Guía de Ejercicios Prácticos con Hooks de Git

Los **hooks de Git** son scripts que se ejecutan automáticamente en respuesta a ciertos eventos en el ciclo de vida de Git, como commits, pushes o merges. En esta guía, exploraremos el uso de los hooks de Git a través de una serie de ejercicios prácticos que te permitirán comprender y aprovechar todo su potencial.

Los ejercicios se centrarán en:

- Gestión de ramas como `main`, `qa` y `dev`.
- Implementación de pruebas unitarias y de integración mediante hooks.
- Creación de hooks para validar el estándar de mensajes de commit.
- Exploración de diversas aplicaciones de hooks para automatizar tareas y mejorar el flujo de trabajo.

Ejercicio 1: Configuración Inicial y Gestión de Ramas

Objetivo:

Aprender a crear y gestionar ramas (`main`, `qa`, `dev`) en un repositorio Git y preparar el entorno para los ejercicios posteriores.

Instrucciones:

1. **Crear un Nuevo Repositorio Git:**

Crea un directorio para el proyecto y navega a él:

```
mkdir proyecto-hooks
cd proyecto-hooks
```

Inicializa un repositorio Git:

```
git init
```

2. **Crear el Archivo `README.md`:**

Crea un archivo `README.md` con contenido básico:

```
echo "# Proyecto Hooks de Git" > README.md
git add README.md
git commit -m "Inicializa el repositorio con README"
```

3. **Crear las Ramas `dev` y `qa`:**

Crea y cambia a la rama `dev`:
```
git checkout -b dev
```

Crea y cambia a la rama `qa` desde `dev`:
```
git checkout -b qa
```

Vuelve a la rama `dev`:
```
git checkout dev
```

4. **Estructura de Ramas:**

   Ahora tienes tres ramas:

   `main`: Rama principal, estable.

   `dev`: Rama de desarrollo donde se implementan nuevas funcionalidades.

   `qa`: Rama de pruebas donde se verifica la calidad antes de fusionar a `main`.

Explicación:

Este ejercicio establece la base para trabajar con distintas ramas que representan diferentes etapas en el flujo de desarrollo. Es esencial para los siguientes ejercicios, donde implementaremos hooks y realizaremos pruebas en estas ramas.

Ejercicio 2: Creación de Hook `commit-msg` para Validar Mensajes de Commit

Objetivo:

Crear un hook `commit-msg` que valide que los mensajes de commit siguen un estándar específico, mejorando la consistencia y calidad del historial de commits.

Instrucciones:

1. **Definir el Estándar de Mensajes de Commit:**

   Usaremos el estándar **Conventional Commits**, que utiliza prefijos como `feat`, `fix`, `docs`, etc.

2. **Crear el Hook `commit-msg`:**

En el directorio del repositorio, crea el directorio `hooks` para almacenar los hooks compartidos:

```
mkdir hooks
```

Crea el archivo `hooks/commit-msg`:

```
touch hooks/commit-msg
chmod +x hooks/commit-msg
```

3. **Escribir el Código del Hook:**

Abre `hooks/commit-msg` en tu editor y agrega el siguiente código:

```
#!/bin/

commit_msg_file="$1"
commit_msg=$(cat "$commit_msg_file")

regex="^(feat|fix|docs|style|refactor|perf|test|chore)(\([\w\-]+\))?
: .{1,50}"

if ! [[ $commit_msg =~ $regex ]]; then
  echo "ERROR: El mensaje de commit no cumple con el estándar
Conventional Commits." >&2
  echo "Ejemplos de mensajes válidos:" >&2
  echo "  feat(login): agregar autenticación de usuario" >&2
  echo "  fix(api): corregir error en la respuesta del endpoint" >&2
  exit 1
fi
```

4. **Configurar Git para Usar el Directorio de Hooks Compartido:**

Configurar Git para utilizar el directorio `hooks` como ruta de hooks:

```
git config core.hooksPath hooks
```

5. **Probar el Hook:**

Intenta hacer un commit con un mensaje que no siga el estándar:

```
echo "Contenido de prueba" > archivo.txt
git add archivo.txt
git commit -m "Agrega archivo de prueba"
```

> Deberías ver un mensaje de error indicando que el mensaje de commit no cumple con el estándar.

Ahora, intenta con un mensaje válido:

```
git commit -m "feat(test): agrega archivo de prueba"
```

> El commit debería proceder sin problemas.

Explicación:

Este hook garantiza que todos los mensajes de commit sigan un formato consistente, lo que facilita la lectura del historial y la integración con otras herramientas.

Ejercicio 3: Implementación de Pruebas Unitarias con Hook `pre-commit`

Objetivo:

Configurar un hook `pre-commit` que ejecute pruebas unitarias antes de permitir que se realice un commit, asegurando que el código cumple con los requisitos de calidad.

Instrucciones:

1. **Preparar el Entorno de Pruebas:**

Suponiendo que trabajamos con Python, crea un directorio `src` y un módulo simple:

```
mkdir src
echo "def suma(a, b): return a + b" > src/operaciones.py
```

Crea el directorio de pruebas y un archivo de prueba:

```
mkdir tests
echo "from src.operaciones import suma
```

def test_suma(): assert suma(2, 3) == 5" > tests/test_operaciones.py ```

2. **Instalar Dependencias:**

Crea un archivo `requirements.txt`:

```
pytest
```

Instala las dependencias:

```
pip install -r requirements.txt
```

3. **Crear el Hook `pre-commit`:**

Crea el archivo `hooks/pre-commit`:

```
touch hooks/pre-commit
chmod +x hooks/pre-commit
```

4. **Escribir el Código del Hook:**

Agrega el siguiente código a `hooks/pre-commit`:

```
#!/bin/

echo "Ejecutando pruebas unitarias..."

pytest
status=$?

if [ $status -ne 0 ]; then
  echo "ERROR: Las pruebas unitarias fallaron. Commit abortado." >&2
  exit 1
fi
```

5. **Probar el Hook:**

   Introduce un error en `src/operaciones.py`, por ejemplo, cambia `return a + b` por `return a - b`.

Intenta hacer un commit:

```
git add .
git commit -m "feat(operaciones): modifica función suma"
```

   El hook debería impedir el commit y mostrar un mensaje de error.

   Corrige el error y vuelve a intentarlo. El commit debería proceder.

Explicación:

Este hook automatiza la ejecución de pruebas unitarias antes de cada commit, asegurando que no se introducen cambios que rompan la funcionalidad existente.

Ejercicio 4: Implementación de Pruebas de Integración con Hook `pre-push`

Objetivo:

Configurar un hook `pre-push` que ejecute pruebas de integración antes de permitir que se realice un push al repositorio remoto, garantizando que el código integrado funciona correctamente.

Instrucciones:

1. **Crear Pruebas de Integración:**

Agrega una prueba de integración en `tests/test_integracion.py`:
python

```
from src.operaciones import suma

def test_integracion_suma():
    resultado = suma(10, 15)
    assert resultado == 25
```

2. **Crear el Hook `pre-push`:**

Crea el archivo `hooks/pre-push`:

```
touch hooks/pre-push
chmod +x hooks/pre-push
```

3. **Escribir el Código del Hook:**

Agrega el siguiente código a `hooks/pre-push`:

```
#!/bin/

echo "Ejecutando pruebas de integración..."

pytest tests/test_integracion.py
status=$?

if [ $status -ne 0 ]; then
  echo "ERROR: Las pruebas de integración fallaron. Push abortado."
>&2
  exit 1
fi
```

4. **Probar el Hook:**

   Introduce un error en `src/operaciones.py`, por ejemplo, cambia `return a + b` por `return a * b`.

   Realiza los commits necesarios.

Intenta hacer push a la rama `dev`:
`git push origin dev`

   El hook debería impedir el push y mostrar un mensaje de error.

   Corrige el error y vuelve a intentarlo. El push debería proceder.

Explicación:

Este hook ejecuta pruebas de integración antes de enviar código al repositorio remoto, asegurando que la integración de diferentes partes del sistema funciona correctamente.

Ejercicio 5: Automatización de Formateo de Código con Hook `pre-commit`

Objetivo:

Crear un hook `pre-commit` que formatee automáticamente el código siguiendo un estándar definido antes de cada commit, mejorando la legibilidad y consistencia del código.

Instrucciones:

1. **Instalar una Herramienta de Formateo:**

Usaremos **Black** para Python:
```
pip install black
```

2. **Modificar el Hook `pre-commit`:**

Agrega el formateo de código al inicio del hook `hooks/pre-commit`:
```
#!/bin/

echo "Formateando código con Black..."

black src/ tests/

echo "Ejecutando pruebas unitarias..."

# Resto del código...
```

3. **Probar el Hook:**

   Introduce código mal formateado en `src/operaciones.py` (por ejemplo, sin espacios adecuados).

Intenta hacer un commit:
```
git add .
git commit -m "feat(operaciones): añade nuevas funciones"
```

   El hook debería formatear el código automáticamente y luego ejecutar las pruebas.

Explicación:

Al formatear el código automáticamente antes de cada commit, se mantiene un estilo consistente en todo el proyecto, lo que facilita la lectura y mantenimiento del código.

Ejercicio 6: Bloquear Commits Directos a la Rama `main` con Hook `pre-commit`

Objetivo:

Crear un hook `pre-commit` que impida hacer commits directamente en la rama `main`, forzando a los desarrolladores a utilizar ramas de funcionalidad y realizar pull requests.

Instrucciones:

1. **Modificar el Hook `pre-commit`:**

Añade el siguiente código al inicio de `hooks/pre-commit`:

```
#!/bin/

branch=$(git rev-parse --abbrev-ref HEAD)

if [ "$branch" == "main" ]; then
  echo "ERROR: No se permite hacer commits directos a 'main'." >&2
  echo "Por favor, utiliza una rama de desarrollo y realiza un pull
request." >&2
  exit 1
fi

# Resto del código...
```

2. **Probar el Hook:**

Cambia a la rama `main`:

```
git checkout main
```

Intenta hacer un commit:

```
echo "Cambios en main" >> archivo.txt
git add archivo.txt
git commit -m "feat(main): cambios directos en main"
```

El hook debería impedir el commit y mostrar un mensaje de error.

Explicación:

Este hook refuerza la política de flujo de trabajo, evitando cambios directos en la rama principal y promoviendo el uso de ramas de desarrollo y revisiones de código.

Ejercicio 7: Integración de Hooks en el Flujo de Trabajo con GitHub

Objetivo:

Aprender a integrar los hooks en un entorno de colaboración utilizando GitHub, asegurando que todos los miembros del equipo utilicen los mismos hooks.

Instrucciones:

1. **Subir el Repositorio a GitHub:**

   Crea un nuevo repositorio en GitHub.

Añade el repositorio remoto y empuja las ramas:

```
git remote add origin
https://github.com/tu_usuario/proyecto-hooks.git
git push -u origin main
git push origin dev
git push origin qa
```

2. **Compartir los Hooks:**

   Como los hooks no se comparten por defecto, necesitamos incluirlos en el repositorio y configurar `core.hooksPath`.

3. **Agregar un Script de Instalación de Hooks:**

Crea un script `install-hooks.sh` en la raíz del proyecto:

```
#!/bin/
git config core.hooksPath hooks
chmod +x hooks/*
```

Añade el script al repositorio:

```
git add install-hooks.sh
git commit -m "chore: agrega script de instalación de hooks"
git push origin dev
```

4. **Instruir al Equipo:**

Indica a los miembros del equipo que ejecuten el script `install-hooks.sh` después de clonar el repositorio:

```
./install-hooks.sh
```

5. **Documentar el Proceso:**

   Agrega instrucciones al `README.md` sobre cómo instalar los hooks.

Explicación:

Al compartir los hooks y proporcionar un script de instalación, aseguramos que todos los miembros del equipo utilicen los mismos hooks, manteniendo la consistencia y las buenas prácticas en el proyecto.

Ejercicio 8: Exploración de Otras Aplicaciones de Hooks

Objetivo:

Descubrir y aplicar otras funcionalidades de los hooks de Git para automatizar tareas y mejorar el flujo de trabajo.

Instrucciones:

1. **Hook `post-merge` para Instalar Dependencias:**

Crea el archivo `hooks/post-merge`:

```
touch hooks/post-merge
chmod +x hooks/post-merge
```

Agrega el siguiente código:

```
#!/bin/

if [ -f requirements.txt ]; then
  echo "Instalando dependencias de Python..."
  pip install -r requirements.txt
fi
```

Este hook se ejecuta después de un `git pull` o `git merge`, y si detecta cambios en `requirements.txt`, instala las dependencias automáticamente.

2. **Hook `pre-rebase` para Prevenir Rebases en Ramas Protegidas:**

Crea el archivo `hooks/pre-rebase`:

```
touch hooks/pre-rebase
chmod +x hooks/pre-rebase
```

Agrega el siguiente código:

```
#!/bin/

branch=$(git rev-parse --abbrev-ref HEAD)

if [ "$branch" == "main" ] || [ "$branch" == "qa" ]; then
  echo "ERROR: No se permite reescribir el historial de la rama
$branch mediante rebase." >&2
  exit 1
fi
```

Este hook impide realizar rebase en las ramas `main` y `qa` para proteger su historial.

3. **Probar los Hooks:**

   Realiza operaciones que desencadenan estos hooks y observa su comportamiento.

Explicación:

Los hooks de Git ofrecen una amplia gama de posibilidades para automatizar tareas y reforzar políticas de flujo de trabajo. Este ejercicio te invita a explorar y adaptar los hooks a las necesidades específicas de tu proyecto.