



2025

Nivel intermedio

# DOMINA GIT

CONTROL DE VERSIONES DE  
PRINCIPIO A FIN



Ariadna García



ariadnagrc

# ÍNDICE

- 1** Conceptos clave.
- 2** Trabajar con ramas.
  - ◆ Qué son y cómo usarlas.
  - ◆ Comandos esenciales para controlar ramas.
- 3** Fusionar ramas con merge.
  - ◆ Inicializar repositorio.
  - ◆ Configurar el remoto.
  - ◆ Ver estado.
  - ◆ Preparación de archivos.
  - ◆ Hacer un commit.
  - ◆ Guardar y subir cambios.
- 4** Rebase vs Merge.
- 5** Resolución de conflictos.
- 6** Uso de stash.
- 7** Historial de commits.
  - ◆ Git reflog.
- 8** Recuperar y deshacer cambios.
  - ◆ Git reset.
  - ◆ Git revert.
  - ◆ Git restore.

## ¿Qué aprenderás en esta guía?

Ahora que ya conoces los comandos básicos de Git y sabes cómo usarlos para gestionar tus archivos en proyectos individuales, es momento de dar un paso más en tu camino como desarrollador/a.

En esta guía intermedia vamos a descubrir nuevas funcionalidades que hacen de Git una herramienta aún más poderosa, especialmente cuando trabajas con **proyectos más grandes** o en **colaboración** con otras personas. Incluso si todavía trabajas solo/a, muchos de estos conceptos te ayudarán a mantener tu proyecto más limpio, organizado y fácil de gestionar.

Este volumen está enfocado en conceptos que, al principio, pueden sonar complicados, pero que una vez entiendes su lógica, se convierten en herramientas imprescindibles en tu día a día con Git. Aprenderás a:

- ◆ **Crear y moverte entre ramas** para trabajar en nuevas ideas sin afectar el código principal.
- ◆ **Combinar diferentes ramas** y manejar los cambios de forma segura.
- ◆ **Resolver conflictos** cuando dos versiones del código no coinciden.
- ◆ **Deshacer errores** o **volver a versiones anteriores** de forma segura.
- ◆ Aplicar buenas prácticas que te prepararán para **trabajar en entornos reales** de desarrollo.

Al finalizar esta guía, habrás adquirido las habilidades necesarias para gestionar tu código de manera más eficiente y organizada, lo que te permitirá trabajar con proyectos más complejos y en colaboración con otros. Este flujo de trabajo te permitirá no solo versionar y organizar tu código de forma más efectiva, sino también enfrentarte a desafíos comunes en entornos reales de desarrollo. Una vez que domines estos conceptos intermedios, estarás listo/a para manejar ramas, resolver conflictos y colaborar con equipos de desarrollo de manera profesional.

Como en el anterior PDF, he creado una breve guía de vocabulario para familiarizarte más rápido con los conceptos de esta edición. Esta guía te servirá para entender los términos clave que usaremos, lo que facilitará el aprendizaje de nuevas funcionalidades de Git. Tener claro el vocabulario desde el principio hará que los siguientes pasos sean más fáciles de aplicar y te ayudará a avanzar con mayor confianza.

## Branch (rama)

Una rama es una línea paralela de desarrollo. Te permite trabajar en nuevas funcionalidades o ideas sin modificar directamente el código principal (normalmente en main o master). Ideal para organizar tu trabajo por partes.

## Merge (fusión)

Unir los cambios de una rama a otra. Por ejemplo, juntar una rama de pruebas con la principal cuando todo funciona bien.

## Merge conflict (conflicto)

Ocurre cuando Git no puede decidir automáticamente cómo combinar los cambios de dos ramas. Te pedirá que elijas qué línea conservar.

## Commit

Cada vez que haces un commit, estás creando un punto de guardado en tu proyecto. Git guarda los cambios y una descripción de lo que hiciste.

## Conflicto

Un conflicto ocurre cuando Git no puede decidir cuál cambio conservar porque dos ramas modificaron la misma línea de un archivo.

## Staging area

Es una zona intermedia donde se preparan los cambios antes de confirmar un commit. Es como una bandeja de entrada donde seleccionas exactamente qué cambios quieres guardar en el historial del proyecto.

## Stash

Stash es un conjunto temporal de cambios no confirmados que has decidido apartar momentáneamente. Al hacer un stash, se guarda el estado actual del directorio de trabajo y del área de staging sin crear un commit, permitiéndote trabajar en otra cosa sin perder el progreso.

## HEAD

Head es un puntero especial en Git que señala al commit actual en el que estás trabajando. Normalmente apunta al último commit de la rama activa.

### ¿Qué son las ramas?

Una rama (branch) en Git es una forma de separar tu trabajo sin afectar la versión principal del proyecto. Puedes pensarla como una copia temporal del proyecto donde puedes experimentar, añadir nuevas funcionalidades o corregir errores.



#### RECUERDA

Git no guarda varios proyectos separados, sino que usa punteros para indicar en qué parte del historial estás trabajando.

La rama por defecto se llama main (o master en proyectos antiguos).

### ¿Para qué utilizar ramas?

Las ramas son una herramienta fundamental en Git que permiten trabajar de forma organizada y segura dentro de un proyecto. Sirven para:

- Desarrollar nuevas funcionalidades *sin interferir con el código principal, lo que evita introducir errores en versiones estables.*
- **Corregir errores** (bugs) **de manera aislada**, asegurando que los arreglos no se mezclen con otros cambios aún en desarrollo.
- **Probar ideas** o **implementar mejoras experimentales** *sin comprometer el proyecto principal.*
- Facilitar el **trabajo en equipo**, ya que cada integrante puede trabajar en su propia rama, lo que reduce conflictos y mejora la colaboración.

### Buenas prácticas

- Usa nombres descriptivos: correccion-login, feature-busqueda, hotfix-header.
- No trabajes directamente en main.
- Antes de fusionar, asegúrate de tener los últimos cambios de main.

## Comandos esenciales para controlar ramas

Estos son los comandos más básicos y esenciales para trabajar con ramas en Git. Aunque existen muchas otras operaciones más avanzadas (renombrar ramas, eliminarlas o configurarlas para seguimiento remoto), con estos comandos puedes llevar un flujo de trabajo completo y eficiente en la mayoría de proyectos.

### Ver ramas

Este comando te permite revisar rápidamente todas las ramas locales en tu proyecto. Es útil para conocer en qué ramas estás trabajando y cuál es la rama activa en ese momento. La rama actual aparecerá con un asterisco \* al lado.

```
>_
git branch
```

### Crear ramas

Este comando es útil cuando deseas crear una rama para un propósito específico, pero prefieres quedarte en tu rama actual para seguir trabajando en lo que estás haciendo. Puedes cambiarte a la nueva rama más tarde cuando estés listo para comenzar a trabajar en ella.

*NO cambia automáticamente a esa rama, solo la crea.*

```
>_
git branch nombre-de-la-rama
```

### Crear una rama y moverte a ella

Una forma más moderna y clara de crear una nueva rama y cambiarte a ella al instante es con git switch -c. Este comando es más legible y recomendado en versiones recientes de Git.

```
>_
git switch -c nombre-de-la-rama
```

### *Móverte entre ramas*

Este comando te permite regresar a una rama existente, como main, para continuar trabajando en ella. Es útil cuando necesitas volver a la rama principal del proyecto o a cualquier otra rama en la que hayas trabajado previamente.

```
>_
```

```
git switch nombre-de-la-rama
```

Con checkout también puedes cambiar de rama, pero es menos recomendado en versiones modernas de Git porque git checkout tiene múltiples usos (como restaurar archivos o hacer checkout de commits específicos), lo que puede llevar a confusión.

```
>_
```

```
git checkout nombre-de-la-rama
```

Aunque sigue funcionando, se recomienda usar git switch para cambiar de rama, ya que es más claro y está pensado específicamente para esa tarea.

### *Eliminar ramas (locales)*

Este comando te permite eliminar una rama que ya no necesitas en tu repositorio local, por ejemplo, después de fusionarla con la rama principal. Es útil para mantener tu lista de ramas limpia y organizada.

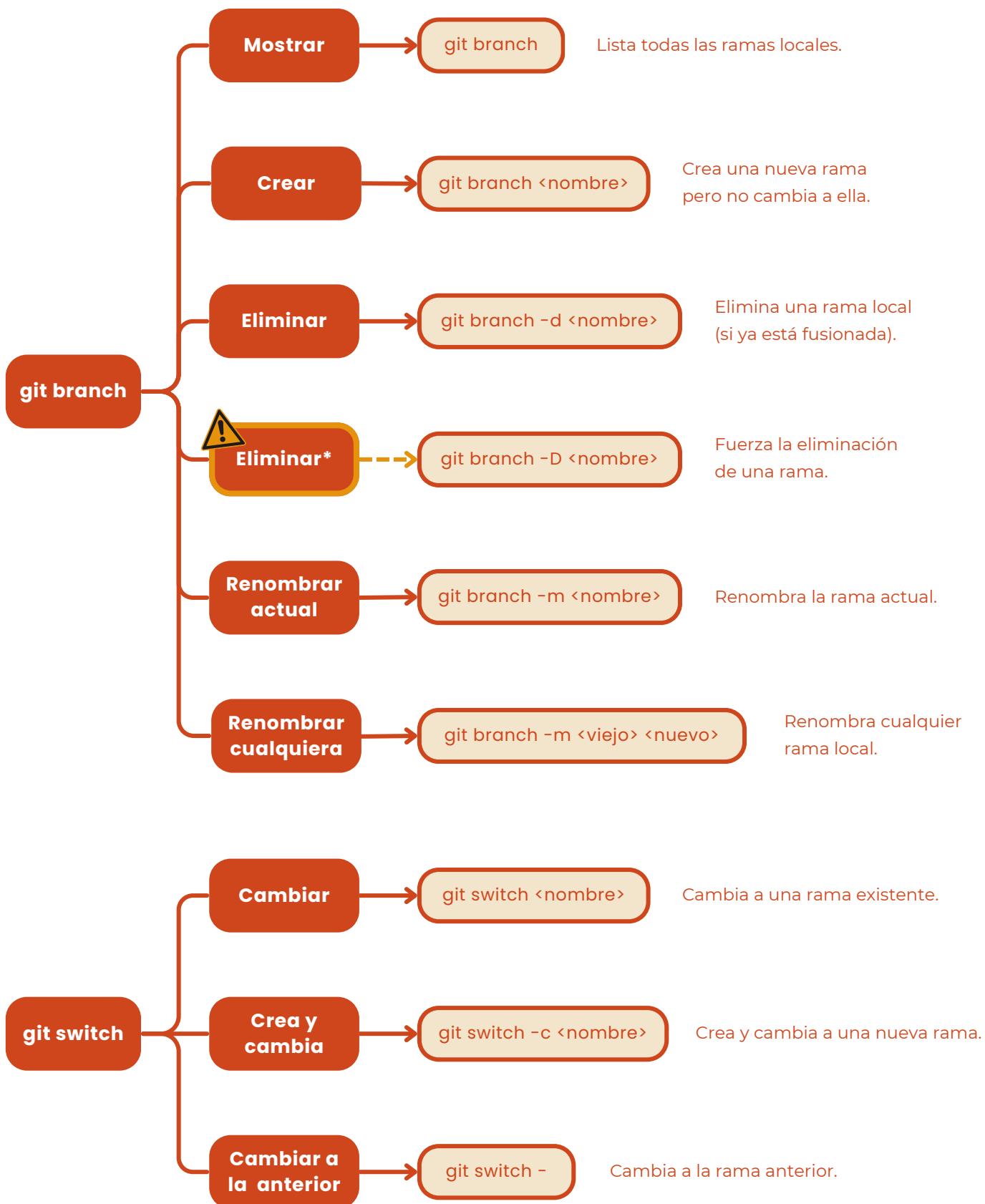
```
>_
```

```
git branch -d nombre-de-la-rama
```



#### **CUIDADO!**

Usa -d solo si la rama ya ha sido fusionada. Si necesitas forzar la eliminación (aunque no esté fusionada), puedes usar -D: elimina la rama sin preguntar y podrías perder cambios no fusionados.



3

## FUSIONAR RAMAS CON MERGE

Cuando terminas de trabajar en una rama (por ejemplo, feature-login) y quieres incorporar esos cambios a la rama principal (main), debes hacer una fusión, también conocida como merge, que integra los cambios de una rama en otra sin eliminar el historial.

Se ejecuta desde la rama en la que quieras recibir los cambios.

```
>_  
git merge nombre-de-la-rama
```

### Ejemplo

```
>_  
  
# Estás en main  
git switch main  
  
# Fusionas la rama "feature-login" a main  
git merge feature-login
```

Si no hay conflictos, Git fusionará automáticamente los cambios.

Si lo hay, primero Git te avisa:

```
>_  
CONFLICT (content): Merge conflict in archivo.txt
```

Abres el archivo y verás algo como:

```
>_  
  
<<<<< HEAD  
Esto viene de la rama actual (main)  
=====br/>Esto viene de la rama que estás fusionando (feature-login)  
>>>>> feature-login
```

Tú decides con qué te quedas (o combinas ambas partes) y guardas. Luego marcas el conflicto como resuelto:

```
>_
```

```
git add archivo.txt  
git commit
```



### RECUERDA

Git no hace automáticamente el commit del merge si hubo conflictos: tú debes resolverlos primero.

## ¿Cómo evitar conflictos?

Evitar conflictos es parte fundamental de un flujo de trabajo limpio y eficiente. Aquí tienes algunas buenas prácticas para reducir al mínimo los problemas al fusionar ramas:

- *Haz git pull antes de empezar a trabajar.*

Así te aseguras de tener la última versión del proyecto y evitas sobrescribir cambios de otros.

- *Trabaja en ramas separadas por tema o funcionalidad.*

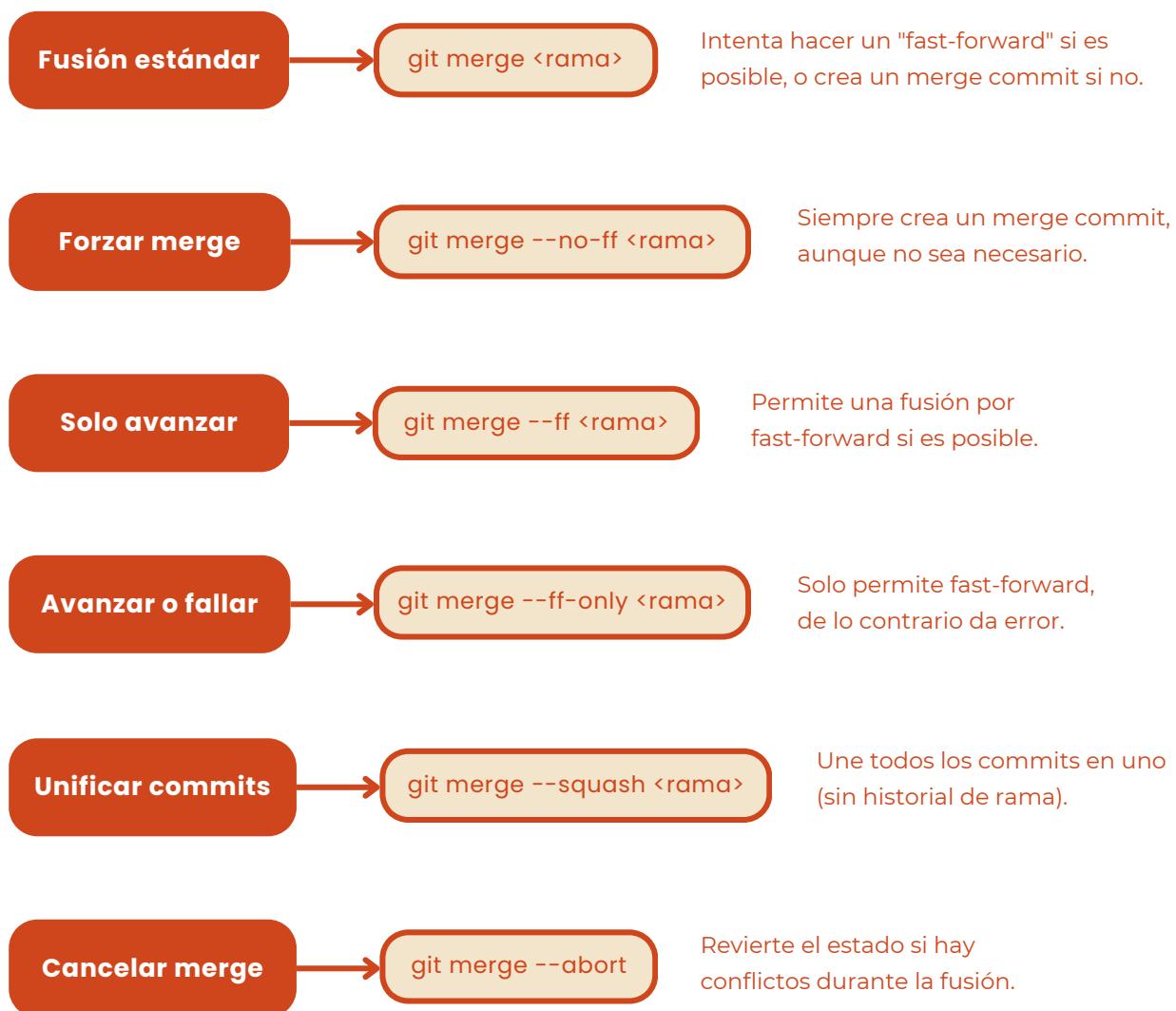
Esto aísla los cambios y facilita la integración sin interferencias con otras tareas

- *Comunícate con tu equipo.*

Coordinar quién trabaja en qué parte del código evita ediciones simultáneas en los mismos archivos.

- *Haz merges frecuentemente, no dejes que las ramas diverjan mucho.*

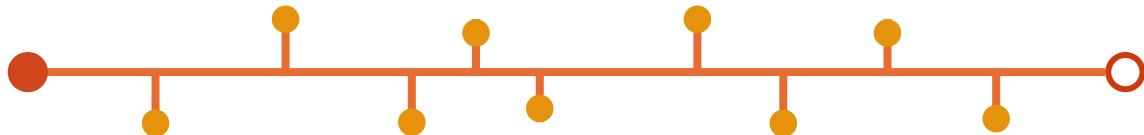
No dejes que las ramas se alejen demasiado de main u otras ramas principales. Cuanto más tiempo pasa, mayor es el riesgo de conflicto.



## REBASE VS MERGE: ¿CUÁL USAR?

Cuando quieras integrar cambios sin generar merge commits, git rebase es una herramienta ideal.

En lugar de crear un nuevo commit de fusión, rebase toma los commits de tu rama y los "reubica" al final de la rama base, como si hubieran sido creados después de los últimos cambios. Esto reescribe el historial y da como resultado una línea de tiempo más ordenada y continua, sin merge commits.



- *Úsalo cuando quieras un historial claro y limpio.*

Cuando estás trabajando en tu rama y quieres actualizarla con los últimos cambios de main antes de fusionarla. También cuando quieres un historial limpio y lineal (por ejemplo, antes de hacer un merge a producción).

- *Evita usarlo en ramas compartidas.*

Como rebase reescribe el historial, puede generar conflictos o confusión si otras personas están trabajando en la misma rama. En entornos colaborativos, es más seguro usar merge.

### ¿Cómo evitar conflictos?

Igual que en merge, si Git no puede aplicar un commit porque hay conflicto, te avisa de dichos archivos, los editas resolviendo los conflictos, y por último haces:

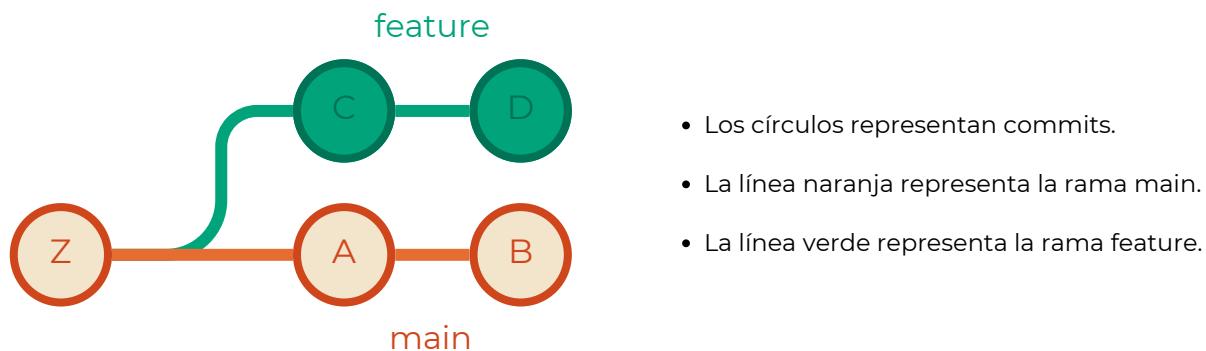
```
>_
git add archivo.txt
git rebase --continue
```

Si quieres cancelar el rebase:

```
>_
git rebase --abort
```

## Ejemplo

Supongamos que estamos trabajando en un proyecto y hemos comenzado con la rama principal, llamada main. En algún momento, decidimos crear una nueva rama llamada feature para trabajar en una funcionalidad específica sin afectar el trabajo estable que ya está en main.



Como se ve, la rama feature se creó a partir de un commit de main. A partir de ahí, se hicieron dos commits nuevos en feature, sin que se afectara la rama main. Esto es útil cuando queremos desarrollar algo nuevo sin arriesgar el código que ya funciona bien.

Si ejecutas:

```
>_
git switch feature
git rebase main
```

Estás diciendo:

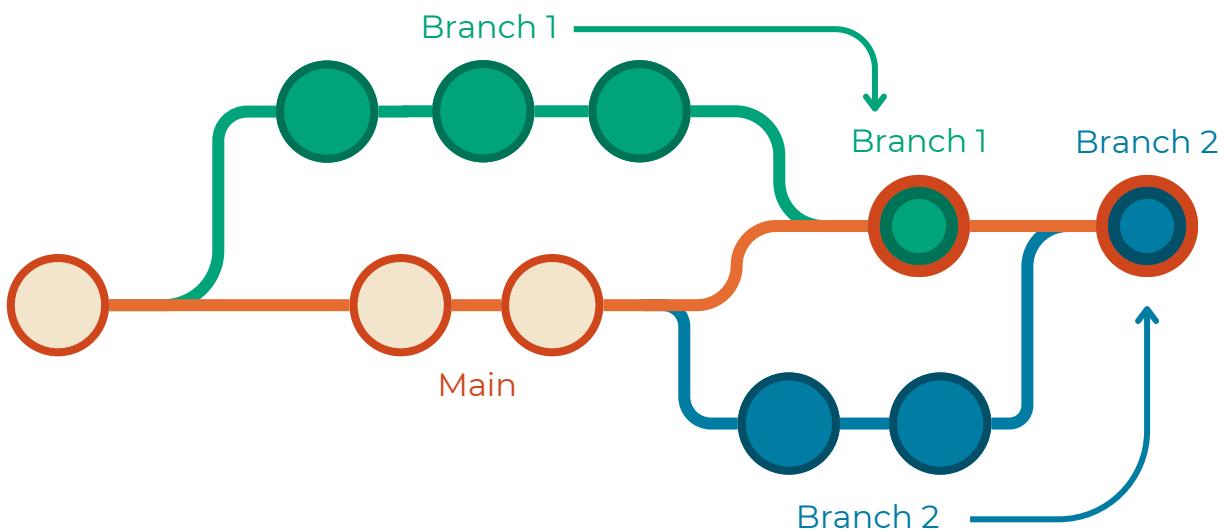
"Quiero mover mi trabajo de la rama feature (commits C y D) como si lo hubiera empezado justo después de lo que hay ahora en main (después de B)".

Git lo que hace es:

1. Quita temporalmente tus commits C y D.
2. Coloca tu rama feature en la punta de main, es decir, después del commit B.
3. Reaplica los commits C y D, pero con nuevos identificadores, ya que técnicamente se están creando de nuevo.

Así, parece que primero se hicieron todos los cambios de la rama main, y luego, de forma seguida, se hicieron los cambios de la rama feature, como si todo el desarrollo hubiera sido una única línea de trabajo, ordenada cronológicamente.

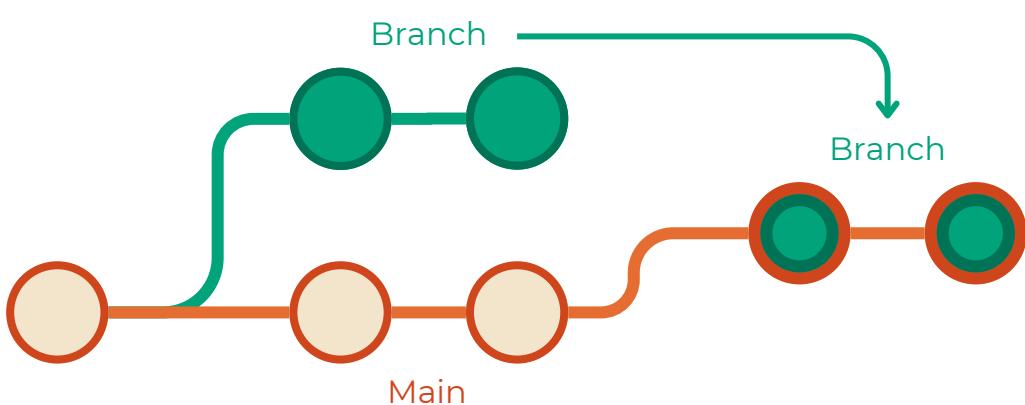
## GIT MERGE



Crea un commit extra para unir las ramas. El historial se bifurca y luego se une.

- Pros: conserva el historial real del trabajo.
- Contras: puede generar un historial más "sucio" o complejo.

## GIT REBASE



Reubica los commits de la rama sobre el final de la otra. El historial queda lineal.

- Pros: historial más limpio y fácil de seguir.
- Contras: reescribe la historia; no apto para ramas compartidas.

## 5

## RESOLUCIÓN DE CONFLICTOS

Cuando estás trabajando con Git y usas comandos como merge o rebase para combinar cambios de distintas ramas, puede ocurrir una situación llamada conflicto. Esto sucede cuando dos ramas han hecho cambios diferentes en la misma parte de un archivo, y Git no puede decidir automáticamente qué versión debe conservar.

Imagina que tú y otra persona están editando el mismo documento. Tú cambias una frase en la línea 10, y esa otra persona también cambia esa misma línea, pero de otra forma. Cuando Git intenta unir esas versiones, no puede adivinar cuál es la correcta. Entonces te dice: "Hay un conflicto, necesito que tú decidas cómo resolverlo".

Este tipo de situaciones pueden parecer confusas al principio, pero con un poco de práctica sabrás cómo resolverlas fácilmente.

### ¿Cómo se ve un conflicto en Git?

Supongamos que dos personas editan el mismo archivo en diferentes ramas.

En la rama main:

```
>_
function saludar() {
  console.log("Hola desde main");
}
```

En la rama feature:

```
>_
function saludar() {
  console.log("Hola desde feature");
}
```

Ahora intentas fusionar feature en main:

```
>_
git checkout main
git merge feature
```

Git detecta que ambos modificaron la misma línea y no puede decidir cuál usar, así que deja el archivo en este estado:

```
>_
function saludar() {
<<<<< HEAD
    console.log("Hola desde main");
=====
    console.log("Hola desde feature");
>>>>> feature
}
```

<<<<< HEAD: representa lo que hay en tu rama actual (main).

=====: separa los dos cambios.

>>>>> feature: muestra lo que viene desde la rama que intentas fusionar.

Ahora es tu turno de intervenir: tú decides qué contenido conservar, qué eliminar o cómo combinar ambas versiones. Puedes hacerlo editando el archivo manualmente o con la ayuda de herramientas visuales, como los editores de Git integrados en VS Code o programas como GitKraken, que facilitan este proceso.

Tienes varias opciones:

- Elegir el código de tu rama actual (HEAD).
- Elegir el código de la rama que estás fusionando (feature).
- Combinar manualmente ambas versiones editando el archivo para unir ideas de ambas versiones.

```
>_
function saludar() {
    console.log("Hola desde main");
    console.log("Hola desde feature");
}
```

- Modificar completamente el bloque, ignoras ambas versiones y escribes una nueva que consideras mejor.

```
>_
function saludar() {
  console.log("Hola a todos desde ambas ramas");
}
```

Después de editar el archivo, siempre recuerda:

```
>_
git add archivo_con_conflicto
git commit
```

En rebase, debes hacer:

```
>_
git rebase --continue
```

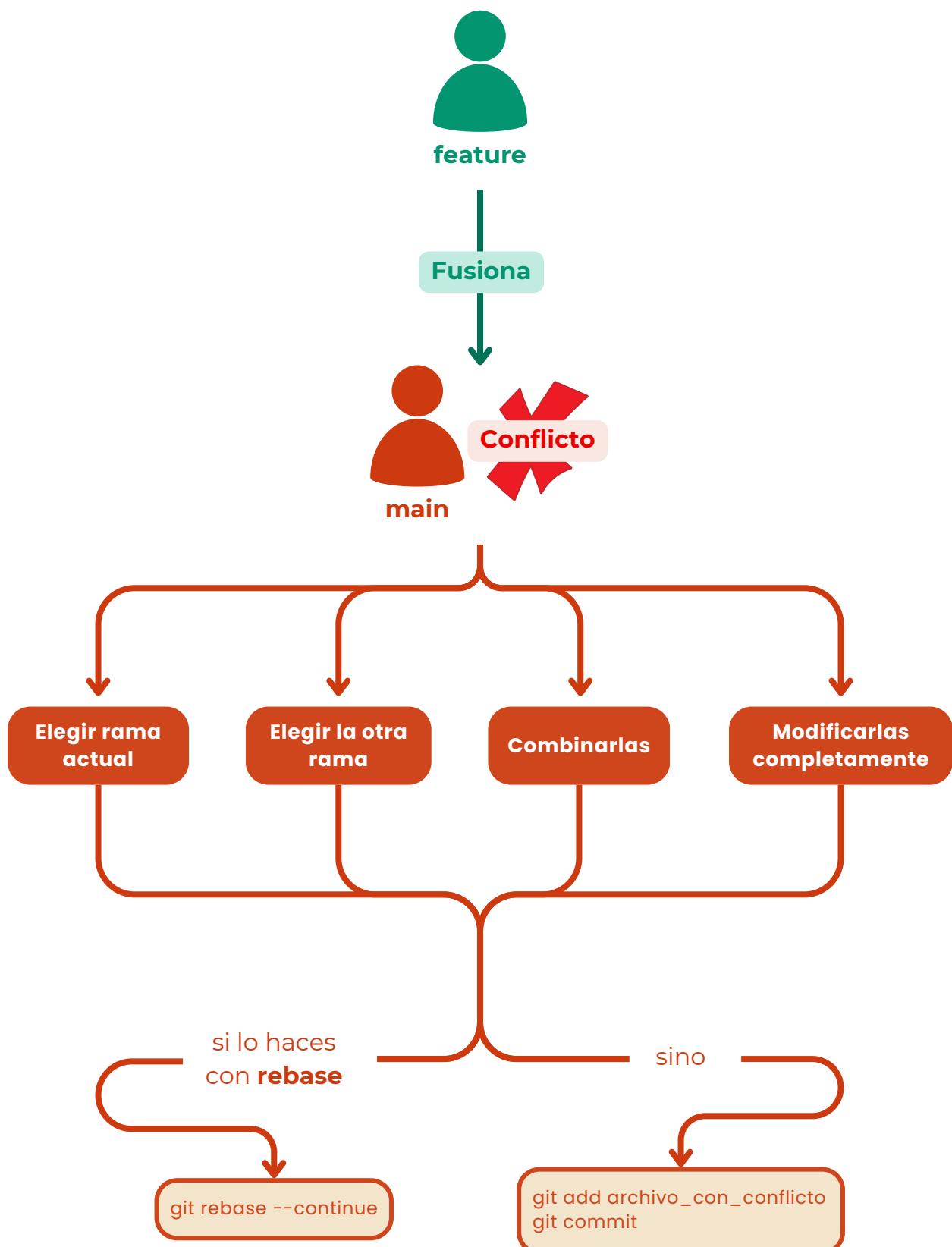


### RECUERDA

Git tiene integración con editores y GUIs que te ayudan a resolver conflictos visualmente:

- git mergetool abre herramientas como Meld, VSCode, etc.
- En VS Code, al abrir un archivo con conflicto, verás botones como:
  - "Aceptar cambio actual"
  - "Aceptar cambio entrante"
  - "Aceptar ambos"
  - "Comparar cambios"

Útil si no te gusta editar a mano oquieres ver mejor las diferencias.



## USO DE STASH: GUARDA CAMBIOS SIN CONFIRMAR

Imagina que estás trabajando en una funcionalidad, haciendo varios cambios en tu código, pero de repente necesitas atender una urgencia: cambiar de rama, probar otra cosa o ayudar a un compañero. El problema es que aún no estás listo para hacer commit, y no quieres perder lo que llevas hecho. Aquí es donde entra en juego git stash.

Este comando te permite guardar temporalmente tus cambios no confirmados (tanto en el área de trabajo como en el área de preparación), sin necesidad de hacer un commit. Es como si guardaras tu trabajo en un cajón, para poder retomarlo más tarde sin estorbar en el resto del proyecto.

Con git stash puedes:

- Cambiar de rama sin perder tu progreso actual.
- Volver a un estado limpio para trabajar en otra cosa.
- Recuperar tus cambios cuando estés listo para seguir.

Piensa en stash como una “pausa técnica”: congelas tu estado actual, haces lo que tengas que hacer, y luego continúas donde lo dejaste.

### Identificación de stash

Cuando trabajas con varios stashes en tu repositorio, no basta con aplicar siempre el más reciente. Afortunadamente, Git te permite hacer referencia a stashes anteriores utilizando un índice. Esto es útil si has guardado varios cambios y necesitas acceder a un stash en particular.

Para referirnos a un stash específico, usamos la sintaxis `stash@{n}`, donde n es el número del stash en la lista (empezando desde 0 para el más reciente).

```
>_
```

```
stash@{0}: WIP on main: fba6b9a Cambios sin terminar
stash@{1}: WIP on feature: 7b8c9d0 Añadir funcionalidad de búsqueda
```

En este caso:

- `stash@{0}` es el stash más reciente.
- `stash@{1}` es el segundo stash guardado, y así sucesivamente.

Git stash es el comando base. Guarda los cambios en archivos ya versionados (tracked) y elimina los cambios del directorio de trabajo. Es pocas palabras, es como hacer un “guardar borrador”.

```
>_
```

```
git stash
```

Aunque la versión moderna de Git ya no recomienda tanto esta forma, es posible añadir un mensaje descriptivo al guardar los cambios utilizando:

```
>_
```

```
git stash save "mensaje opcional"
```

A medida que vas utilizando git stash, es posible que termines acumulando varios cambios guardados. Para revisar todo lo que has stashed hasta el momento, puedes usar el siguiente comando:

```
>_
```

```
git stash list
```

*Este comando muestra una lista con todos los stashes que tienes guardados, cada uno identificado con un nombre como stash@{0}.*

Cuando estés listo para recuperar tu trabajo, puedes aplicar los cambios guardados con:

```
>_
```

```
git stash apply
```

*Este comando toma el stash más reciente y lo aplica sobre tu directorio de trabajo. Lo interesante es que no elimina el stash de la lista, por lo que puedes volver a usarlo más adelante si lo necesitas de nuevo.*

Si deseas aplicar un stash específico de la lista, también puedes indicarlo directamente, por ejemplo:

```
>_
```

```
git stash apply stash@{2}
```

Si ya no necesitas conservar el stash después de aplicarlo, puedes usar:

```
>_
```

```
git stash pop
```

*Este comando aplica los cambios guardados y además los elimina automáticamente de la lista de stashes. Es una forma más ordenada de trabajar, ya que evita acumular entradas innecesarias una vez que has recuperado tu trabajo. Ideal si sabes que no vas a necesitar reutilizar ese stash más adelante.*

A veces, puedes darte cuenta de que un stash antiguo ya no te sirve o simplemente quieres limpiar tu lista. Para borrar un stash en concreto, puedes usar:

```
>_
```

```
git stash drop stash@{0}
```

*Solo necesitas indicar el identificador correspondiente (que puedes ver con git stash list). Esto elimina únicamente ese stash, sin afectar a los demás.*

*Si decides hacer borrón y cuenta nueva, y quieres eliminar todos los stashes guardados, puedes ejecutar:*

```
>_
```

```
git stash clear
```

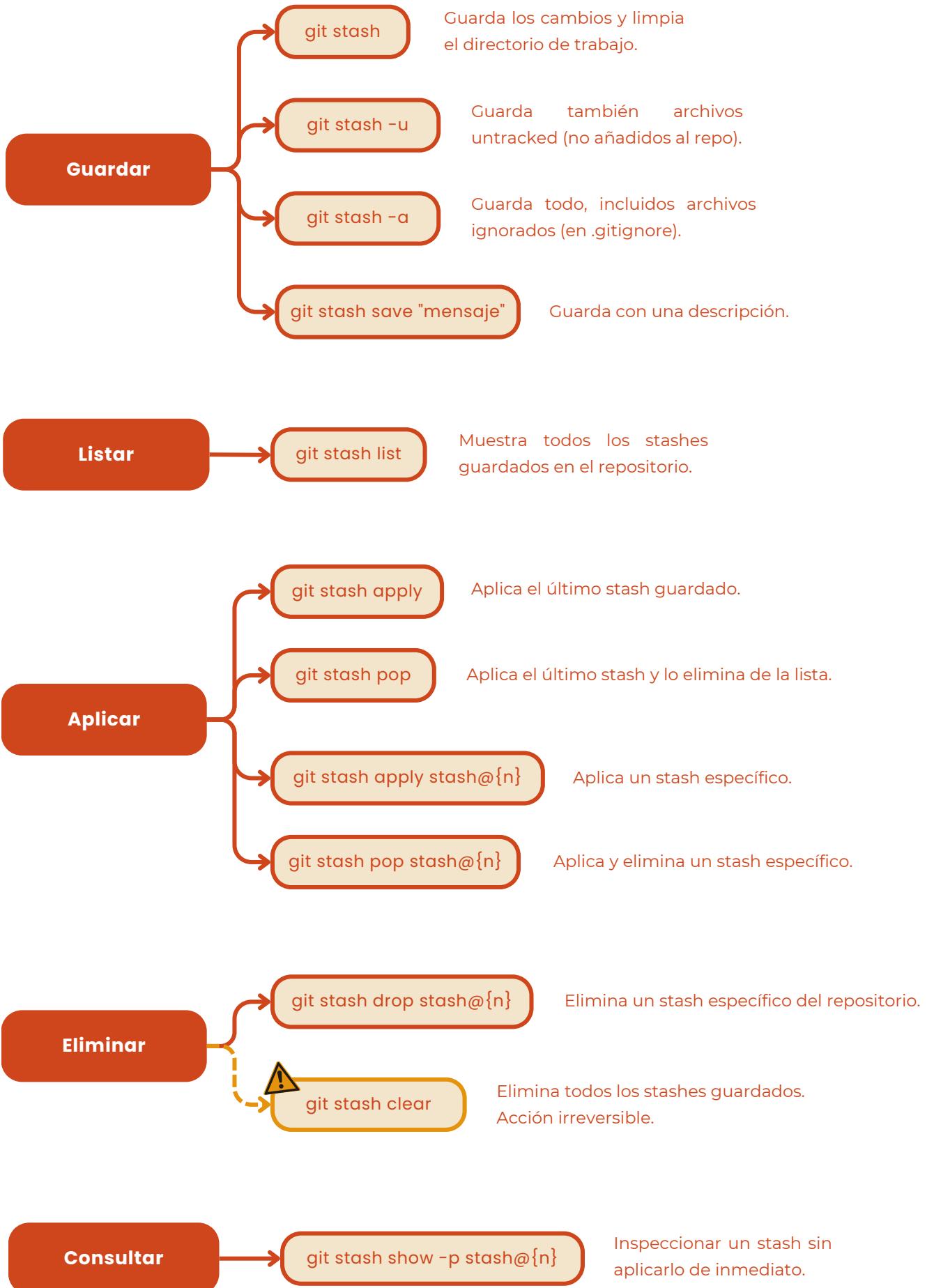
*Ten cuidado con este comando: borra todos los stashes sin pedir confirmación y no hay forma de recuperarlos una vez eliminados. Asegúrate de que no necesitas ninguno antes de usarlo.*

Si no estás seguro de qué stash aplicar, primero mira el contenido de los stashes con:

```
>_
```

```
git stash show -p stash@{n}
```

*Esto te permite inspeccionar un stash sin aplicarlo de inmediato.*



Git no solo mantiene un historial de los commits realizados, sino que también ofrece herramientas poderosas para rastrear y recuperar información crucial, incluso sobre los cambios que parecían perdidos.

En la edición anterior de esta guía ya exploramos el uso de `git log` para revisar el historial de commits. Ahora vamos un paso más allá con una herramienta menos conocida pero increíblemente útil: `git reflog`. Este guarda un historial interno y local de todas las acciones que han afectado el puntero de `HEAD` y otras referencias, incluso si esas acciones no están reflejadas en el historial visible con `git log`.

Es tu última línea de defensa para recuperar commits perdidos, especialmente después de un:

- `git reset --hard`
- `git rebase`
- `git commit --amend`
- o un borrado accidental de ramas

```
>_
```

```
git reflog
```

Este comando te mostrará una lista de acciones con sus correspondientes identificadores, como `HEAD@{0}`, `HEAD@{1}`, `HEAD@{2}`, etc., junto con mensajes que indican qué ocurrió en cada punto (`checkout`, `commit`, `reset`...).

Verás algo como:

```
>_
```

```
e5c12c3 HEAD@{0}: reset: moving to HEAD~1
a8f29e4 HEAD@{1}: commit: corrige error en login
9b7fa7e HEAD@{2}: checkout: moving from main to feature
```

Una vez identificado el punto al que quieras volver, tienes dos formas principales de recuperar tu proyecto.

### **Volver temporalmente a un estado anterior**

Si solo quieres inspeccionar o revisar ese estado anterior sin afectar tu rama actual, puedes hacer:

```
>_
```

```
git checkout HEAD@{1}
```

Esto te llevará a ese punto específico en modo detached HEAD, lo que significa que no estás en ninguna rama. Desde aquí puedes ver archivos, copiar código o incluso crear una nueva rama desde ahí con:

```
>_
```

```
git switch -c nombre-nueva-rama
```

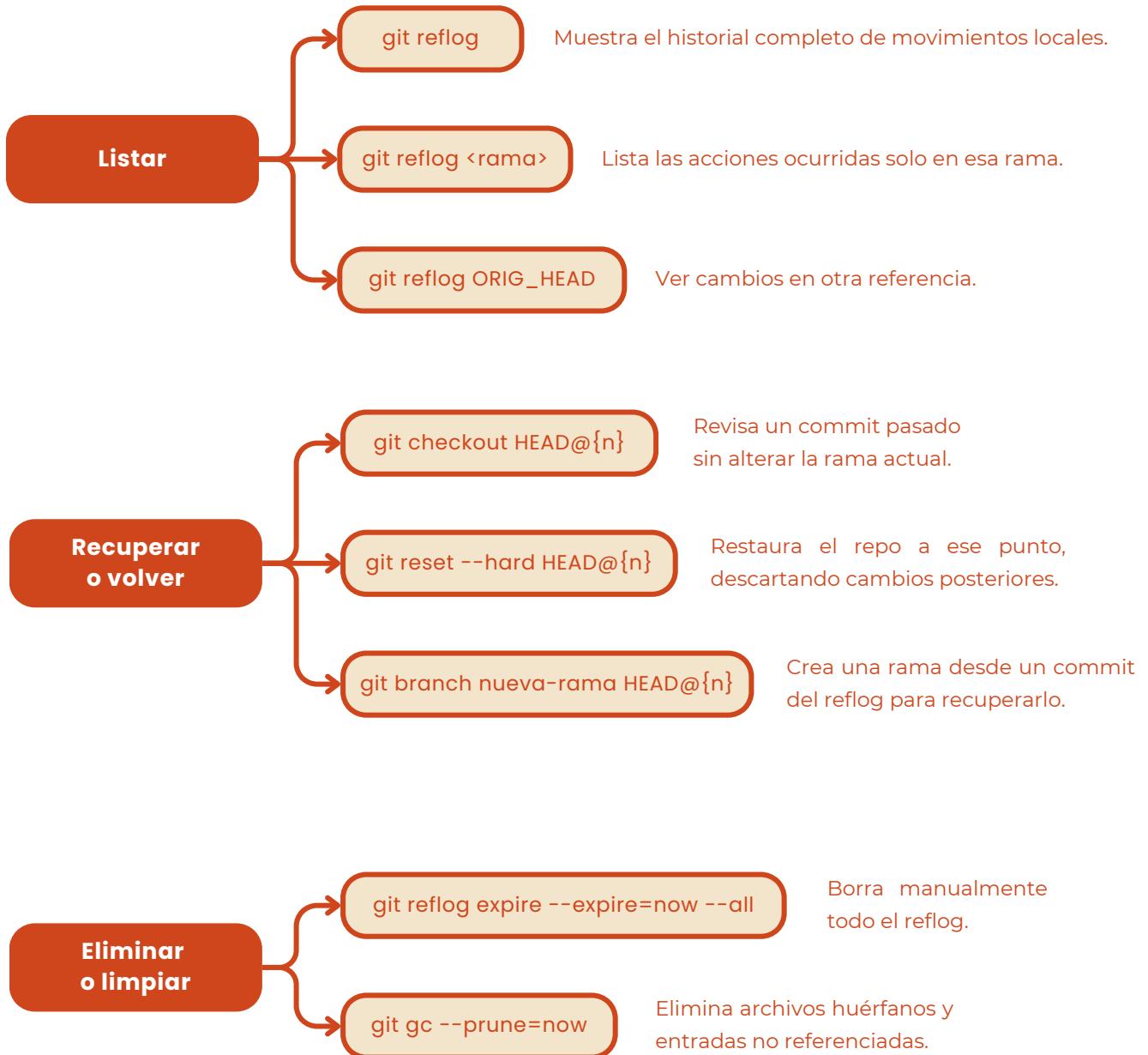
### **Restaurar completamente ese estado**

Si lo que necesitas es devolver tu rama actual a ese punto específico (como si deshicieras todo lo que hiciste después), puedes usar:

```
>_
```

```
git reset --hard HEAD@{1}
```

Esto moverá tu rama y tu directorio de trabajo exactamente al estado en que estaba en ese punto del reflog. Ten cuidado: como es un --hard, perderás cualquier cambio sin guardar que tengas actualmente.



A lo largo del trabajo con Git, es normal cometer errores: un commit adelantado, cambios no deseados o incluso borrar algo importante. La buena noticia es que Git ofrece herramientas muy poderosas para deshacer, recuperar y corregir.

## Git reset

Reset es una herramienta poderosa que permite deshacer commits y controlar el estado del historial, el área de staging (index) y el directorio de trabajo. Dependiendo del modo que elijas (--soft, --mixed o --hard), puedes ajustar con precisión hasta dónde quieres deshacer tus pasos.

### Soft mueve HEAD, deja staging y archivos intactos

```
>_
git reset --soft HEAD~1
```

*Este modo elimina el último commit pero deja intactos los cambios en el área de staging. Es como si el commit nunca hubiera ocurrido, pero tus archivos siguen listos para ser confirmados nuevamente. Es ideal si hiciste un commit apresurado y quieres rehacerlo con un mensaje más claro o agruparlo con otros cambios.*

Supón que hiciste esto:

```
>_
git commit -m "fix"
```

Y quieres cambiar el mensaje a algo más descriptivo. Puedes usar:

```
>_
git reset --soft HEAD~1
git commit -m "Corrige validación de email"
```

Así, recuperas los cambios y haces un nuevo commit con un mensaje adecuado.

## Mixed (por defecto): HEAD y staging se actualizan, archivos quedan igual

```
>_
```

```
git reset HEAD~1
```

Este comando también elimina el último commit, pero además saca los cambios del staging, dejándolos solo en tus archivos locales. Es útil si quieres revisar o modificar los cambios antes de volver a añadirlos. Es la opción adecuada cuando necesitas reorganizar tu trabajo sin perder nada.

Supongamos que, después de hacer un commit, te das cuenta de que quieres modificar algunos archivos:

```
>_
```

```
git reset HEAD~1
```

```
# Ahora los archivos están modificados, pero no están en staging
```

Puedes revisar, editar lo necesario y luego volver a añadirlos con git add.

## -Hard: Todo se borra, sin dejar rastro (visible)

```
>_
```

```
git reset --hard HEAD~1
```

Este es el más radical. Elimina el último commit, borra los cambios del staging y del directorio de trabajo. Tu proyecto vuelve exactamente al estado anterior al commit, como si nunca hubiera hecho esos cambios. Úsalo solo si estás completamente seguro de que no necesitas los cambios, o si ya los tienes guardados en otra parte.

Supongamos que te equivocaste completamente con un commit y no quieres quedarte con nada:

```
>_
```

```
git reset --hard HEAD~1
```

Esta acción es destructiva. Los cambios se pierden a menos que los recuperes desde el reflog.

## Git revert

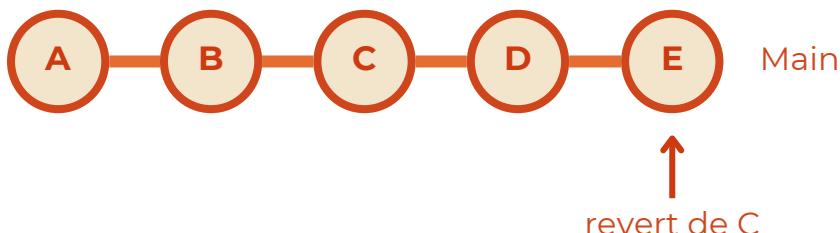
A diferencia de otros comandos como git reset, git revert no elimina los commits del historial. En lugar de eso, crea un nuevo commit que deshace los cambios introducidos por un commit anterior. Es una forma segura de revertir un cambio, ya que mantiene el historial limpio y no reescribe nada. Esto lo convierte en una herramienta ideal cuando trabajas en un repositorio compartido, ya que no afecta el trabajo de otros colaboradores.

Supongamos que tienes el siguiente historial de commits:



Y decides que el commit C necesita ser revertido. Ejecutas:

```
>_
git revert C
```



El commit E contiene los cambios inversos a los del commit C, efectivamente deshaciendo sus efectos, pero sin modificar ni eliminar los commits posteriores.

## Conflictos

En ocasiones, al revertir un commit que introduce cambios complejos, Git puede encontrar conflictos. Cuando esto suceda, Git te indicará qué archivos están en conflicto.

Resuelve los conflictos manualmente editando los archivos afectados. Una vez resueltos, agrega los cambios al área de staging (git add .). Despues, continúa el proceso de reversión con:

```
>_
git revert --continue
```

## Git restore

Es un comando diseñado para ayudarte a deshacer cambios en tu directorio de trabajo y/o en el área de staging. Es especialmente útil cuando quieres descartar cambios no confirmados o cuando necesitas mover archivos fuera del área de staging sin perder el trabajo realizado. A diferencia de otros comandos como git reset o git checkout, git restore está enfocado en restaurar archivos a su versión anterior sin afectar otras partes del historial.

Diferencias clave con otros comandos:

- **git reset**: Afecta tanto al área de staging como al directorio de trabajo, y también se utiliza para deshacer commits.
- **git checkout**: Aunque se puede usar para restaurar archivos a un estado anterior, su propósito principal es cambiar de ramas. git restore se recomienda por su claridad y enfoque específico en la restauración de archivos.

## Básico

Si realizaste cambios en un archivo pero aún no lo has añadido al área de staging, puedes descartarlos sin perder el archivo de tu directorio de trabajo con:

```
>_
git restore archivo.txt
```

*Esto revertirá el archivo archivo.txt a la versión que tenía en el último commit, pero el archivo seguirá existiendo en tu directorio de trabajo.*

## Sacar un archivo del área de staging

Si has añadido un archivo al área de staging pero te arrepientes y no deseas que forme parte del próximo commit, puedes quitarlo del staging con:

```
>_
git restore --staged archivo.txt
```

*Este comando moverá el archivo fuera del área de staging, pero mantendrá los cambios realizados en el archivo dentro del directorio de trabajo.*

## Descartar todos los cambios

Si has realizado modificaciones en varios archivos y decides que no quieres conservar los cambios, puedes descartar todos los cambios en el directorio de trabajo de la siguiente manera:

```
>_
git restore .
```

*Este comando revertirá todos los archivos a su estado en el último commit, eliminando cualquier cambio no confirmado.*

## Recuperar un archivo de un commit anterior

Si necesitas restaurar un archivo a su versión en un commit específico (no necesariamente el último), puedes hacerlo con:

```
>_
git restore --source <commit-id> archivo.txt
```

*Este comando restaurará el archivo archivo.txt a su estado en el commit que especifiques, permitiéndote recuperar una versión anterior del archivo sin afectar el resto del proyecto.*



### CUIDADO!

Aunque git restore no es destructivo en su uso básico, ya que solo descarta los cambios no confirmados, hay que tener precaución cuando uses opciones más complejas, como --source. Si restauras un archivo desde un commit anterior sin especificar correctamente el commit, podrías perder cambios locales importantes.

Es recomendable hacer un backup o verificar los cambios antes de usarlo en situaciones donde estés trabajando con versiones específicas o archivos cruciales.