Guía completa de Git y GitHub

Edison Achalma
Escuela Profesional de Economía, Universidad Nacional de San Cristóbal de Huamanga

Resumen

Esta guía completa de Git y GitHub ofrece una visión integral y actualizada del sistema de control de versiones más utilizado en el desarrollo de software. Desde los fundamentos teóricos (arquitectura, objetos y estados de archivos) hasta comandos esenciales, flujos de trabajo diarios reales, resolución avanzada de conflictos, mejores prácticas modernas (incluyendo Conventional Commits y trunk-based development) y ejemplos prácticos aplicados en 2025, este documento sirve como referencia exhaustiva tanto para principiantes que dan sus primeros pasos como para desarrolladores intermedios que desean optimizar su flujo de trabajo diario en entornos individuales, equipos pequeños y proyectos colaborativos en GitHub.

Palabras Claves: Git, GitHub, Control de versiones

Tabla de contenidos

Edison Achalma https://orcid.org/0000-0001-6996-3364

El autor no tiene conflictos de interés que revelar. Los roles de autor se clasificaron utilizando la taxonomía de roles de colaborador (CRediT; https://credit.niso.org/) de la siguiente manera: Edison Achalma: conceptualización, redacción

La correspondencia relativa a este artículo debe dirigirse a Edison Achalma, Email: elmer.achalma.09@unsch.edu.pe

Guía completa de Git y GitHub

Git es un sistema de control de versiones distribuido rápido, escalable y eficiente. Fue creado por Linus Torvalds en 2005 para el desarrollo del kernel de Linux.

Características principales:

- Sistema distribuido (cada desarrollador tiene una copia completa del historial)
- Extremadamente rápido
- Soporte robusto para desarrollo no lineal (branching y merging)
- Integridad de datos garantizada mediante SHA-1

Arquitectura de Git

Git almacena los datos como instantáneas (snapshots) del proyecto completo, no como diferencias entre archivos. Cada commit es una instantánea completa del estado del proyecto.

Tres áreas principales:

1. **Working Directory**: Tu directorio de trabajo actual
2. **Staging Area (Index)**: Área de preparación para el próximo commit
3. **Repository (.git directory)**: Base de datos de objetos y metadata

1 Instalación y Configuración

1.1 Instalación

1.1.1 Método 1: Paquetes predeterminados (rápido y estable)

Linux (Ubuntu/Debian):

```
sudo apt-get update
sudo apt-get install git
```

Linux (Fedora):

```
sudo dnf install git
```

macOS:

```
brew install git
```

Windows: Descargar desde: https://git-scm.com/download/win

1.1.2 Método 2: Desde la fuente (versión más reciente)

1. Instala las dependencias:

```
sudo apt update
sudo apt install libz-dev libssl-dev libcurl4-gnutls-dev \
libexpat1-dev gettext cmake gcc
```

2. Descarga y descomprime la versión deseada (ejemplo: 2.34.1):

```
mkdir tmp && cd tmp
curl -o git.tar.gz \
https://mirrors.edge.kernel.org/pub/software/scm/git/git-2.34.1.tar.gz
tar -zxf git.tar.gz
cd git-*
```

3. Compila e instala:

```
make prefix=/usr/local all
sudo make prefix=/usr/local install
exec bash
```

1.2 Verificar Instalación

```
git --version
```

1.3 Configuración Inicial

Configurar identidad (obligatorio):

```
git config --global user.name "Tu Nombre"
git config --global user.email "tu.email@ejemplo.com"
```

Editor por defecto:

```
git config --global core.editor "vim"
git config --global core.editor "nano"
git config --global core.editor "code --wait"  # VS Code
```

Configurar rama principal:

```
git config --global init.defaultBranch main
```

Ver configuración:

```
git config --list
git config --list --show-origin  # Ver de dónde viene cada configuración
git config user.name             # Ver configuración específica
```

Niveles de configuración:

- `--system`: Para todos los usuarios del sistema (/etc/gitconfig)
- `--global`: Para tu usuario (~/.gitconfig)
- `--local`: Para el repositorio actual (.git/config) [predeterminado]

1.4 Configuración de claves SSH para GitHub

1.4.1 Generar una clave SSH

1. Verifica claves existentes:

```
ls -al ~/.ssh
```

Si no hay claves, crea el directorio: `mkdir ~/.ssh`.

2. Genera un par de claves:

```
ssh-keygen -t rsa -b 4096 -C "tu.email@example.com"
```

Acepta el nombre predeterminado y añade una contraseña (opcional).

1.4.2 Añadir la clave a ssh-agent

1. Inicia el agente:

```
eval "$(ssh-agent -s)"
```

2. Añade la clave privada:

```
ssh-add ~/.ssh/id_rsa
```

1.4.3 Vincular la clave a GitHub

1. Copia la clave pública:
   - Linux/Mac: `cat ~/.ssh/id_rsa.pub`
   - Windows: `clip < ~/.ssh/id_rsa.pub`
2. En GitHub, ve a *Settings* > *SSH and GPG keys* > *New SSH key*, pega la clave y guárdala.
3. Prueba la conexión:

```
ssh -T git@github.com
```

Resultado esperado: `Hi tu_usuario! You've successfully authenticated...`

2 Conceptos Fundamentales

2.1 Objetos de Git

Git maneja cuatro tipos de objetos:

1. **Blob**: Contenido de archivos
2. **Tree**: Estructura de directorios (apunta a blobs y otros trees)
3. **Commit**: Instantánea del proyecto (apunta a un tree y commits padres)
4. **Tag**: Referencia nombrada a un commit específico

2.2 Referencias Simbólicas

- **HEAD**: Apunta al commit actual (normalmente la punta de una rama)
- **Rama (branch)**: Puntero móvil a un commit
- **Tag**: Puntero fijo a un commit
- **Remote**: Referencias a ramas en repositorios remotos

2.3 Estados de Archivos

1. **Untracked**: Archivos nuevos que Git no rastrea
2. **Unmodified**: Archivos rastreados sin cambios
3. **Modified**: Archivos rastreados con cambios
4. **Staged**: Archivos preparados para el próximo commit

2.4 SHA-1 y Hashes

Cada objeto en Git tiene un identificador único de 40 caracteres hexadecimales (hash SHA-1). Puedes usar las primeras 7 caracteres para referencias cortas.

3 Comandos Básicos

3.1 Crear e Inicializar Repositorios

Crear nuevo repositorio:

```
git init
git init nombre-proyecto
git init --bare  # Repositorio sin working directory (para servidores)
```

Clonar repositorio existente:

```
git clone <url>
git clone https://github.com/usuario/repositorio.git
git clone <url> <directorio-destino>
```

Solo las últimas n confirmaciones:

```
git clone --depth 1 <url>  # Clone superficial (solo último commit)
git clone -b <rama> <url>  # Clonar rama específica, o,
git clone --branch=nombre-rama <url>
```

3.2 Estado y Diferencias

Ver estado del repositorio:

```
git status
git status -s              # Formato corto
git status --short         # Formato abreviado
git status -b              # Mostrar rama e información de tracking
```

Ver cambios:

```
git diff                   # Cambios no staged (no confirmados)
git diff --staged          # Cambios staged
git diff --cached          # Sinónimo de --staged
git diff HEAD              # Todos los cambios desde último commit
git diff <rama1> <rama2>   # Diferencias entre ramas
git diff <commit1> <commit2>  # Diferencias entre commits
git diff --stat            # Resumen estadístico
git diff --name-only       # Solo nombres de archivos
```

3.3 Añadir Archivos (Staging)

```
git add <archivo>        # Añadir archivo específico
git add .                # Añadir todos los archivos del directorio actual
git add -A               # Añadir todos los archivos del repositorio
git add *.js             # Añadir por patrón
git add -p               # Añadir interactivamente por fragmentos
git add -u               # Añadir solo archivos ya rastreados modificados
```

3.4 Commits

Crear commit:

```
git commit -m "Mensaje del commit"
git commit -am "Mensaje"   # Add + commit (solo tracked files)
git commit                 # Abre editor para mensaje largo
git commit --amend         # Modificar último commit
git commit --amend -m "Nuevo mensaje"  # Cambiar mensaje del último commit
git commit --amend --no-edit  # Añadir cambios sin cambiar mensaje
```

Buenas prácticas para mensajes:

- Primera línea: resumen conciso (50 caracteres o menos)
- Línea en blanco
- Descripción detallada (72 caracteres por línea)
- Usar imperativo: “Añade” no “Añadido”

3.5 Historial

Ver historial:

```
git log
git log --oneline          # Una línea por commit
git log --graph            # Vista gráfica
git log --all              # Todas las ramas
git log --decorate         # Mostrar referencias (ramas, tags)
git log -p                 # Mostrar diferencias (patch)
git log -n 5               # Últimos 5 commits
git log --since="2 weeks"  # Commits de las últimas 2 semanas
git log --author="Edison"  # Commits de un autor
git log --grep="palabra"   # Buscar en mensajes de commit
git log -- archivo.txt     # Historial de un archivo
git log --follow archivo.txt  # Incluir renombrados
```

Formatos personalizados:

```
git log --pretty=format:"%h - %an, %ar : %s"
git log --pretty=format:"%h %s" --graph
git log --graph --oneline --all --decorate
git config --global alias.tree "log --graph --decorate --all --oneline"
git tree
```

Placeholders útiles:

- %h: Hash abreviado
- %H: Hash completo
- %an: Nombre del autor
- %ae: Email del autor
- %ad: Fecha del autor
- %ar: Fecha relativa
- %s: Mensaje del commit

3.6 Deshacer Cambios

Descartar cambios en working directory:

```
git checkout -- <archivo>  # Método antiguo
git restore <archivo>      # Método moderno (Git 2.23+)
git restore .              # Restaurar todos los archivos
```

Quitar archivos del staging area:

```
git reset HEAD <archivo>   # Método antiguo
git restore --staged <archivo>  # Método moderno
git reset HEAD             # Quitar todos
```

Deshacer commits:

```
git reset --soft HEAD~1   # Mantiene cambios en staging
git reset --mixed HEAD~1  # Mantiene cambios en working directory [default]
git reset --hard HEAD~1   # CUIDADO: Elimina todo
git reset <commit>        # Volver a un commit específico
```

Revertir commit (seguro):

```
git revert <commit>        # Crea nuevo commit que deshace cambios
git revert HEAD            # Revertir último commit
git revert --no-commit HEAD~3..HEAD  # Revertir últimos 3 commits
```

3.7 Eliminar y Mover Archivos

```
git rm <archivo>           # Eliminar archivo
git rm --cached <archivo>  # Quitar del tracking pero mantener en disco
git rm -r directorio/      # Eliminar directorio recursivamente
git mv <origen> <destino>  # Renombrar/mover archivo
```

4 Branching y Merging

4.1 Conceptos de Ramas

Una rama en Git es simplemente un puntero móvil a un commit. La rama por defecto se llama `main` (anteriormente `master`).

4.2 Operaciones con Ramas

Crear ramas:

```
git branch <nombre-rama>   # Crear rama
git checkout -b <nombre-rama>  # Crear y cambiar
git switch -c <nombre-rama>    # Método moderno (Git 2.23+)
```

Listar ramas:

```
git branch                 # Ramas locales
git branch -a              # Todas las ramas (locales + remotas)
git branch -r              # Solo ramas remotas
git branch -v              # Con último commit de cada rama
git branch --merged        # Ramas fusionadas con la actual
git branch --no-merged     # Ramas no fusionadas
```

Cambiar de rama:

```
git checkout <rama>        # Método antiguo
git switch <rama>          # Método moderno (Git 2.23+)
git checkout -             # Volver a rama anterior
```

Eliminar ramas:

```
git branch -d <rama>       # Eliminar rama fusionada
git branch -D <rama>       # Forzar eliminación
git push origin --delete <rama>  # Eliminar rama remota
```

Renombrar rama:

```
git branch -m <nuevo-nombre>     # Renombrar rama actual
git branch -m <viejo> <nuevo>    # Renombrar otra rama
```

4.3 Merging (Fusionar)

Merge básico:

```
git merge <rama>           # Fusionar rama en la actual
git merge --no-ff <rama>   # Forzar commit de merge
git merge --squash <rama>  # Fusionar como un solo commit
git merge --abort          # Cancelar merge en conflicto
```

Tipos de merge:

1. **Fast-forward**: Avance directo sin commit de merge
2. **Three-way merge**: Crea commit de merge con dos padres
3. **Squash merge**: Combina todos los commits en uno

4.4 Rebasing

Rebase reescribe el historial moviendo commits a una nueva base.

```
git rebase <rama-base>     # Rebasar rama actual
git rebase main            # Rebasar sobre main
git rebase -i HEAD~3       # Rebase interactivo (últimos 3 commits)
git rebase --continue      # Continuar después de resolver conflictos
git rebase --abort         # Cancelar rebase
git rebase --skip          # Saltar commit actual
```

Rebase interactivo - comandos:

- `pick`: Usar commit
- `reword`: Cambiar mensaje
- `edit`: Editar commit
- `squash`: Fusionar con commit anterior
- `fixup`: Como squash pero descarta mensaje
- `drop`: Eliminar commit

REGLA DE ORO: Nunca hagas rebase de commits públicos/compartidos

4.5 Cherry-pick

Aplicar commits específicos de una rama a otra:

```
git cherry-pick <commit>   # Aplicar un commit
git cherry-pick <commit1> <commit2>   # Varios commits
git cherry-pick <commit1>..<commit2>  # Rango de commits
git cherry-pick --continue # Continuar después de conflictos
git cherry-pick --abort    # Cancelar
```

5 Trabajo Remoto

5.1 Repositorios Remotos

Ver remotos:

```
git remote                 # Listar remotos
git remote -v              # Con URLs
git remote show origin     # Información detallada
```

Añadir remotos:

```
git remote add <nombre> <url>
git remote add origin https://github.com/usuario/repo.git
```

Modificar remotos:

```
git remote rename <viejo> <nuevo>
git remote remove <nombre>
git remote set-url origin <nueva-url>
```

5.2 Fetch, Pull y Push

Fetch (descargar sin fusionar):

```
git fetch                   # Fetch de origin
git fetch origin            # Fetch de origin explícitamente
git fetch --all             # Fetch de todos los remotos
git fetch origin <rama>     # Fetch de rama específica
```

Pull (fetch + merge):

```
git pull                    # Pull de rama actual
git pull origin main        # Pull de rama específica
git pull --rebase           # Pull con rebase en lugar de merge
git pull --rebase origin main
```

Push (enviar cambios):

```
git push                      # Push de rama actual
git push origin main          # Push a rama específica
git push -u origin main       # Push y establecer upstream
git push --all                # Push de todas las ramas
git push --tags               # Push de todos los tags
git push origin --delete <rama>  # Eliminar rama remota
git push --force              #  Forzar push (PELIGROSO)
git push --force-with-lease   # Forzar pero más seguro
```

5.3 Tracking Branches

```
git branch -u origin/main  # Establecer upstream de rama actual
git branch --set-upstream-to=origin/main
git branch -vv             # Ver tracking branches
```

6 Comandos Avanzados

6.1 Stash (Guardar Temporalmente)

Guardar cambios sin hacer commit:

```
git stash                  # Guardar cambios
git stash save "mensaje"   # Con mensaje descriptivo
git stash -u               # Incluir archivos untracked
git stash --all            # Incluir todo (untracked + ignored)
git stash list             # Listar stashes
git stash show             # Ver cambios del último stash
git stash show -p          # Ver diff completo
git stash apply            # Aplicar último stash (lo mantiene)
git stash pop              # Aplicar y eliminar último stash
```

```
git stash apply stash@{2}  # Aplicar stash específico
git stash drop stash@{0}   # Eliminar stash específico
git stash clear            # Eliminar todos los stashes
git stash branch <rama>    # Crear rama desde stash
```

6.2 Tags

Marcar puntos importantes en el historial:
Tags ligeros:

```
git tag v1.0               # Tag ligero
git tag v1.0 <commit>      # Tag en commit específico
```

Tags anotados (recomendado):

```
git tag -a v1.0 -m "Versión 1.0"
git tag -a v1.0 <commit> -m "mensaje"
```

Operaciones con tags:

```
git tag                    # Listar tags
git tag -l "v1.8.*"        # Listar con patrón
git show v1.0              # Ver información del tag
git push origin v1.0       # Push de tag específico
git push origin --tags     # Push de todos los tags
git tag -d v1.0            # Eliminar tag local
git push origin --delete v1.0  # Eliminar tag remoto
git checkout v1.0          # Checkout a tag (detached HEAD)
```

6.3 Buscar y Encontrar

Grep (buscar en archivos):

```
git grep "texto"           # Buscar en working directory
git grep "texto" <commit>  # Buscar en commit específico
git grep -n "texto"        # Con números de línea
git grep --count "texto"   # Contar coincidencias
git grep -i "texto"        # Case insensitive
```

Bisect (búsqueda binaria de bugs):

```
git bisect start           # Iniciar bisect
git bisect bad             # Marcar commit actual como malo
git bisect good <commit>   # Marcar commit bueno conocido
# Git checkout commits intermedios automáticamente
git bisect good            # Marcar como bueno
git bisect bad             # Marcar como malo
git bisect reset           # Terminar bisect
```

Blame (ver quién modificó cada línea):

```
git blame <archivo>        # Ver autor de cada línea
git blame -L 10,20 <archivo>  # Solo líneas 10-20
git blame -e <archivo>     # Mostrar emails
git blame -w <archivo>     # Ignorar cambios de whitespace
```

6.4 Reflog

Registro de todos los cambios a HEAD (incluso commits "perdidos"):

```
git reflog                 # Ver reflog
git reflog show            # Igual que git reflog
git reflog --all           # Reflog de todas las referencias
git reset --hard HEAD@{2}  # Volver a estado anterior
```

6.5 Worktrees

Tener múltiples working directories del mismo repositorio:

```
git worktree add <path> <branch>  # Crear worktree
git worktree list          # Listar worktrees
git worktree remove <path> # Eliminar worktree
git worktree prune         # Limpiar worktrees obsoletos
```

6.6 Submodules

Incluir repositorios dentro de repositorios:

```
git submodule add <url> <path>           # Añadir submodule
git submodule init                       # Inicializar submodules
git submodule update                     # Actualizar submodules
git submodule update --init --recursive  # Init + update recursivo
git clone --recurse-submodules <url>     # Clonar con submodules
```

6.7 Archivos y Bundles

Archive (crear archivo del repositorio):

```
git archive --format=zip --output=proyecto.zip HEAD
git archive --format=tar.gz --output=proyecto.tar.gz main
```

Bundle (repositorio portátil):

```
git bundle create repo.bundle HEAD --all
git clone repo.bundle repo-clonado
git bundle verify repo.bundle
```

7 Resolución de Conflictos

7.1 Identificar Conflictos

Cuando hay conflictos durante merge o rebase:

```
git status                  # Ver archivos con conflictos
git diff                    # Ver conflictos
git diff --ours             # Ver nuestra versión
git diff --theirs           # Ver su versión
```

7.2 Marcadores de Conflicto

```
<<<<<<< HEAD
Tu versión del código
=======
Su versión del código
>>>>>>> rama-a-fusionar
```

7.3 Resolver Conflictos

Manualmente:

1. Editar archivos para resolver conflictos
2. Eliminar marcadores <<<<<<<, =======, >>>>>>>
3. `git add <archivo>` para marcar como resuelto
4. `git commit` o `git rebase --continue`

Con herramientas:

```
git mergetool                          # Abrir herramienta de merge
git mergetool --tool=vimdiff
git config --global merge.tool meld  # Configurar herramienta
```

Estrategias:

```
git merge -X ours <rama>          # Preferir nuestra versión
git merge -X theirs <rama>        # Preferir su versión
git checkout --ours <archivo>     # Tomar nuestra versión
git checkout --theirs <archivo>   # Tomar su versión
```

8 Git Workflows

8.1 Git Flow

Modelo de branching con ramas específicas:

- **main**: Código de producción
- **develop**: Rama de desarrollo
- **feature/***: Nuevas características
- **release/***: Preparación de releases

- **hotfix/***: Correcciones urgentes

Comandos (con extensión git-flow):

```
git flow init
git flow feature start nueva-caracteristica
git flow feature finish nueva-caracteristica
git flow release start 1.0.0
git flow release finish 1.0.0
git flow hotfix start fix-critico
git flow hotfix finish fix-critico
```

8.2 GitHub Flow

Workflow más simple:

1. Crear rama desde `main`
2. Hacer commits
3. Abrir Pull Request
4. Revisión de código
5. Merge a `main`
6. Deploy automático

8.3 Trunk-Based Development

- Una rama principal (`main` o `trunk`)
- Commits frecuentes y pequeños
- Feature flags para ocultar trabajo en progreso
- CI/CD robusto

9 Ejemplos de uso diario de Git

Flujos de trabajo más comunes que uso **todos los días** en diferentes tipos de proyectos y entornos (individuales, equipos pequeños, medianos y open-source).

9.1 Flujo individual / Freelance / Proyectos personales

```
# Al empezar el día
git pull origin main  # Siempre sincronizo primero
git status            # Muestra el estado actual del repositorio.

# Trabajo en una nueva funcionalidad
git switch -c feat/login-social   # Creo rama con convención clara

# ... desarrollo varias horas ...
git add .
git commit -m "feat: implementar login con Google y GitHub"

# Pequeña corrección rápida
```

```
git add src/components/LoginForm.tsx
git commit -m "fix: corregir validación de email en formulario"

# Al final del día o antes de push
git branch -M main # Muestra las ramas existentes.
git switch main
git pull                 # Vuelvo a sincronizar por si alguien más empujó
git switch feat/login-social
git rebase main          # Mantengo mi rama actualizada (rebase limpio)
git switch main
git merge --ff-only feat/login-social   # Fast-forward si es posible
git push -u origin main  # Envía los cambios al repositorio remoto.
git branch -d feat/login-social   # Elimino la rama ya que está integrada

# Opcional: commit squash al final (muy común en proyectos pequeños)
git merge --squash feat/login-social
git commit -m "feat: login con proveedores sociales + validaciones"
git push
```

Ventajas: Muy rápido, historial relativamente limpio, poco overhead.

9.2 Flujo típico de equipo mediano/pequeño con Pull Requests

```
# Inicio de jornada / después de stand-up
git fetch --prune          # Limpio referencias remotas obsoletas
git switch main
git pull

# Nueva tarea (ej: ticket #456 - mejorar rendimiento de dashboard)
git switch -c feature/456-dashboard-perf

# Desarrollo normal (varios commits)
git commit -m "feat(dashboard): lazy loading de gráficos pesados"
git commit -m "refactor: extraer hook useChartData"

# Antes de terminar el día
git rebase -i origin/main    # O rebase main si ya está actualizado
# Squash/reword si quiero limpiar commits antes de PR

# Subo y creo Pull Request
git push -u origin feature/456-dashboard-perf

# En GitHub/GitLab:
#   → Creo PR → añado reviewers → espero revisión
#   → Después de aprobar y pasar CI/CD:
#     Merge squash / merge commit /
#     rebase & merge (depende de la política del equipo)
```

```
# Una vez mergeado (normalmente por el reviewer o bot)
git switch main
git pull
git fetch --prune
git branch -d feature/456-dashboard-perf   # Limpio localmente
```

Variante muy común en 2025:
Merge squash → un solo commit limpio en `main` con el título del PR

9.3 Corrección rápida de bug en producción

```
# Opción más rápida y segura (recomendada)
git switch main
git pull
git switch -c hotfix/789-boton-pago-falla

# Arreglo rápido (1-3 commits)
git commit -m "fix: corregir validación de tarjeta en checkout"

git push -u origin hotfix/789-boton-pago-falla

# → Crear PR rápido hacia main
# → Revisión exprés (o auto-merge si es crítico)
# → Merge (preferiblemente squash o rebase)

# Después del deploy
git switch main
git pull
git tag hotfix-2025-12-30-boton-pago   # O v1.2.3-hotfix si usas semver
git push --tags
```

9.4 Flujo cuando trabajas en varias tareas al mismo tiempo

```
# Estoy en medio de feature/cuenta-premium
git status

# → Aparece bug urgente
git stash push -m "WIP premium checkout"

# Arreglo rápido
git switch -c hotfix/791-api-timeout
# ... fix ...
git commit -m "fix: aumentar timeout en llamada a /reports"
git push -u origin hotfix/791-api-timeout
# → PR rápido → merge
```

```
# Vuelvo a lo mío
git switch feature/cuenta-premium
git stash pop
# Continúo donde estaba...
```

9.5 Mini-resumen de comandos que más se usan en la práctica diaria

Acción	Comando más común en 2025	Frecuencia
Sincronizar al empezar	`git pull` / `git fetch --prune && git pull`	
Crear rama de tarea	`git switch -c feat/ticket-xxx-nombre`	
Commit atómico	`git commit -m "tipo: descripción corta"`	
Actualizar rama antes de PR	`git rebase main` o `git merge main`	
Subir rama nueva	`git push -u origin nombre-rama`	
Limpiar después de merge	`git branch -d rama-antigua`	
Guardar trabajo temporal	`git stash push -m "..."` / `git stash pop`	
Ver estado rápido	`git status -sb` o alias `gs`	
Historial bonito	`git log --oneline --graph --decorate --all`	
Comparar con main	`git diff main...`	

Recomendación final para tu día a día
Adopta el hábito de:

1. Siempre `pull` al empezar
2. Trabajar en ramas cortas y con nombres claros
3. Commits pequeños y descriptivos (convención **Conventional Commits** es la más usada actualmente)
4. Actualizar tu rama frecuentemente (`rebase` o `merge main`)
5. Eliminar ramas una vez integradas

10 Mejores Prácticas

10.1 Commits

1. **Commits atómicos**: Un commit = un cambio lógico
2. **Mensajes descriptivos**: Explica el "qué" y el "por qué"
3. **Commits frecuentes**: Mejor muchos commits pequeños que pocos grandes
4. **No commitear archivos generados**: Usar .gitignore
5. **Revisar antes de commitear**: `git diff --staged`

10.2 Branching

1. **Nombres descriptivos**: `feature/nueva-funcionalidad`, `bugfix/issue-123`
2. **Ramas de vida corta**: Fusionar frecuentemente

3. **Mantener ramas actualizadas**: Hacer merge/rebase regular de main
4. **Eliminar ramas fusionadas**: Mantener repositorio limpio
5. **Proteger rama principal**: Requerir pull requests y revisiones

10.3 Colaboración

1. **Pull antes de push**: Evitar conflictos
2. **Revisar código**: Pull requests con revisiones
3. **No reescribir historial público**: Evitar rebase/amend de commits pusheados
4. **Comunicación**: Documentar decisiones importantes
5. **CI/CD**: Automatizar tests y despliegues

10.4 .gitignore

Ejemplos comunes:

```
# Node.js
node_modules/
npm-debug.log
.env

# Python
__pycache__/
*.py[cod]
venv/
.pytest_cache/

# IDE
.vscode/
.idea/
*.swp

# OS
.DS_Store
Thumbs.db

# Build
dist/
build/
*.log
```

Comandos útiles:

```
git config --global core.excludesfile ~/.gitignore_global
git check-ignore -v <archivo>  # Ver por qué archivo es ignorado
```

11 Comandos de Bajo Nivel (Plumbing)

Comandos internos de Git para operaciones avanzadas:

11.1 Inspección de Objetos

```
git cat-file -p <hash>     # Ver contenido de objeto
git cat-file -t <hash>     # Ver tipo de objeto
git cat-file -s <hash>     # Ver tamaño de objeto
git ls-tree <tree-hash>    # Listar contenido de tree
git ls-files               # Listar archivos en index
git ls-files -s            # Con información detallada
git ls-remote <remote>     # Listar referencias remotas
```

11.2 Manipulación del Index

```
git update-index --add <archivo>     # Añadir al index
git update-index --remove <archivo>  # Quitar del index
git read-tree <tree>       # Leer tree al index
git write-tree             # Crear tree desde index
```

```
git show-ref               # Mostrar todas las referencias
git update-ref refs/heads/main <commit>  # Actualizar referencia
git symbolic-ref HEAD      # Ver a qué apunta HEAD
git for-each-ref           # Iterar sobre referencias
```

11.3 Objetos y Hashes

```
git hash-object <archivo>     # Calcular hash de archivo
git hash-object -w <archivo>  # Escribir objeto
git rev-parse HEAD            # Convertir referencia a hash
git rev-list HEAD             # Listar commits alcanzables
```

12 Hooks y Automatización

12.1 Git Hooks

Scripts que se ejecutan automáticamente en eventos de Git.
Ubicación: `.git/hooks/`
Hooks comunes:
Client-side:

- `pre-commit`: Antes de crear commit
- `prepare-commit-msg`: Antes de abrir editor de commit
- `commit-msg`: Validar mensaje de commit
- `post-commit`: Después de crear commit
- `pre-push`: Antes de hacer push

Server-side:

- `pre-receive`: Antes de aceptar push

- `update`: Por cada rama actualizada
- `post-receive`: Después de aceptar push

Ejemplo pre-commit (tests):

```
#!/bin/sh
# .git/hooks/pre-commit

npm test
if [ $? -ne 0 ]; then
    echo "Tests fallaron. Commit cancelado."
    exit 1
fi
```

Ejemplo commit-msg (validar formato):

```
#!/bin/sh
# .git/hooks/commit-msg

commit_msg=$(cat "$1")
pattern="^(feat|fix|docs|style|refactor|test|chore): .+"

if ! echo "$commit_msg" | grep -qE "$pattern"; then
    echo "Error: El mensaje debe seguir el formato:"
    echo "(feat|fix|docs|style|refactor|test|chore): descripción"
    exit 1
fi
```

12.2 Aliases

Crear comandos personalizados:

```
# Configurar aliases
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.st status
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD'
git config --global alias.visual 'log --oneline --graph --all --decorate'
git config --global alias.amend 'commit --amend --no-edit'

# Usar aliases
git co main
git visual
```

Aliases útiles:

```
git config --global alias.lg "log --graph --pretty=format:'%Cred%h%Creset \
-%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"
git config --global alias.undo 'reset HEAD~1 --mixed'
git config --global alias.contributors 'shortlog -sn'
```

13 Troubleshooting

13.1 Problemas Comunes

Descartar todos los cambios locales:

```
git reset --hard HEAD
git clean -fd  # Eliminar archivos untracked
```

Recuperar commit eliminado:

```
git reflog  # Encontrar hash del commit
git checkout <hash>
git branch recovery <hash>  # Crear rama desde commit
```

Cambiar último commit:

```
git commit --amend  # Modificar mensaje o añadir archivos
```

Mover commits a otra rama:

```
git checkout rama-correcta
git cherry-pick <commit>
git checkout rama-incorrecta
git reset --hard HEAD~1  # Eliminar de rama incorrecta
```

Dividir commit grande:

```
git reset HEAD~1  # Deshacer commit pero mantener cambios
git add <archivo1>
git commit -m "Primera parte"
git add <archivo2>
git commit -m "Segunda parte"
```

Sincronizar fork con upstream:

```
git remote add upstream <url-original>
git fetch upstream
git checkout main
git merge upstream/main
git push origin main
```

Resolver “detached HEAD”:

```
git branch temp-branch  # Crear rama desde HEAD actual
git checkout main
git merge temp-branch
```

Limpiar referencias obsoletas:

```
git remote prune origin  # Eliminar referencias remotas obsoletas
git fetch --prune        # Fetch y prune simultáneamente
```

Comprimir repositorio:

```
git gc               # Garbage collection
git gc --aggressive  # Más agresivo
git prune            # Eliminar objetos inalcanzables
```

Ver archivos grandes:

```
git rev-list --objects --all \
  | git cat-file --batch-check='%(objecttype) %(objectname) %(objectsize) %(rest)' \
  | sed -n 's/^blob //p' \
  | sort --numeric-sort --key=2 \
  | tail -n 10
```

Eliminar archivo del historial:

```
git filter-branch --tree-filter 'rm -f archivo-sensible' HEAD
# O usar git-filter-repo (más rápido)
git filter-repo --path archivo-sensible --invert-paths
```

14 Referencia Rápida

14.1 Sintaxis de Referencias

```
HEAD                  # Commit actual
HEAD^                 # Padre de HEAD (HEAD~1)
HEAD^^                # Abuelo de HEAD (HEAD~2)
HEAD~3                # 3 commits antes de HEAD
main^2                # Segundo padre de main (en merge)
<branch>@{yesterday}  # Posición ayer
HEAD@{5}              # 5 movimientos atrás en reflog
```

14.2 Especificar Rangos

```
<commit1>..<commit2>   # Commits alcanzables desde commit2 pero no desde commit1
<commit1>...<commit2>  # Commits alcanzables desde cualquiera pero no desde ambos
<branch>^@             # Todos los padres de branch
<commit>^!             # El commit pero no sus padres
```

14.3 Variables de Entorno

```
GIT_AUTHOR_NAME        # Nombre del autor
GIT_AUTHOR_EMAIL       # Email del autor
GIT_COMMITTER_NAME     # Nombre del committer
GIT_COMMITTER_EMAIL    # Email del committer
GIT_EDITOR             # Editor predeterminado
GIT_PAGER              # Pager para output
GIT_TRACE              # Activar tracing
```

14.4 Configuración Útil

```
# Colorear output
git config --global color.ui auto

# Guardar credenciales
git config --global credential.helper cache
git config --global credential.helper 'cache --timeout=3600'

# Autocorrección de comandos
git config --global help.autocorrect 10

# Rebase por defecto al hacer pull
git config --global pull.rebase true

# Prune automático
git config --global fetch.prune true

# Diff mejorado
git config --global diff.algorithm histogram

# Rerere (reuse recorded resolution)
git config --global rerere.enabled true
```

14.5 Comandos de Emergencia

```
# Deshacer TODO y volver limpio
git reset --hard HEAD
git clean -fd

# Recuperar trabajo después de reset --hard
git reflog
git reset --hard <hash>

# Salir de cualquier operación en progreso
git merge --abort
```

```
git rebase --abort
git cherry-pick --abort

# Verificar integridad del repositorio
git fsck --full

# Reparar repositorio corrupto
git fsck --full --no-dangling
git gc --aggressive --prune=now
```

14.6 Atajos de Teclado (Shell)

Añadir a `~/.bashrc` o `~/.zshrc`:

```
alias g='git'
alias gs='git status'
alias ga='git add'
alias gc='git commit'
alias gp='git push'
alias gl='git pull'
alias gd='git diff'
alias gco='git checkout'
alias gb='git branch'
alias glg='git log --graph --oneline --all --decorate'
```

14.7 Formato de Commit Convencional

```
<tipo>(<ámbito>): <descripción>

<cuerpo>

<pie>
```

Tipos:

- `feat`: Nueva funcionalidad
- `fix`: Corrección de bug
- `docs`: Documentación
- `style`: Formato (no afecta código)
- `refactor`: Refactorización
- `test`: Tests
- `chore`: Tareas de mantenimiento
- `perf`: Mejora de rendimiento

Ejemplo:

```
feat(auth): añadir autenticación con OAuth2

Implementa login con Google y GitHub usando OAuth2.
```

```
Incluye manejo de tokens y refresh automático.

Closes #123
```

15 Recursos Adicionales

15.1 Documentación Oficial

- **Manual de Git**: `man git` o https://git-scm.com/docs
- **Libro Pro Git**: https://git-scm.com/book/es/v2
- **Git Reference**: https://git-scm.com/docs

15.2 Tutoriales y Cursos

- **Learn Git Branching**: https://learngitbranching.js.org
- **Git Tutorial de Atlassian**: https://www.atlassian.com/git/tutorials
- **GitHub Learning Lab**: https://lab.github.com

15.3 Herramientas

- **GitKraken**: Cliente visual multiplataforma
- **SourceTree**: Cliente visual de Atlassian
- **Fork**: Cliente Git para Mac y Windows
- **lazygit**: Cliente TUI (terminal) simple y potente
- **tig**: Navegador de texto para repositorios Git

15.4 Extensiones

- **git-extras**: Comandos útiles adicionales
- **git-flow**: Extensión para Git Flow workflow
- **git-lfs**: Large File Storage
- **git-filter-repo**: Reescritura avanzada de historial

16 Glosario

Blob: Objeto que contiene el contenido de un archivo.
Branch (Rama): Puntero móvil a un commit que representa una línea de desarrollo.
Checkout: Cambiar el working directory a un commit, rama o tag específico.
Clone: Crear una copia local de un repositorio remoto.
Commit: Instantánea del proyecto en un momento específico.
Conflict (Conflicto): Situación donde Git no puede fusionar cambios automáticamente.
Detached HEAD: Estado donde HEAD apunta directamente a un commit en lugar de a una rama.
Diff: Diferencias entre dos versiones de archivos.
Fast-forward: Tipo de merge donde simplemente se avanza el puntero de la rama.
Fetch: Descargar objetos y referencias desde un repositorio remoto sin fusionar.
Fork: Copia de un repositorio en tu propia cuenta.
HEAD: Referencia al commit actual.
Index: Área de staging donde se preparan cambios para el próximo commit.
Merge: Fusionar cambios de una rama a otra.
Origin: Nombre por defecto del repositorio remoto principal.
Pull: Fetch + Merge en un solo comando.

Pull Request: Solicitud para fusionar cambios (terminología de GitHub).
Push: Enviar commits locales a un repositorio remoto.
Rebase: Mover o combinar commits a una nueva base.
Remote: Repositorio alojado en otro lugar (servidor).
Repository (Repositorio): Colección de commits, ramas y configuración de un proyecto.
SHA-1: Algoritmo hash usado para identificar objetos de Git.
Staging Area: Sinónimo de Index.
Stash: Guardar temporalmente cambios sin hacer commit.
Tag: Referencia permanente a un commit específico.
Tree: Objeto que representa un directorio.
Upstream: Rama remota que una rama local rastrea.
Working Directory (Directorio de Trabajo): Directorio actual con archivos del proyecto.

17 Conclusión

Git es una herramienta poderosa y flexible que requiere práctica para dominar. Esta guía cubre desde conceptos básicos hasta técnicas avanzadas, pero la mejor forma de aprender es usándolo en proyectos reales.

Consejos finales:

1. Practica regularmente
2. Experimenta en repositorios de prueba
3. Lee los mensajes de error (son informativos)
4. Usa `git help <comando>` cuando tengas dudas
5. No temas equivocarte (casi todo se puede deshacer)

Comandos más importantes para recordar:

- `echo "# Léeme" >> README.md`: Crea un archivo README.md con el texto "# Léeme".
- `git init`: Inicia un nuevo repositorio en el directorio actual.
- `git status` - Siempre saber dónde estás
- `git add` - Preparar cambios
- `git commit -m "Primer commit"` - Guardar cambios
- `git branch -M main`: Muestra las ramas existentes.
- `git push -u origin main` - Compartir cambios
- `git pull` - Obtener cambios
- `git log` - Ver historial de commits
- `git diff` - Ver diferencias
- `git checkout [branch]`: Cambia a una rama específica.
- `git merge [branch]`: Fusiona una rama con la actual.

18 Publicaciones Similares

Si te interesó este artículo, te recomendamos que explores otros blogs y recursos relacionados que pueden ampliar tus conocimientos. Aquí te dejo algunas sugerencias:

1. Comandos De Informacion Windows

2. Adb
3. Limpieza Y Optimizacion De Pc
4. Usando Apk En Windown 11
5. Gestionar Versiones De Jdk En Kubuntu
6. Instalar Tor Browser
7. Crear Enlaces Duros O Hard Link En Linux
8. Comandos Vim
9. Guia De Git Y Github
10. 00 Primeros Pasos En Linux
11. 01 Introduccion Linux
12. 02 Distribuciones Linux
13. 03 Instalacion Linux
14. 04 Administracion Particiones Volumenes
15. Atajos De Teclado Y Comandos Para Usar Vim
16. Instalando Specitify
17. Gestiona Tus Dotfiles Con Gnu Stow

Esperamos que encuentres estas publicaciones igualmente interesantes y útiles. ¡Disfruta de la lectura!