



Introducción a GIT

Sistema de Control de Versiones

Autor: Steven Ayala

¿Qué es un Sistema de Control de Versiones?

Un Sistema de Control de Versiones (SCV) es una herramienta fundamental para gestionar proyectos de desarrollo, permitiendo un control completo sobre la evolución del código y facilitando la colaboración entre equipos.



Gestión de Cambios

Registra quién y cuándo hace cada modificación en el proyecto



Versiones Previas

Permite volver a estados anteriores del proyecto cuando sea necesario



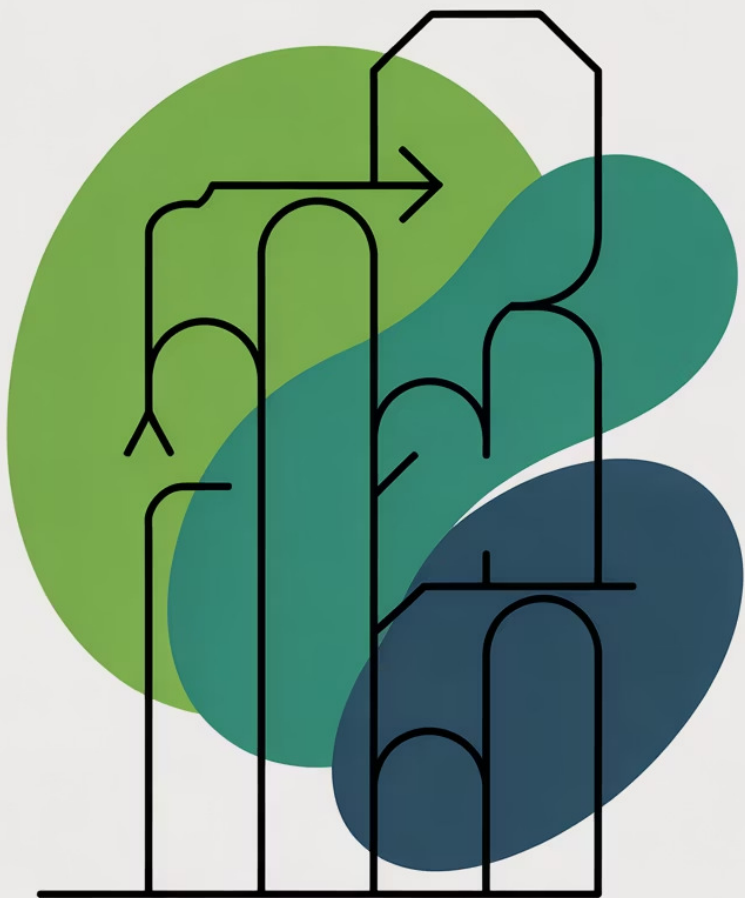
Múltiples Ramas

Facilita el desarrollo paralelo de diferentes funcionalidades



Colaboración

Permite que múltiples usuarios trabajen simultáneamente



¿Para qué sirve un SCV?



Control de historial

Mantiene un registro completo de todos los cambios realizados



Seguridad ante errores

Protege el proyecto permitiendo revertir cambios problemáticos



Trabajo colaborativo

Facilita la coordinación entre múltiples desarrolladores



Múltiples versiones

Soporta diferentes versiones del proyecto simultáneamente



Ramas y fusiones

Permite desarrollar funcionalidades de forma independiente



¿Qué es Git?

Git es un sistema de control de versiones distribuido creado por Linus Torvalds, diseñado para manejar proyectos de cualquier tamaño con velocidad y eficiencia excepcionales.

Sistema Distribuido

Cada usuario tiene una copia completa del historial del proyecto, no solo la versión más reciente

Creado por Linus Torvalds

Desarrollado por el creador de Linux para gestionar el kernel del sistema operativo

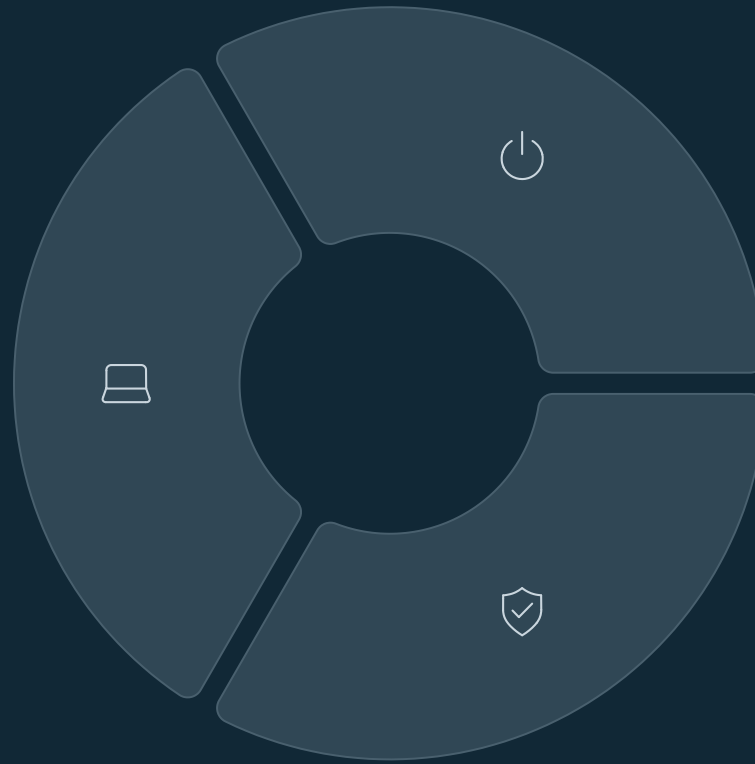
Rápido y Eficiente

Optimizado para operaciones veloces incluso en proyectos de gran escala

Arquitectura Distribuida

A diferencia de los sistemas centralizados, Git utiliza una arquitectura distribuida que ofrece ventajas significativas en términos de flexibilidad, seguridad y rendimiento.

Copia Completa
Cada usuario posee una copia completa del repositorio con todo su historial



Trabajo Offline

Se puede trabajar sin conexión a internet y sincronizar después

Redundancia

Mayor seguridad gracias a múltiples copias del proyecto

Configuración de Git

Antes de comenzar a usar Git, es necesario configurar tu identidad y preferencias. Estos ajustes se aplican globalmente a todos tus repositorios.

```
git config --global user.name "Tu Nombre"  
git config --global user.email "tu-correo"  
git config --global color.ui auto  
git config --global merge.conflictstyle diff3  
git config --list
```

01

Configurar nombre de usuario

Identifica quién realiza los cambios

03

Activar colores en la interfaz

Mejora la legibilidad de los comandos

02

Configurar correo electrónico

Asocia commits con tu identidad

04

Verificar configuración

Revisa todos los ajustes aplicados

Creación de Repositorios

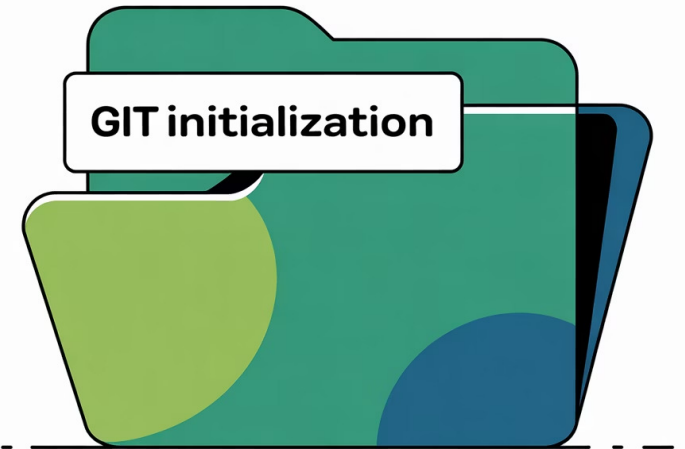
git init

El comando `git init` crea un nuevo repositorio Git en el directorio actual. Este comando inicializa una carpeta oculta llamada `.git` que contiene toda la estructura necesaria para el control de versiones.

```
git init
```

Al ejecutar este comando, Git crea:

- Carpeta `.git` con el historial del proyecto
- Estructura de objetos para almacenar cambios
- Configuración inicial del repositorio
- Referencias a ramas y commits



Clonar Repositorios

git clone

El comando `git clone` crea una copia completa e independiente de un repositorio remoto existente. Esta copia incluye todo el historial de cambios, ramas y configuraciones.



Repositorio Remoto

Proyecto alojado en servidor



Clonación

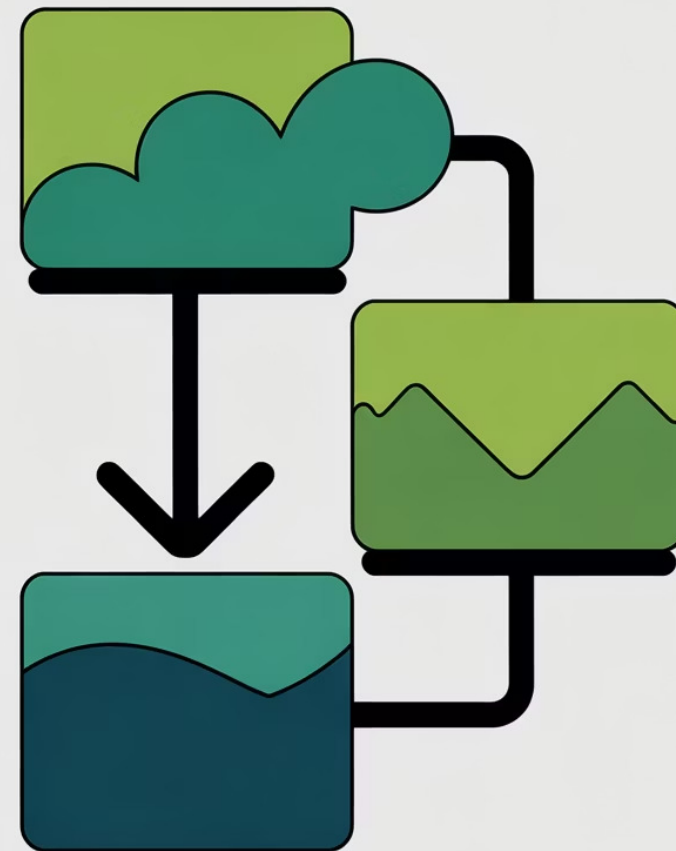
Descarga completa del historial

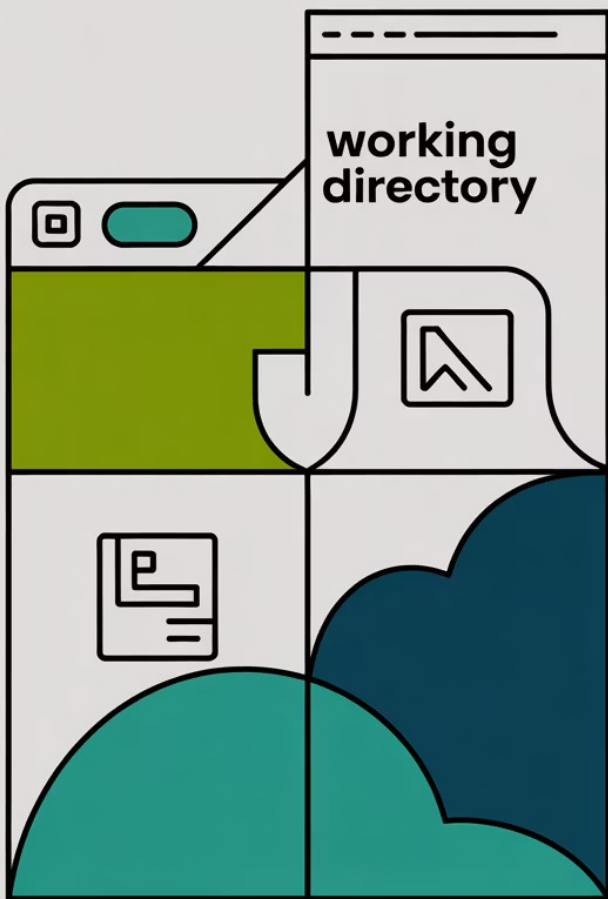


Copia Local

Repositorio independiente en tu máquina

```
git clone https://github.com/usuario/repositorio.git
```





Estados de Git

Git maneja tres estados principales para los archivos en tu proyecto. Comprender estos estados es fundamental para trabajar eficientemente con el sistema de control de versiones.



Directorio de Trabajo

Archivos modificados pero no preparados para commit. Es donde realizas cambios activamente en tu proyecto.



Staging Area

Zona temporal donde preparas los cambios que quieres incluir en el próximo commit. También llamada "índice".



Repositorio

Commits definitivos guardados en el historial. Los cambios están permanentemente registrados en la base de datos de Git.

Añadir Cambios a Staging

git add

El comando `git add` mueve archivos del directorio de trabajo al área de staging, preparándolos para ser incluidos en el próximo commit.

Comandos Principales

```
git add <archivo>  
git add <directorio>  
git add .
```

El punto (.) añade todos los archivos modificados del directorio actual y sus subdirectorios.

Casos de Uso

- Añadir archivo específico modificado
- Añadir todos los archivos de una carpeta
- Añadir todos los cambios del proyecto
- Preparar archivos nuevos para seguimiento



Confirmar Cambios

git commit

El comando `git commit` crea una nueva versión del proyecto, guardando permanentemente los cambios del staging area en el historial del repositorio.

1

Commit con mensaje

```
git commit -m "mensaje"
```

Crea un nuevo commit con un mensaje descriptivo de los cambios realizados

2

Modificar último commit

```
git commit --amend
```

Permite editar el mensaje del último commit o añadir cambios olvidados

📌 **Buenas prácticas:** Escribe mensajes de commit claros y descriptivos que expliquen qué cambios se realizaron y por qué.

Estructura Interna del Historial

Git almacena el historial del proyecto utilizando una estructura de objetos interconectados. Cada commit contiene referencias a los archivos y su contenido en ese momento específico.

Commit

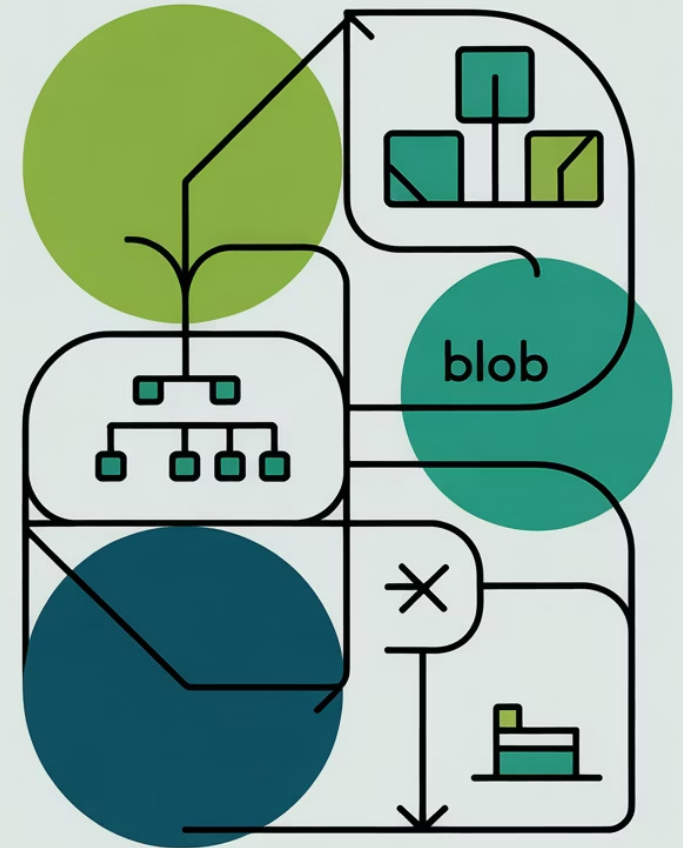
Contiene información del autor, fecha, mensaje descriptivo y referencia al árbol de archivos

Tree

Representa la estructura de directorios y archivos del proyecto en ese commit

Blob

Almacena el contenido comprimido de cada archivo individual



Referenciar Commits

Git ofrece múltiples formas de referenciar commits específicos en el historial. Estas referencias facilitan la navegación y manipulación del historial del proyecto.



Hash SHA-1

Identificador único de 40 caracteres hexadecimales que identifica cada commit de forma inequívoca



HEAD

Referencia especial que apunta al último commit de la rama actual en la que estás trabajando



HEAD~1

Referencia al commit inmediatamente anterior al HEAD actual



HEAD~2

Referencia al commit dos posiciones antes del HEAD actual

Estado del Repositorio

git status

El comando `git status` es una de las herramientas más útiles de Git. Muestra el estado actual del repositorio, incluyendo cambios no confirmados, archivos en staging y archivos nuevos sin seguimiento.

```
git status
```

Cambios No Confirmados

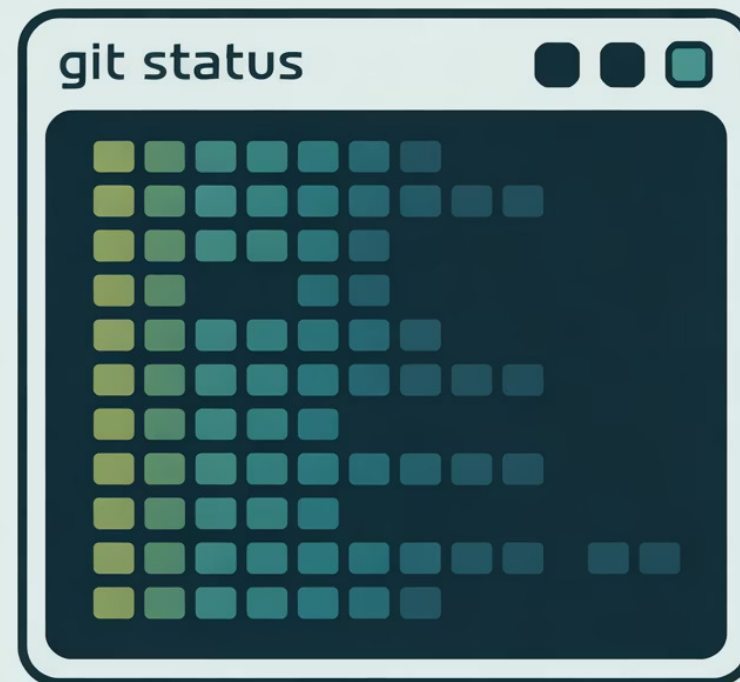
Archivos modificados en el directorio de trabajo

Archivos en Staging

Cambios preparados para el próximo commit

Archivos Nuevos

Archivos sin seguimiento por Git





Historial del Repositorio

git log

El comando `git log` muestra el historial de commits del repositorio, permitiendo revisar todos los cambios realizados a lo largo del tiempo.



Vista Estándar

```
git log
```

Muestra información completa de cada commit



Vista Compacta

```
git log --oneline
```

Muestra cada commit en una sola línea



Vista Gráfica

```
git log --graph
```

Visualiza el historial como un grafo de ramas

Inspeccionar Commits

git show

El comando `git show` permite examinar en detalle un commit específico, mostrando las diferencias introducidas, información del autor, fecha y mensaje descriptivo.

```
git show <commit>
```

- **Información del Commit**

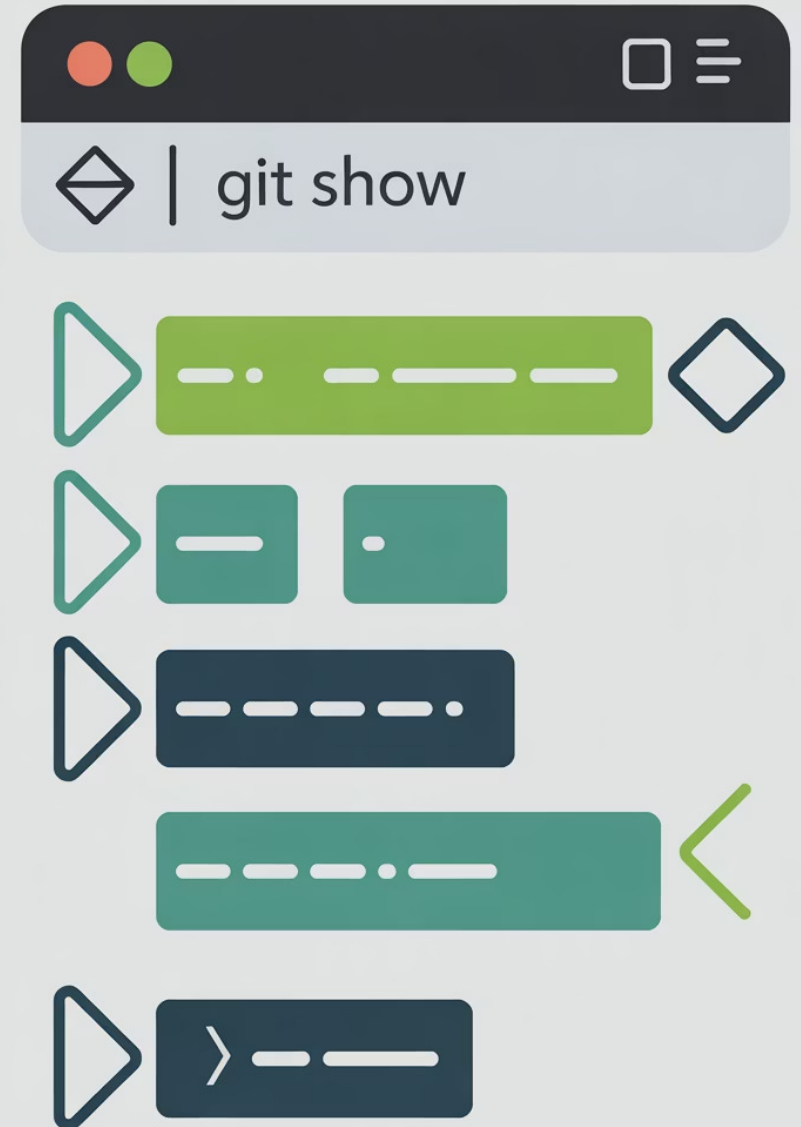
Muestra autor, fecha y mensaje descriptivo del commit seleccionado

- **Diferencias de Código**

Presenta las líneas añadidas y eliminadas en cada archivo modificado

- **Archivos Afectados**

Lista todos los archivos que fueron modificados en ese commit



Historial por Archivo

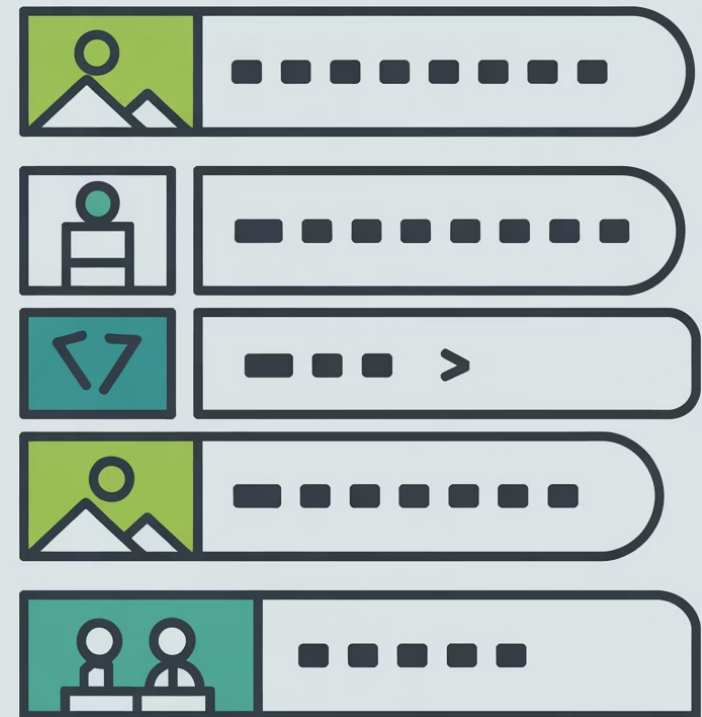
git annotate

El comando `git annotate` muestra el historial de cambios línea por línea de un archivo específico, indicando quién modificó cada línea, cuándo lo hizo y en qué commit.

```
git annotate <archivo>
```

Esta herramienta es especialmente útil para:

- Identificar el autor de código específico
- Rastrear cuándo se introdujo un cambio
- Comprender la evolución de un archivo
- Investigar el origen de bugs



Comparación de Versiones

git diff

El comando `git diff` muestra las diferencias entre distintas versiones de archivos, permitiendo comparar cambios en el directorio de trabajo, staging area y commits.



Diferencias en directorio de trabajo

```
git diff
```

Compara archivos modificados con la última versión confirmada



Diferencias en staging

```
git diff --cached
```

Muestra cambios preparados para el próximo commit



Diferencias con HEAD

```
git diff HEAD
```

Compara todos los cambios con el último commit

Deshacer Cambios: Checkout

git checkout

El comando `git checkout HEAD -- <archivo>` restaura un archivo específico a su estado en el último commit, descartando cualquier modificación no confirmada en el directorio de trabajo.

```
git checkout HEAD -- <archivo>
```

📄 ⚠️ **Advertencia:** Este comando descarta permanentemente los cambios no guardados en el archivo especificado. Úsalo con precaución.

Antes del Checkout

Archivo con modificaciones no confirmadas en el directorio de trabajo

Después del Checkout

Archivo restaurado a la versión del último commit

Quitar Archivos del Staging

git reset

El comando `git reset <archivo>` remueve archivos del área de staging, devolviéndolos al directorio de trabajo. Los cambios en los archivos se mantienen intactos, simplemente dejan de estar preparados para el commit.

```
git reset <archivo>
```

1

Staging Area

Archivo preparado para commit

2

git reset

Comando de remoción

3

Directorio de Trabajo

Archivo con cambios preservados

Este comando es útil cuando añadiste archivos al staging por error y quieres reorganizar qué cambios incluir en el próximo commit.

Eliminar Cambios Completamente

`git reset --hard`



CUIDADO

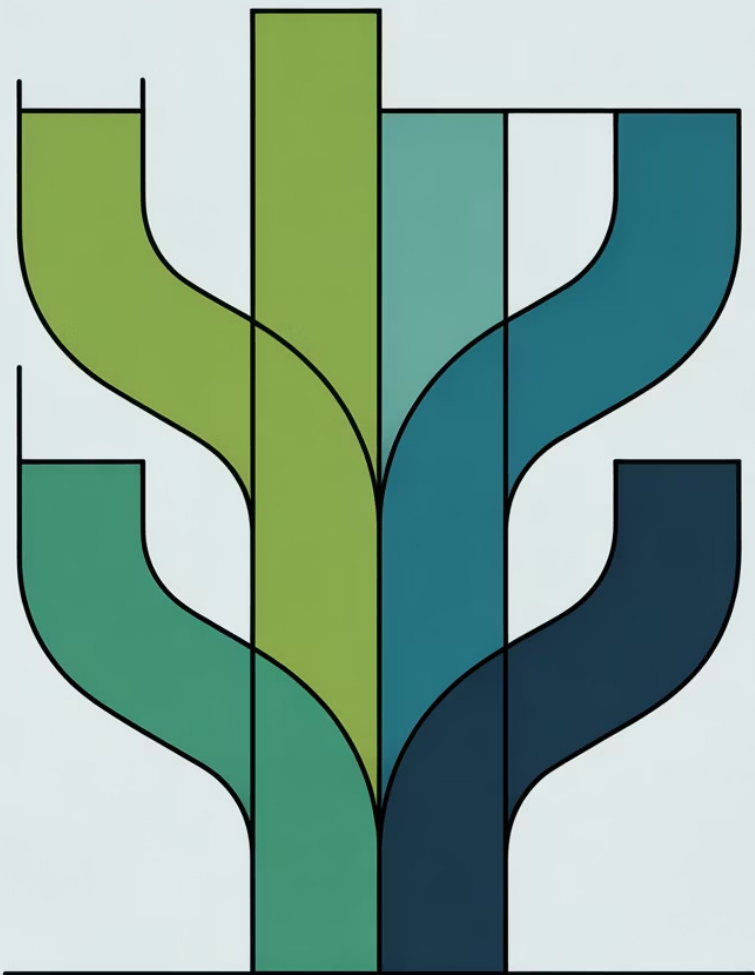
El comando `git reset --hard` elimina completamente todos los cambios no guardados, tanto del staging area como del directorio de trabajo. Esta operación es **irreversible**.

```
git reset --hard
```

“

Advertencia Crítica: Este comando descarta permanentemente todos los cambios no confirmados. No hay forma de recuperarlos después de ejecutarlo. Úsalo solo cuando estés completamente seguro de que quieres eliminar todo el trabajo no guardado.

”



Ramas en Git

Las ramas son una de las características más poderosas de Git. Permiten desarrollar nuevas funcionalidades, experimentar con ideas o corregir bugs sin afectar la rama principal del proyecto.

Desarrollo Paralelo

Trabaja en múltiples funcionalidades simultáneamente sin interferencias

Rama Principal

La rama principal se llama master o main y contiene el código estable

Experimentación Segura

Prueba ideas sin riesgo de romper el código funcional

Crear y Cambiar Ramas

Git ofrece comandos sencillos para crear nuevas ramas y cambiar entre ellas, permitiendo una gestión flexible del flujo de trabajo.

01

Crear nueva rama

```
git branch <nombre-rama>
```

Crea una nueva rama pero no cambia a ella

02

Cambiar a una rama

```
git checkout <nombre-rama>
```

Cambia el directorio de trabajo a la rama especificada

03

Crear y cambiar simultáneamente

```
git checkout -b <nombre-rama>
```

Crea una nueva rama y cambia a ella en un solo comando



Visualizar Ramas

Git proporciona herramientas para visualizar la estructura de ramas del repositorio, facilitando la comprensión del flujo de trabajo y las relaciones entre diferentes líneas de desarrollo.

Listar Ramas

```
git branch
```

Muestra todas las ramas locales. La rama actual se marca con un asterisco (*).

Vista Gráfica

```
git log --graph --oneline --all
```

Visualiza el historial de todas las ramas como un grafo ASCII.

Fusionar Ramas

git merge

El comando `git merge` integra los cambios de una rama en la rama actual, combinando el historial de ambas líneas de desarrollo.

```
git merge <nombre-rama>
```



Git intentará fusionar automáticamente los cambios. Si hay conflictos, deberás resolverlos manualmente antes de completar la fusión.



Conflictos de Fusión

Los conflictos ocurren cuando dos ramas modifican la misma parte de un archivo de formas diferentes. Git no puede decidir automáticamente qué cambios mantener, por lo que requiere intervención manual.

¿Cuándo Ocurren?

Cuando dos ramas editan las mismas líneas de un archivo o cuando un archivo es modificado en una rama y eliminado en otra

Resolución Manual

Debes editar los archivos en conflicto, eligiendo qué cambios mantener y eliminando los marcadores de conflicto que Git añade

Herramientas de Ayuda

Utiliza herramientas como KDiff3 o meld para visualizar y resolver conflictos de forma más intuitiva

Reorganizar Historial

git rebase

El comando `git rebase` replica los cambios de una rama sobre otra, creando un historial lineal sin bifurcaciones visuales. Es una alternativa a `git merge` que produce un historial más limpio.

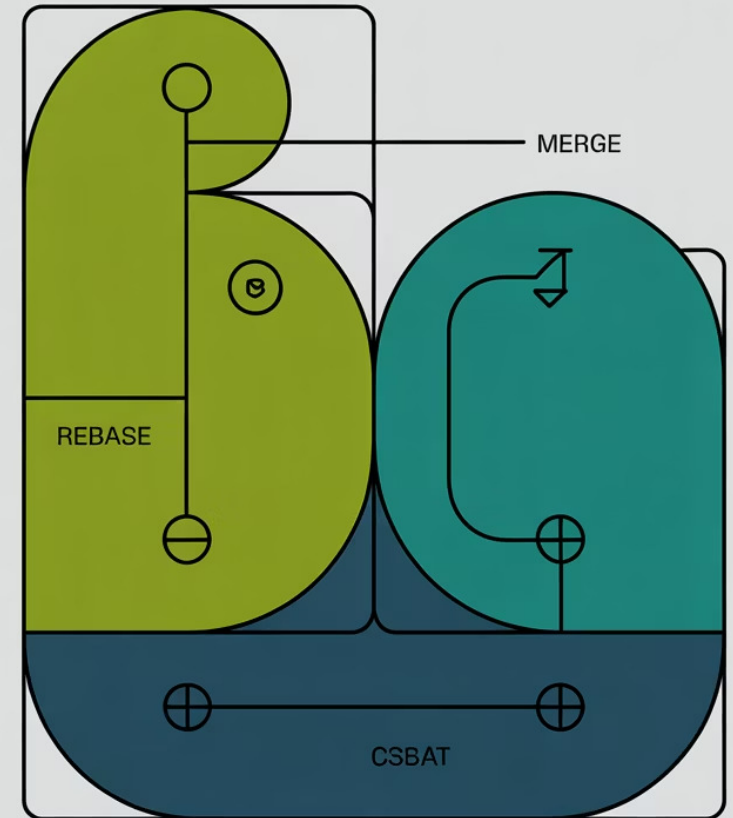
```
git rebase <rama-base>
```

Ventajas del Rebase

- Historial lineal y más limpio
- Facilita la lectura del log
- Elimina commits de fusión innecesarios

Consideraciones

- No usar en ramas públicas compartidas
- Reescribe el historial de commits
- Puede complicar la colaboración



Eliminar Ramas

Una vez que una rama ha cumplido su propósito y sus cambios han sido fusionados, es buena práctica eliminarla para mantener el repositorio organizado.

Eliminación Segura

```
git branch -d <nombre-rama>
```

Elimina la rama solo si ha sido fusionada. Git previene la pérdida accidental de trabajo no fusionado.

Eliminación Forzada

```
git branch -D <nombre-rama>
```

Elimina la rama incluso si no ha sido fusionada. Úsalo con precaución ya que puedes perder cambios.

 **Buena práctica:** Elimina ramas regularmente después de fusionarlas para mantener tu repositorio limpio y organizado.

Repositorios Remotos

Los repositorios remotos son versiones del proyecto alojadas en servidores de internet o en la red. Permiten la colaboración entre múltiples desarrolladores y actúan como respaldo del código.



¿Qué es GitHub?

GitHub es la plataforma de alojamiento de repositorios Git más popular del mundo. Ofrece herramientas de colaboración que van más allá del simple control de versiones.

Repositorios Públicos y Privados

Aloja proyectos de código abierto o privados con control de acceso

Pull Requests

Sistema de revisión de código y propuestas de cambios

Issues y Wikis

Seguimiento de tareas, bugs y documentación del proyecto

Añadir Repositorios Remotos

git remote add

El comando `git remote add` conecta tu repositorio local con un repositorio remoto, estableciendo un enlace para sincronizar cambios.

```
git remote add <nombre> <url>
```

Por convención, el repositorio remoto principal se llama `origin`. Este nombre se usa automáticamente cuando clonas un repositorio.

Ejemplo:

```
git remote add origin https://github.com/usuario/proyecto.git
```

Puedes tener múltiples repositorios remotos con diferentes nombres.



Listar Repositorios Remotos

Git proporciona comandos para visualizar qué repositorios remotos están configurados y sus URLs asociadas.

Listar Nombres

```
git remote
```

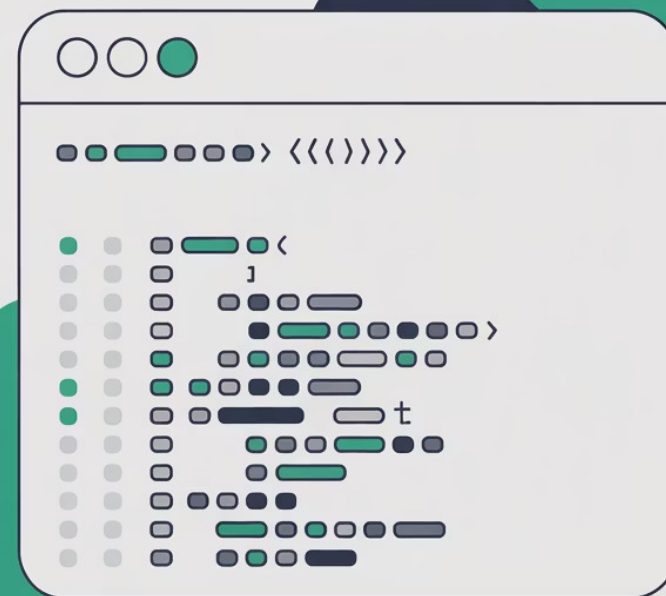
Muestra los nombres de todos los repositorios remotos configurados

Listar con URLs

```
git remote -v
```

Muestra nombres y URLs completas de fetch y push para cada remoto

La opción `-v` (verbose) es especialmente útil para verificar que los repositorios remotos están correctamente configurados y apuntan a las URLs correctas.



Descargar Cambios del Remoto

Git ofrece dos comandos principales para obtener cambios desde un repositorio remoto. Aunque ambos descargan información, funcionan de manera diferente.

git pull

```
git pull
```

Descarga cambios del repositorio remoto y los integra automáticamente en tu rama actual. Es equivalente a ejecutar `git fetch` seguido de `git merge`.

git fetch

```
git fetch
```

Descarga cambios del repositorio remoto pero NO los integra automáticamente. Te permite revisar los cambios antes de fusionarlos manualmente.

📌 **Recomendación:** Usa `git fetch` cuando quieras revisar cambios antes de integrarlos. Usa `git pull` cuando confíes en los cambios remotos.

Subir Cambios al Remoto

git push

El comando `git push` envía tus commits locales al repositorio remoto, compartiendo tu trabajo con otros colaboradores y respaldando tus cambios en la nube.

```
git push <remoto> <rama>
```



Repositorio Local

Commits en tu máquina



git push

Envío de cambios



Repositorio Remoto

Commits sincronizados en servidor

Ejemplo común:

```
git push origin main
```

Colaboración en GitHub: Colaborador

Cuando eres añadido como colaborador directo en un repositorio de GitHub, tienes permisos para hacer push directamente. Este es el flujo de trabajo más sencillo para equipos pequeños.



Ser Añadido como Colaborador

El propietario del repositorio te añade en la configuración de colaboradores



Clonar el Repositorio

Descargas una copia completa del proyecto a tu máquina local



Realizar Cambios

Modificas archivos, creas commits con tus cambios



Sincronizar

Ejecutas `git pull` para obtener cambios recientes, luego `git push` para subir tus commits

Colaboración por Fork

El flujo de trabajo con fork es ideal para contribuir a proyectos de código abierto o cuando no tienes permisos de escritura directos en el repositorio original.





Referencias y Recursos

Continúa tu aprendizaje de Git con estos recursos oficiales y tutoriales recomendados. Desde documentación oficial hasta guías prácticas, estos enlaces te ayudarán a profundizar en el dominio de Git.

Documentación Oficial

- [Git: Sitio oficial \(git-scm.com\)](https://git-scm.com)
- [GitHub: Sitio oficial \(github.com\)](https://github.com)
- [Pro Git - Libro oficial gratuito](#)

Tutoriales y Guías

- [Ry's Git Tutorial](#)
- [Gitcheats - Trucos y consejos](#)
- [Chuleta de comandos de Git](#)

Flujos de Trabajo

- [Flujos de trabajo con Git](#)
- [Mejores prácticas de branching](#)
- [Estrategias de colaboración](#)