestión de contribuciones y revisión de código:**
    - Uso de pull requests y revisión de código en GitHub.
    - Buenas prácticas en mensajes de commit y documentación.
- **Estrategias de ramificación y flujo de trabajo colaborativo con rama `qa`:**
    - Implementación de GitFlow adaptado para incluir una rama `qa`.
    - Trabajo con ramas de funcionalidad, `qa` y lanzamiento.
    - Proceso de pruebas en `qa` y corrección de errores.
- **Uso de herramientas de integración continua y gestión de proyectos en GitHub:**
    - Configuración de GitHub Actions para automatizar pruebas unitarias, análisis de código y pruebas de seguridad en la rama `qa`.
    - Configuración de una acción para realizar rollback en caso de fallos en el despliegue.
    - Uso de GitHub Projects para gestionar tareas y automatizar el flujo de trabajo.

Este ejercicio te proporciona una experiencia práctica en la simulación de un entorno colaborativo de desarrollo en Python, aplicando buenas prácticas y utilizando las herramientas integradas que GitHub ofrece, incluyendo la incorporación de una rama `qa` para mejorar el proceso de pruebas y calidad.