# BUENAS PRÁCTICAS Y OPTIMIZACIÓN DEL USO DE GIT

El uso eficiente de Git es esencial para el desarrollo de software moderno. Una gestión adecuada de este sistema de control de versiones no solo mejora la productividad, sino que también facilita la colaboración entre los miembros del equipo y el mantenimiento del código a largo plazo. A continuación, se detallan buenas prácticas y estrategias para optimizar el uso de Git, proporcionando explicaciones claras y ejemplos prácticos para cada punto y subpunto. Estas recomendaciones ayudarán a los desarrolladores a mantener un flujo de trabajo organizado, un historial de cambios limpio y a aprovechar al máximo las capacidades que Git ofrece en proyectos de cualquier escala. Al implementar estas prácticas, se logrará un desarrollo más eficiente y una mayor calidad en los proyectos de software

## 1. Normas de Escritura de Mensajes de Commit

Los mensajes de commit son fundamentales para comprender la evolución de un proyecto. Escribir mensajes claros y consistentes facilita la colaboración entre los miembros del equipo, la revisión del código y el mantenimiento a largo plazo del software. Un mensaje de commit bien redactado ayuda a otros desarrolladores a entender rápidamente qué cambios se realizaron y por qué, lo que es crucial para la eficiencia y la resolución de problemas. Por ello, es importante seguir normas de escritura al redactar estos mensajes, asegurando que sean descriptivos, concisos y útiles para todos los involucrados en el proyecto

### 1.1. Importancia de los Mensajes de Commit Claros

**Comunicación efectiva:** Los mensajes de commit sirven como una forma de comunicación entre desarrolladores. Un mensaje bien redactado ayuda a otros miembros del equipo a entender rápidamente qué cambios se realizaron y por qué.

**Historial Legible:** Un historial de commits con mensajes claros permite navegar y buscar cambios específicos de manera eficiente, lo que es crucial cuando se investiga un problema o se revisa el historial del proyecto.

**Facilitan el Depurado y la Resolución de Problemas:** Si surge un error, poder identificar cuándo y por qué se introdujo un cambio facilita enormemente el proceso de depuración y solución de problemas.

### 1.2. Normas Generales para Escribir Mensajes de Commit

1. **Usar el Modo Imperativo en Tiempo Presente:**

   Escribir los mensajes como si se estuviera dando una orden. Por ejemplo, "Agrega nueva funcionalidad" en lugar de "Agregar nueva funcionalidad" o "Agregado nueva funcionalidad".

**Ejemplo Correcto:** "Corrige error en el cálculo de impuestos".

**Ejemplo Incorrecto:** "Corregido error en el cálculo de impuestos".

**Explicación:** Esto ayuda a mantener consistencia y se alinea con cómo Git presenta los mensajes, por ejemplo, "Este commit hará X". El uso del imperativo facilita la lectura y comprensión de las acciones realizadas.

2. **Ser Breve y Conciso:**

   La línea de asunto (título) no debe exceder los 50 caracteres. Esto asegura que el mensaje sea legible en diversas interfaces y herramientas sin cortes ni truncamientos.

   Evitar detalles técnicos excesivos en el título; estos pueden incluirse en el cuerpo del mensaje si es necesario.

   **Explicación:** Un título corto y claro facilita la lectura rápida del historial y la identificación de cambios relevantes. Ayuda a los desarrolladores a comprender el propósito del commit de un vistazo.

3. **Incluir un Cuerpo de Mensaje (si es necesario):**

   Si el cambio es complejo o requiere contexto adicional, agregar un cuerpo al mensaje de commit separado del título por una línea en blanco.

   El cuerpo debe proporcionar detalles adicionales, explicar el "por qué" detrás del cambio y cualquier implicación que pueda tener.

   **Explicación:** Proporcionar contexto adicional ayuda a futuros desarrolladores (o a uno mismo) a entender las razones detrás de los cambios, lo que es especialmente útil en proyectos a largo plazo o cuando se retoma el proyecto después de un tiempo.

4. **Explicar el "Por Qué" y no solo el "Cómo":**

   El código muestra "cómo" se implementa algo; el mensaje de commit debe explicar "por qué" se hizo.

   **Ejemplo:** En lugar de "Cambia variable X a Y", puede ser "Optimiza el rendimiento cambiando la variable X a Y".

   **Explicación:** Conocer la razón detrás de un cambio es crucial para entender su propósito y evaluar su impacto en el proyecto. Esto es especialmente importante durante revisiones de código o auditorías.

5. **Referenciar Issues o Tickets:**

   Si el commit está relacionado con un issue o ticket en el sistema de seguimiento de incidencias, incluir una referencia como "Closes #123" o "Fixes #456".

   **Explicación:** Esto crea una conexión directa entre el código y el seguimiento de tareas, facilitando la trazabilidad, el seguimiento del progreso y el cierre automático de issues cuando se fusiona el commit.

**1.3. Estructura de un Mensaje de Commit**

- **Título (Asunto):**

  Debe ser un resumen breve y conciso del cambio realizado.

  Escrito en modo imperativo y tiempo presente.

  **Ejemplo:** "Agrega soporte para autenticación OAuth".

- **Cuerpo (Opcional):**

  Proporciona detalles adicionales si es necesario, como el motivo del cambio, cómo se implementa y cualquier información relevante.

**Ejemplo:**

```
Se implementa autenticación utilizando OAuth 2.0 para permitir a los
usuarios iniciar sesión con proveedores externos como Google y
Facebook.

Esto mejora la experiencia del usuario al simplificar el proceso de
inicio de sesión y aumenta la seguridad al delegar la gestión de
credenciales.
```

- **Pie de Mensaje (Opcional):**

  Incluye información adicional como referencias a tickets, notas de despliegue o advertencias.

  **Ejemplo:** "Closes #89"

### 1.4. Ejemplos Prácticos

**Ejemplo 1:**

```
Agrega función de búsqueda avanzada

Se implementa la función de búsqueda avanzada que permite filtrar
resultados por categoría y fecha. Esto mejora la usabilidad y cumple
con los requerimientos del cliente solicitados en la última reunión.
Closes #25
```

**Explicación:**

- El título resume claramente el cambio principal utilizando el modo imperativo.
- El cuerpo proporciona contexto adicional sobre qué se hizo y por qué, mencionando incluso la relación con requerimientos del cliente.
- La referencia al issue #25 facilita el seguimiento y cierre automático del ticket asociado en el sistema de gestión de proyectos.

**Ejemplo 2:**

```
Corrige error en el cálculo de impuestos

Se corrige un error donde los impuestos no se aplicaban
correctamente a productos exentos. Ahora, el cálculo se ajusta según
las regulaciones actuales establecidas en la ley fiscal.
Fixes #42
```

**Explicación:**

- El título indica claramente que se trata de una corrección de error, siguiendo las normas de escritura.
- El cuerpo explica la naturaleza del error y cómo se soluciona, proporcionando contexto sobre su impacto.
- La referencia al issue #42 ayuda en la trazabilidad y asegura que el problema reportado se considera resuelto.

**Ejemplo 3:**

```
Actualiza documentación de la API
Se agregan ejemplos de uso para los nuevos endpoints de
autenticación. Esto facilita la integración por parte de
desarrolladores externos y mejora la claridad de la documentación
técnica.
```

**Explicación:**

- El título comunica la acción de actualizar la documentación de manera concisa.
- El cuerpo explica qué se añadió y el beneficio que aporta, enfatizando la utilidad para desarrolladores externos.
- Aunque no hay una referencia a un issue, proporciona suficiente contexto para entender el cambio y su propósito.

## 2. Estrategias para Mantener un Historial de Cambios Limpio y Manejable

Un historial de cambios bien organizado facilita la revisión, mantenimiento y colaboración en un proyecto. Implementar estrategias para mantener este historial limpio es esencial para la eficiencia del equipo.

### 2.1. Uso de Ramas (Branches) para Aislar Trabajo

- **Feature Branches (Ramas de Funcionalidad):**

  Crear una nueva rama para cada funcionalidad o tarea específica que se vaya a desarrollar.

**Ejemplo:**

```
git checkout -b feature/autenticacion-oauth
```

**Explicación:** Esto permite desarrollar y probar cambios de manera aislada, sin afectar la rama principal o el trabajo de otros desarrolladores. Facilita la gestión de código y reduce conflictos al fusionar.

- **Ramas de Corrección (Hotfixes):**

  Utilizar ramas específicas para corregir errores críticos que deben solucionarse de inmediato.

**Ejemplo:**

```
git checkout -b hotfix/correccion-impuestos
```

**Explicación:** Permite aplicar correcciones rápidas directamente en producción o en la rama principal, asegurando que los errores críticos se resuelvan con prioridad sin esperar al ciclo de desarrollo normal.

### 2.2. Realizar Commits Atómicos

- **Commits Pequeños y Específicos:**

  Cada commit debe representar un cambio lógico y completo, como corregir un error específico o añadir una funcionalidad aislada.

**Explicación:** Esto facilita la revisión de cambios, permite revertir commits individuales sin afectar otras partes del código y mejora la comprensión del historial. También ayuda a identificar rápidamente la introducción de errores.

- **Ejemplo:**

En lugar de hacer un commit que incluya múltiples cambios no relacionados, dividirlos en commits separados y coherentes:

```
git commit -m "Agrega validación de campos en el formulario de registro"
git commit -m "Actualiza estilos CSS para mejorar la interfaz"
```

### 2.3. Rebase y Squash para Limpiar el Historial

- **Rebase Interactivo (`git rebase -i`):**

  Permite reescribir el historial de commits, combinando, editando o reordenando commits antes de fusionarlos con la rama principal.

**Ejemplo:**

```
git checkout feature/autenticacion-oauth
git rebase -i develop
```

**Explicación:** Ayuda a mantener un historial lineal y limpio, eliminando commits innecesarios o combinando varios commits pequeños en uno más significativo. Esto facilita la lectura y comprensión del historial por parte de todo el equipo.

- **Squash Commits:**

  Combinar múltiples commits en uno solo para simplificar el historial y agrupar cambios relacionados.

  Durante el rebase interactivo, reemplazar `pick` por `squash` en los commits que se deseen combinar.

**Ejemplo en el editor:**

```
pick a1b2c3d Agrega funcionalidad de autenticación
squash b2c3d4e Ajusta interfaz de inicio de sesión
squash c3d4e5f Corrige error en autenticación
```

**Explicación:** Esto resulta en un único commit que resume todos los cambios relacionados con la autenticación, facilitando la revisión y manteniendo el historial más limpio y manejable.

### 2.4. Uso de Tags para Marcar Versiones

- **Etiquetar Versiones Importantes:**

  Utilizar etiquetas (tags) para marcar puntos específicos en el historial, como versiones de lanzamiento o hitos significativos.

**Ejemplo:**

```
git tag -a v2.0.0 -m "Lanzamiento de versión 2.0.0 con nuevas funcionalidades"
git push origin v2.0.0
```

**Explicación:** Los tags facilitan la referencia a versiones específicas del código, lo que es útil para despliegues, retrocesos y documentación. Permiten identificar rápidamente el estado del código en un punto dado.

### 2.5. Eliminación de Ramas Obsoletas

- **Mantener el Repositorio Ordenado:**

  Después de fusionar una rama, eliminarla tanto local como remotamente para evitar confusiones y reducir el desorden.

**Ejemplo:**

```
git branch -d feature/autenticacion-oauth
git push origin --delete feature/autenticacion-oauth
```

**Explicación:** Esto ayuda a mantener un conjunto de ramas activo y relevante, facilitando la gestión del repositorio y evitando errores al trabajar con ramas obsoletas o inactivas.

### 2.6. Evitar Commits de Archivos Binarios Grandes

- **No Almacenar Archivos Pesados en Git:**

  Git no maneja eficientemente archivos binarios grandes, lo que puede ralentizar el rendimiento del repositorio y aumentar su tamaño de forma innecesaria.

  **Solución:** Utilizar herramientas como **Git Large File Storage (Git LFS)** para manejar archivos grandes.

**Ejemplo de Instalación de Git LFS:**

```
git lfs install
git lfs track "*.psd"
git add .gitattributes
git commit -m "Configura Git LFS para archivos PSD"
```

**Explicación:** Git LFS almacena archivos grandes fuera del repositorio principal, manteniendo el rendimiento y permitiendo el manejo eficiente de archivos pesados sin afectar negativamente al equipo.

### 2.7. Ejemplos Prácticos

**Rebase Interactivo para Limpiar Historial:**

**Antes del Rebase:**
```
git log --oneline
c3d4e5f Corrige error en la autenticación
b2c3d4e Ajusta interfaz de inicio de sesión
a1b2c3d Agrega funcionalidad de autenticación
```

**Comando de Rebase Interactivo:**
```
git rebase -i HEAD~3
```

**En el Editor, Cambiar a:**
```
pick a1b2c3d Agrega funcionalidad de autenticación
squash b2c3d4e Ajusta interfaz de inicio de sesión
squash c3d4e5f Corrige error en la autenticación
```

- **Después del Rebase:**

  Se obtiene un único commit que resume todos los cambios relacionados con la autenticación, haciendo que el historial sea más fácil de entender y revisar.

**Eliminar Ramas Fusionadas:**

**Después de Fusionar una Rama:**
```
git checkout develop
git merge feature/autenticacion-oauth
git branch -d feature/autenticacion-oauth
git push origin --delete feature/autenticacion-oauth
```

- **Explicación:** Esto mantiene el conjunto de ramas manejable y evita confusiones sobre el estado de las ramas, asegurando que el equipo trabaje con información actualizada.

**Etiquetar una Versión:**

**Crear y Empujar una Etiqueta:**
```
git tag -a v1.5.0 -m "Lanzamiento de versión 1.5.0 con mejoras de rendimiento"
git push origin v1.5.0
```

- **Explicación:** Las etiquetas permiten identificar fácilmente versiones específicas y son útiles para despliegues, retrocesos y referencias futuras en documentación o soporte técnico.

## 3. Automatización de Tareas Comunes con Hooks de Git

Los hooks de Git son scripts que se ejecutan automáticamente en respuesta a eventos en Git. Permiten automatizar tareas, mantener estándares y mejorar la eficiencia del flujo de trabajo del equipo de desarrollo.

### 3.1. Concepto de Hooks de Git

- **Ubicación y Configuración:**

  Los hooks se encuentran en el directorio `.git/hooks` de cada repositorio local.

  Por defecto, vienen con archivos de ejemplo con la extensión `.sample`. Para activarlos, se renombra el archivo sin la extensión y se asegura que sea ejecutable.

- **Tipos de Hooks:**

  **Del Lado del Cliente (Client-Side Hooks):**

  Se ejecutan en acciones locales como commits, merges y checkouts.

  **Ejemplos:** `pre-commit`, `commit-msg`, `post-commit`.

  **Del Lado del Servidor (Server-Side Hooks):**

  Se ejecutan en el servidor en eventos como la recepción de pushes.

  **Ejemplos:** `pre-receive`, `post-receive`.

### 3.2. Ejemplos Prácticos de Hooks

**Ejemplo 1: Validación de Mensajes de Commit con `commit-msg`**

- **Objetivo:** Asegurar que los mensajes de commit siguen un formato específico, como el estándar **Conventional Commits**, para mantener consistencia y claridad en el historial.
- **Implementación:**

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

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

**Escribir el Código del Hook:**

```
#!/bin/bash

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
```

**Explicación:**

El hook obtiene el mensaje de commit y lo valida contra una expresión regular que define el formato esperado.

Si el mensaje no cumple con el formato, muestra un error y aborta el commit.

Esto ayuda a mantener consistencia en los mensajes de commit, facilitando la lectura, el mantenimiento y la automatización de procesos como el versionado semántico.

**Ejemplo 2: Prevenir Commits Directos a `main` con `pre-commit`**

- **Objetivo:** Evitar que los desarrolladores hagan commits directamente en la rama `main`, promoviendo el uso de ramas de funcionalidad y revisiones de código a través de pull requests.
- **Implementación:**

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

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

**Escribir el Código del Hook:**

```
#!/bin/bash

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

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

**Explicación:**

El hook verifica la rama actual en la que se está trabajando.

Si la rama es `main`, muestra un mensaje de error y aborta el commit.

Esto asegura que los cambios en `main` sean controlados, revisados y aprobados, manteniendo la estabilidad del código en producción.

**Ejemplo 3: Formateo Automático de Código con `pre-commit`**

- **Objetivo:** Garantizar que el código cumpla con un estilo consistente antes de cada commit, utilizando un formateador automático como Prettier para JavaScript.
- **Implementación:**

**Instalar Prettier (para JavaScript):**

```
npm install --save-dev prettier
```

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

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

**Escribir el Código del Hook:**

```
#!/bin/bash

echo "Formateando código con Prettier..."

files=$(git diff --cached --name-only --diff-filter=ACM | grep
'\.js$')

if [ -n "$files" ]; then
  npx prettier --write $files
  git add $files
fi
```

**Explicación:**

El hook identifica los archivos JavaScript modificados que están en el área de preparación.

Ejecuta Prettier para formatear esos archivos, asegurando que se adhieren a los estándares de estilo definidos.

Vuelve a añadir los archivos formateados al área de preparación para que los cambios se incluyan en el commit.

Esto asegura que todo el código comiteado sigue el mismo estilo, mejorando la legibilidad y facilitando la colaboración.

**Ejemplo 4: Ejecutar Pruebas Unitarias con `pre-push`**

- **Objetivo:** Asegurarse de que el código pasa todas las pruebas antes de ser enviado al repositorio remoto, evitando introducir código defectuoso en la rama compartida.
- **Implementación:**

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

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

**Escribir el Código del Hook (usando Pytest para Python):**

```
#!/bin/bash

echo "Ejecutando pruebas unitarias..."

pytest
status=$?

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

**Explicación:**

El hook ejecuta las pruebas unitarias antes de permitir el push al repositorio remoto.

Si las pruebas fallan, muestra un mensaje de error y aborta el push, evitando que código potencialmente defectuoso se comparta.

Esto ayuda a mantener la integridad del código en la rama principal y reduce el riesgo de introducir bugs.

**Ejemplo 5: Bloquear Commits de Archivos Grandes con `pre-commit`**

- **Objetivo:** Prevenir que se añadan archivos de gran tamaño al repositorio, lo que puede afectar el rendimiento y aumentar innecesariamente el tamaño del repositorio.
- **Implementación:**

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

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

**Escribir el Código del Hook:**

```
#!/bin/bash

max_size=1048576  # 1MB en bytes

files=$(git diff --cached --name-only --diff-filter=A)

for file in $files; do
  if [ -f "$file" ]; then
    size=$(wc -c <"$file")
    if [ $size -gt $max_size ]; then
      echo "ERROR: El archivo $file es demasiado grande ($size
bytes). Tamaño máximo permitido es $max_size bytes." >&2
      exit 1
    fi
  fi
done
```

**Explicación:**

El hook verifica el tamaño de los archivos nuevos añadidos al área de preparación.

Si algún archivo supera el límite definido (1MB en este caso), muestra un mensaje de error y aborta el commit.

Esto mantiene el repositorio eficiente y evita problemas de rendimiento asociados con archivos de gran tamaño.

**Uso de Hooks para los Primeros Dos Puntos**

Además de los ejemplos anteriores, los hooks pueden utilizarse específicamente para reforzar las normas de escritura de mensajes de commit y estrategias para mantener un historial limpio y manejable.

**Ejemplo 6: Recordatorio de Actualización de Documentación con `prepare-commit-msg`**

- **Objetivo:** Recordar a los desarrolladores actualizar la documentación o agregar notas de versión cuando realizan cambios significativos en el código.
- **Implementación:**

**Crear el Hook `prepare-commit-msg`:**

```
touch .git/hooks/prepare-commit-msg
chmod +x .git/hooks/prepare-commit-msg
```

    - 

**Escribir el Código del Hook:**

```
#!/bin/bash

# Archivo de mensaje de commit
commit_msg_file="$1"

# Agregar nota al final del mensaje de commit
echo -e "\n\n[Recordatorio] Por favor, verifica si es necesario actualizar la documentación o los archivos de configuración relacionados." >> "$commit_msg_file"
```

**Explicación:**

El hook modifica el mensaje de commit antes de que se abra el editor, añadiendo un recordatorio importante.

Esto promueve buenas prácticas al asegurar que la documentación y otros recursos se mantienen actualizados con los cambios en el código.

Ayuda a mantener la coherencia y evita problemas causados por documentación desactualizada.

**Ejemplo 7: Forzar Rebase en Lugar de Merge con `pre-rebase`**

- **Objetivo:** Promover el uso de `rebase` en lugar de `merge` para mantener un historial lineal y limpio en las ramas de desarrollo, evitando commits de merge innecesarios.
- **Implementación:**

**Crear el Hook `pre-rebase`:**

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

- 

**Escribir el Código del Hook:**

```
#!/bin/bash

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

if [ "$branch" == "main" ] || [ "$branch" == "develop" ]; then
  echo "ERROR: No se permite reescribir el historial de la rama $branch mediante rebase." >&2
  echo "Por favor, utiliza 'git merge' para integrar cambios en esta rama." >&2
  exit 1
else
  echo "Realizando rebase en la rama $branch para mantener el historial limpio."
fi
```

**Explicación:**

El hook impide que se realice `rebase` en las ramas principales (`main`, `develop`) para evitar reescribir su historial compartido y potencialmente causar conflictos.

Permite y fomenta el uso de `rebase` en ramas de funcionalidad, lo que mantiene un historial más lineal y legible en esas ramas.

Esto equilibra la necesidad de mantener un historial limpio con la integridad de las ramas compartidas.

**Conclusión**

Implementar buenas prácticas en el uso de Git y automatizar tareas con hooks mejora significativamente la calidad del código y la eficiencia del equipo. Al adoptar estas estrategias, se logran mensajes de commit claros y consistentes, lo que facilita la comunicación y el mantenimiento del proyecto al asegurar que todos entienden los cambios realizados. Además, se mantiene un historial de cambios limpio y manejable, simplificando la revisión y comprensión de la evolución del código, aspecto esencial para la colaboración y el soporte a largo plazo. La automatización de tareas y el mantenimiento de estándares garantizan que se siguen las políticas del equipo, reduciendo errores humanos y mejorando la consistencia en todo el proyecto. En conjunto, estas prácticas optimizan el uso de Git y fortalecen el flujo de trabajo del equipo de desarrollo.