

Git

Guías de estudio

Guía rápida

Paso a paso (lo más básico – trabajo inicial)

Resumen de trabajo inicial.

Se recomienda antes de iniciar, en caso de ser nuevo proyecto, crear primero el repositorio en Github, en caso contrario clonarlo (explicado en la guía completa).

1. iniciar el repositorio

```
git init
```

2. Agregar todos los archivos

```
git add .
```

3. Confirmar los cambios

```
git commit -m "mensaje descriptivo de lo realizado"
```

- opcional: en caso de querer crear una nueva rama, de lo contrario se sube al [master](#)

```
git -M <nuevaRama>
```

4. Cargar los archivos al repositorio de github

```
git remote add origin <url del repositorio de github>
```

Ejemplo:

```
git remote add origin https://github.com/ches2409/etp.git
```

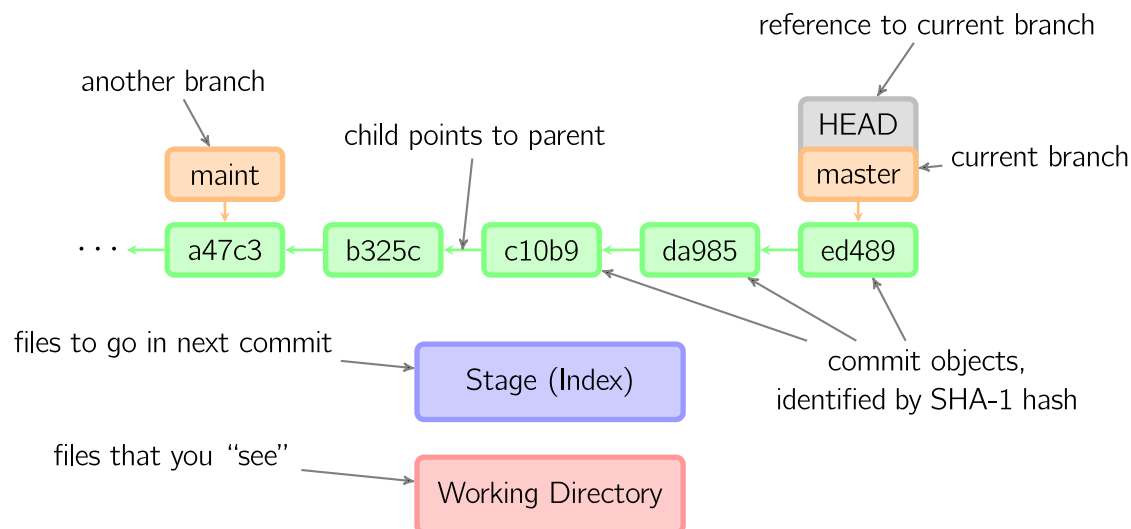
5. Subir los archivos al servidor, en este caso a la rama [master](#)

```
git push origin -u master
```

Guía completa de uso

Guía completa de trabajo con Git

Imágen de convenciones

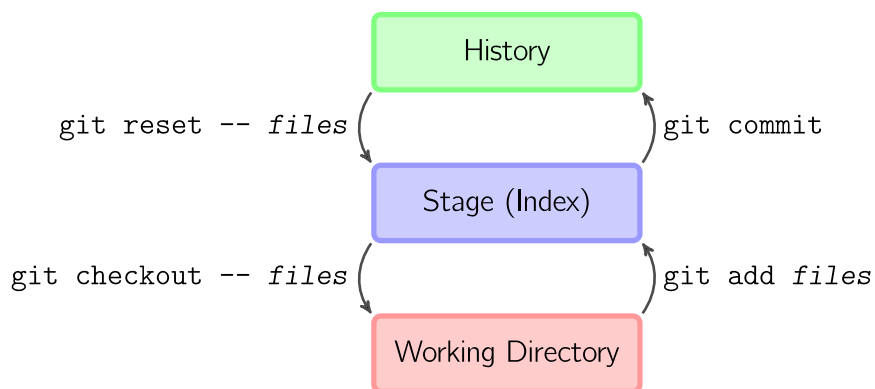


Esquema básico

Git trabaja con `branch` (ramas) como manera de separar desarrollos que pueden ir paralelos y que se irán juntando a medida que estén terminados (la rama principal se conoce como `master`)

La estructura está compuesta por 3 "árboles" administrados por git:

1. `working directory` (directorio de trabajo): El desarrollo local, contiene todos los archivos del proyecto.
2. `stage` (index): Zona intermedia, Es el registro donde se añaden los archivos a señalar.
3. `history` (repository): Donde se guardan todos los cambios de los archivos añadidos en el `index`, lo que se conoce como `commit`, este apunta al último.



El desarrollo típico se resume en:

1. Trabajar en una `branch` concreta (por lo general la `master`)
2. A medida que se avanza se van realizando `stage` y `commit` según la necesidad.
3. Cada cierto tiempo se sincroniza con el repositorio `remote`

workflow:

1. `stage` via `git add`
2. `commit` via `commit -m "mensaje"`
3. `push` via `git push origin master`

Crear el repositorio

Con el directorio creado y abierto en el editor, ejecutar:

```
git init
```

✓ para tener en cuenta

para ver el estado de nuestro proyecto se usa el comando:

```
git status
```

archivos `Untracked` se encuentran aún en el `working directory` falta confirmar

Agregar (add)

Agregar los archivos del `working directory` al `stage`

```
git add <nombreArchivo>
```

también podemos usar los siguiente

✎ opcional a <nombreArchivo>

Agregar todos los archivos al `stage`

- `git add .`
- `git --all`
- `git -A`

en caso de querer remover el archivo del `stage` y regresarlo al `working`:

```
git rm --cached <nombreArchivo>
```

para remover por completo el archivo del `stage` y también del `working`:

```
git rm --f <nombreArchivo>
```

Si el archivo solo se encuentra en el `working` se puede borrar de manera sencilla:

```
git rm <nombreArchivo>
```

✎ Nota

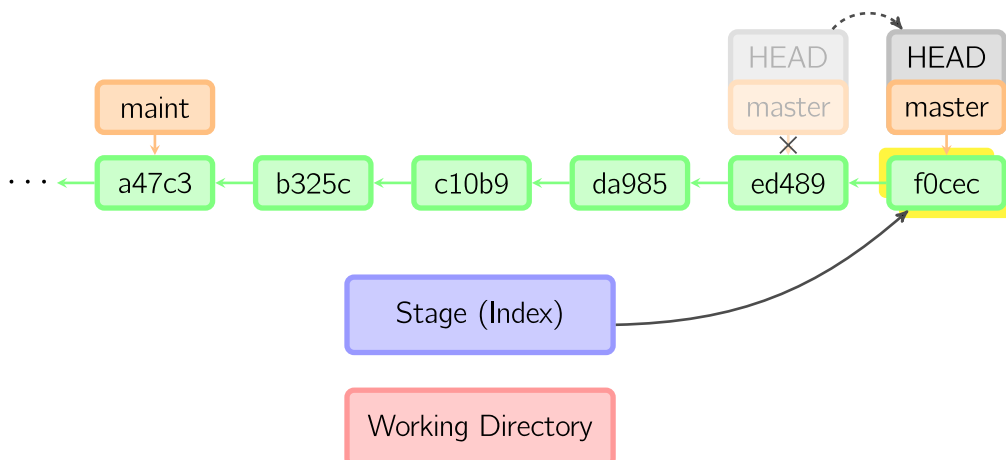
es recomendable para llevar el historial de forma ordenada, realizar un `commit` despues del borrado

Confirmar(commit)

Pasar los archivos del `stage` al `head` (repository)

```
git commit -m "mensaje descriptivo de lo realizado"
```

`git commit`



AMEND

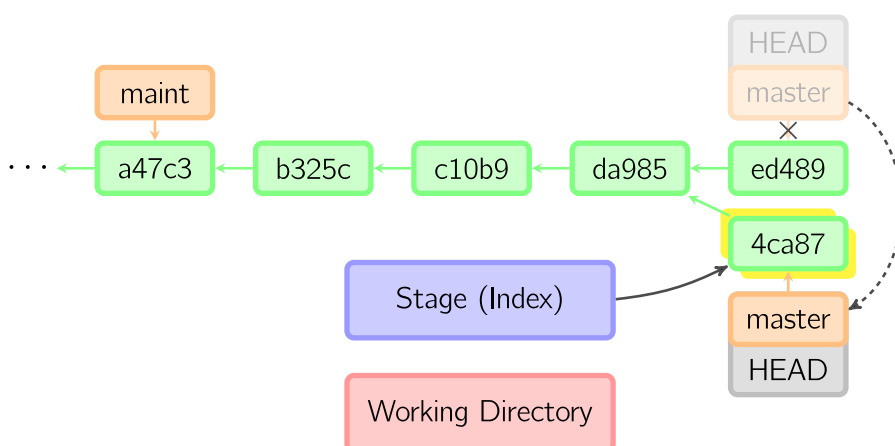
el comando `git --amend` es una manera práctica de modificar la confirmación más reciente. Permite combinar los cambios preparados con la confirmación anterior en lugar de crear una confirmación nueva. También puede usarse para editar el mensaje de la confirmación anterior sin cambiar la instantánea.

```
git commit --amend "mensaje de cambio"
```

para agregar un cambio al commit realizado previamente se usa:

```
git commit --amend -m "mensaje modificado, reemplaza el anterior"
```

`git commit --amend`



versionar o etiquetar (tag)

Para realizar un versionamiento del proyecto se usa la etiqueta `git tag`

existen dos tipos de etiquetas:

- Anotadas
- Ligeras ligeras: la version a etiquetar, Ejemplo

```
git tag 0.5
```

Anotada: contiene nombre de referencia y la anotación o mensaje, Ejemplo

```
git tag -a 0.5 -m "Versión estable del proyecto"
```

para ver los tag's realizados:

```
git tag -l
```

Con esto etiquetamos el ultimo commit que se realizó.

Para etiquetar versiones pasadas del proyecto, se hace conociendo el `SHA-1 hash` del `commit`, el cual se copia y se pega al comando:

```
git tag <numeroVersion> <SHA-1 hash>
```

ejemplo: etiqueta ligera

```
git tag 0.6 8db3310839002cf22f19b412739b613ccd2a38ac
```

para borrar tag:

```
git tag -d <numeroVersionABorrar>
```

```
git tag -d 0.6
```

para renombrar tag:

```
git tag -f -a <numeroVersionABorrar> -m <"mensaje"> <SHA-1 hash>
```

```
git tag -f -a 0.6 -m "correccion de versión" 8db3310839002cf22f19b412739b613ccd2a38ac
```

luego de esto se borra la version anterior a la que se modificó

```
git tag -d 0.6
```

historia del proyecto (log)

Con el comando `git log` se revisa todos los cambios del proyecto

estructura del log

línea 1 se muestra el `SHA-1 hash`

commit `8db3310839002cf22f19b412739b613ccd2a38ac`

línea 2 Author: <nombreAutor> <correo electrónico>

línea 3 Date: <fecha y hora>

línea 3 <Mensaje del commit>

El comando `git log` se puede personalizar

- resume el log: el `SHA-1 hash` lo muestra en un conjunto de caracteres mas pequeño (se puede usar también para hacer las referencias) e indica el mensaje del commit.

```
git log --oneline>
```

- mostrar el grafico de como se avanza en la historia

```
git log --oneline --graph>
```

- hacer log de commit's que se quieran visualizar

```
git log -<numeroCommit>
```

para cerrar la pantalla del `log` en el terminal, basta con presionar la tecla `q`

Diferencias entre versiones (diff)

Para ver los cambios entre los commit se usa el comando `git diff`

- estado inicial con el git que se compare:

```
git diff <SHA-1 hash>
```

- Comparar dos commit's

```
git diff <SHA-1 hashUno> <SHA-1 hashDos>
```

re-escribir (reset)

para realizar cambios o sobrescribir dentro los commit se usa el comando `git reset`

para tener en cuenta

este procedimiento es de cuidado debido a que se rescribe el proyecto y por lo tanto se puede perder cosas del mismo.

hay tres tipos principales de reset

- reset soft
- reset mixed
- reset hard

RESET SOFT

Se quita desde un commit a traves su `SHA-1 hash`, dejando el commit al que se hizo referencia por su `hash` en el estado original.

el `soft` quita el commit pero no los archivos modificados ni los archivos de `stage`, quedando preparados para hacer nuevamente el `commit`

en resumen quita un cambio pero lo mantiene en `stage`

```
git reset --soft <SHA-1>
```

Ejemplo de uso

por medio del `log` ubicamos el `hash` a quitar.

```
43f3570 agregado nuevo hero // commit a quitar
8db3310 agregado el header // hash a llamar
052f52b inicializar el landing
```

```
git reset --soft 8db3310
git log --oneline
```

```
8db3310 agregado el header
052f52b inicializar el landing
```

RESET MIXED

Descarta cambios, quita los commit's y los archivos del `stage` al contrario de como lo hace `soft`, los de deja en el `working directory`, por lo tanto hay que realizar nuevamente el `add` y el `commit`

```
git reset --mixed <SHA-1>
```

RESET HARD

Borra todo, tanto commit como archivos del `stage` y del `working directory`

- actua dentro del momento en el que se encuentre (dentro del head)

```
git reset --hard
```

si los archivos solo se encuentran el workig directory y se encuentra en estado **untracked** (no han sido usados, recientemente agregados)

- con esta opcion se borra de manera permanente todo lo que interviene en ese commit y no se puede deshacer

```
git reset --hard <SHA-1>
```

⚡ para tener cuenta

en caso de haber borrado con **--hard**, la unica forma para volver a ese estado es tener una copia del log y repetir el paso desde el ultimo hash o del que quiere repetir

ramas (branch)

Las ramas son utilizadas para desarrollar funcionalidades aisladas unas de otras. La rama **master** es la rama por defecto cuando se crea un repositorio, se crean nuevas durante el desarrollo y se fusionan a la rama principal cuando este termina.

crear ramas

```
git branch <nombreRama>
```

para listar ramas (-l, --list)

```
git branch --list
```

borrar ramas (-d, --delete)

- no permite borrar cuando hay cambios dentro de ella (commit)

```
git branch -d <nombreRama>
```

- para forzar el borrado se usa (--delete --force, -D)

```
git branch -D <nombreRama>
```

renombrar ramas (-m, --move)

```
git branch -m <nombreRamaOriginal> <nombreRamaFinal>
```

Movimiento entre ramas (checkout)

moverse entre ramas

```
git checkout <nombreRama>
```

tambien se puede usar el **checkout** para moverse entre commit, por medio este podemos movernos a ese commit y revisar como estuvo en ese tiempo el proyecto sin borrar nada.(crea la rama virtual con el nombre del SHA-1 hash)

```
git checkout <hash>
```

crear rama y ubicarse en ella

```
git checkout -b <nombreRama>
```

Con checkout se puede resetear modificaciones

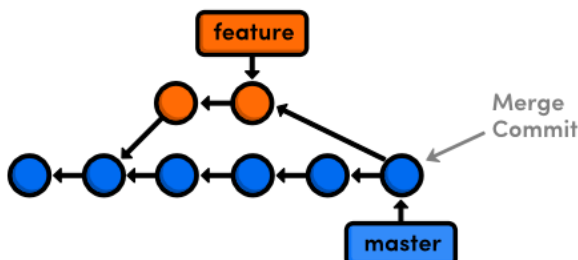
```
git checkout -- <nombreArchivoaQuitarModificacion>
```

Trabajar entre ramas

Una vez terminado el trabajo en cada una de las ramas se procede a realizar la union de todas la ramas.

Ubicarse en la rama que va recibir los cambios (master)

```
git merge <ramaAMezclar>
```

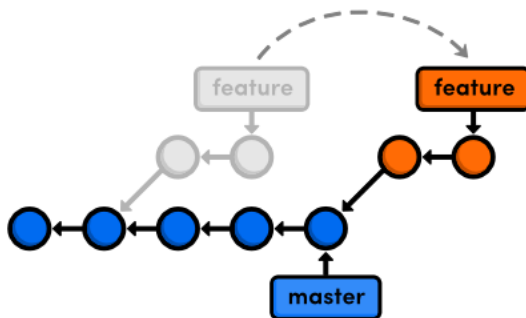


metodos internos de git a usar en el merge

- **Fast-forward**: la rama que se va a unir, parte directamente desde la rama master, continuación directa.
- **Auto-mergin**: abre el ditor para confirmar cambios (commit), este tipo de combinacion se da cuando la rama ha partido del master pero ya se han realizado cambios en ella.
- **Auto-merging CONFLICT**: Cuando se mezclan con archivos iguales, se revisan los cambios manuales y se deja una version.

reescribir la historia (rebase)

Una alternativa a **merge**, en lugar de enlazar ramas con **commit** de merge, el **rebase** mueve completamente la rama con la nueva característica hacia la punta del **master**



```
git rebase <ramaAMezclar>
```

Ventajas:

- Resulta en una historia lineal del proyecto
- Oportunidad de limpiar commits locales

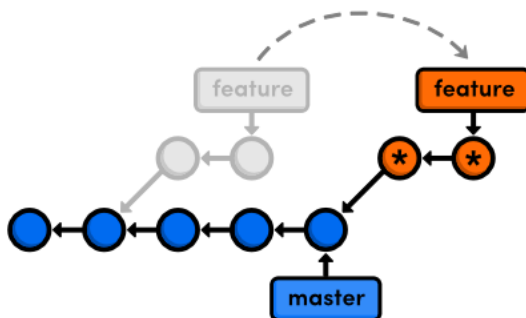
contras:

- hace la unión de todas las ramas

⚡ Para tener en cuenta

Al no realizarse de manera correcta, puede ser una de las operaciones más peligrosas que se le puede realizar a un repositorio

hacer un rebase no mueve los commits en una nueva rama. En su lugar, crea nuevos commits que contienen los cambios deseados.



★ = Brand New Commits

Después de hacer un rebase, los commits en **feature** tendrán diferentes hashes. Esto significa que no solo posicionamos una rama —literalmente, reescribimos la historia del proyecto. Esto es un efecto secundario muy importante de **rebase**.

INTERACTIVO (INTERACTIVE REBASE -I)

El rebase interactivo deja definir precisamente como cada commit será movido hacia la nueva base.

```
git rebase -i <ramaAMezclar>
```

De esta manera se confirman las modificaciones (commit), es útil para cambiar los mensajes de los commits anteriores así como reorganizar el historial de confirmaciones, equivalente al **--amend**.

Cambios temporales (stash)

El comando `stash` almacena temporalmente (o guarda en un stash) los cambios que se hayan efectuado en el código en el que se está trabajando. Guardar los cambios en stashes resulta práctico si se tiene que cambiar rápidamente de contexto, y no se tiene listo el código para confirmar los cambios.

```
git stash
```

al regresar a la rama donde se dejó, se puede hacer una visualización de los stash:

```
git stash list
```

```
listado-stash
```

```
stash@{0}: WIP on <branch>: <hash> <commit>
stash@{1}: WIP on <branch>: <hash> <commit>
```

`stash@{#}`: identificador

`WIP`: Trabajo en curso

se recomienda comentar los `stash` con una descripción mediante el comando

```
git stash save "comentario del stash"
```

- ver las diferencias de un stash

```
git stash show
```

Otra opción es utilizar la opción `-p` (o `--patch`) para ver todas las diferencias de un stash:

```
git stash show -p
```

- eliminar

```
git stash drop <identificador>
```

- eliminar todos los stash

```
git stash clear
```

- aplicar el ultimo cambio realizado (`stash@{0}`)

```
git stash apply
```

- para aplicar cambio de un determinado stash se invoca su numero de stash:

```
git stash apply <identificador>
```

- Crear una nueva rama a partir del stash

```
git stash branch
```

Seleccionando commits (cherry-pick)

`cherry-pick` es un potente comando que permite que las confirmaciones arbitrarias de Git se elijan por referencia y se añadan al actual `HEAD` de trabajo. La ejecución de `cherry-pick` es el acto de elegir una confirmación de una rama y aplicarla a otra. `cherry-pick` puede ser útil para deshacer cambios.

Por ejemplo, suponiendo que una confirmación se aplica accidentalmente en la rama equivocada. Se Puede cambiar a la rama correcta y ejecutar `cherry-pick` en la confirmación para aplicarla a donde pertenece.

Para tener en cuenta

`cherry-pick` es una herramienta útil, pero no siempre es una práctica recomendada, La ejecución de `cherry-pick` puede generar duplicaciones de confirmaciones.

```
git cherry-pick
```

Caso de uso

Ejemplo de uso del `cherry-pick`

Identificación

```
6544057 (HEAD -> responsive) cambio de stilos responsive
0894d24 hotfix2 // archivo de arreglo no corresponde a la rama
a8657f4 cambio2 responsive
7ce6c0e cambio1 responsive
```

paso a seguir:

sacar el commit del "hotfix2" y reubicarlo en `master`

Optimo

```
git checkout master
// crear rama de arreglo desde master y ubicación en ella
git checkout -b hotfix2
//en esta rama se debía generar el cambio
```

ubicado en la rama donde tendría que estar el arreglo:

Reubicación

```
(hotfix2) git cherry-pick 0894d24
```

finalizar el arreglo con un merge

finalizando

```
(hotfix2) git checkout master
(master) git merge hotfix2
```

Opciones para trabajar con `cherry-pick`:

- `-edit`: Git solicitará un mensaje de confirmación antes de aplicar la operación `cherry-pick`.
- `--no-commit`: Se ejecuta el comando `cherry-pick`, pero en lugar de hacer una nueva confirmación, se mueve el contenido de la confirmación de destino al directorio de trabajo de la rama actual.
- `--signoff`: Añade una línea de firma 'signoff' al final del mensaje de confirmación de `cherry-pick`.

`cherry-pick` también cuenta con una variedad de opciones de estrategia de fusión (<https://www.atlassian.com/es/git/tutorials/using-branches/merge-strategy>)