

# **GIT- Una Guía Algo Rápida**

Javier Alan Ventura

2025



## Presentación

La mejor manera de aprender algo es enseñando, así que aprendamos juntos, espero el material les resulte útil, y además que logren entenderlo en gran medida, pero recuerda complementar con la documentación oficial de git, y otros libros y artículos que hay en internet que también dejare en la bibliografía, sin más que decir continuemos

## Índice

1. Presentación
2. Objetivos
3. Contenido
4. Conclusión
5. Bibliografía

## Objetivos

- Dominar git al final de este texto
- Aprender la definición y conceptos de git y su uso
- Instalación de git
- Usar git mediante la terminal y aprender a usar algunos comandos básicos bash.
- Aprender comandos git, aprender a crear commits, ramas, fusionar ramas con diferentes métodos y enfoques, resolver conflictos.
- Realizar algunos ejercicios.

## Contenido

### Definición de git

**Git es un sistema de control de versiones**, lo que significa que es una herramienta que nos ayuda a gestionar los cambios realizados en el código o en un proyecto a lo largo del tiempo. Dado que el código se modifica constantemente, especialmente cuando varias personas trabajan en él, Git permite registrar quién hizo cada cambio, cuándo lo hizo y qué se modificó.

Además, nos da la posibilidad de acceder a versiones anteriores del proyecto en caso de que necesitemos volver atrás

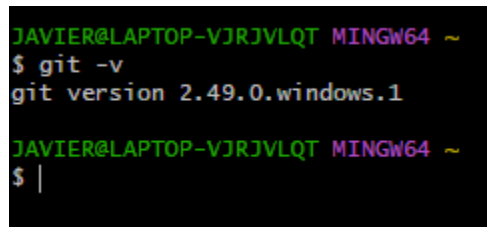
## Instalación de git

Para la instalación de git, debemos resaltar que este es un programa, el cual debemos ejecutar en nuestra máquina, para eso debemos ir a la pagina oficial, y descargarlo para tu SO, ya sea Windows, Linux o MacOS

<https://git-scm.com/downloads>

El siguiente paso es instalarlo, en nuestro caso abran muchas opciones y casillas para escoger, pero escogeremos las que están por defecto.

Para verificar la instalación y la versión de git, escribimos en la terminal de git bash el comando git -v



```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~  
$ git -v  
git version 2.49.0.windows.1  
  
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~  
$ |
```

Podemos usar git en un entorno gráfico, pero para entenderlo mejor empezaremos usando comandos en la terminal.

## Comandos

Una terminal o una consola, es una herramienta que nos permite interactuar con nuestro sistema operativo mediante formato texto.

Algunos de los comandos que mas se usan en el interprete de comandos bash son:

**ls:** Muestra una lista con los archivos y carpetas del directorio actual, definamos directorio, son las carpetas con las que trabajamos.

**cd:** Nos ayuda a movernos entre directorios del sistema de archivos. Se usa de la siguiente manera:

cd nombre\_de\_carpetas (cambia a una carpeta especifica)

cd .. (va a la carpeta anterior, es decir retrocede una carpeta)

cd / (va al directorio raíz del sistema)

**cd ~** (va a tu carpeta personal home)

**pwd:** muestra la ruta actual del directorio

**mkdir:** Este comando crea una carpeta en el directorio actual, debemos usarlo de la forma: `mkdir nueva_carpeta`

**touch:** Crea un archivo vacío en el directorio actual, por ejemplo `touch nuevo_archivo.txt`

**rm:** Elimina ya sea un archivo o un directorio, la manera de usarlo es `rm` seguido del nombre del archivo o directorio, si queremos eliminar un archivo con todo su contenido lo hacemos así: `rm -r nuevo_archivo`, `-r` elimina el contenido dentro del archivo.

**cp:** se usa para copiar archivos o carpetas en la terminal.

```
$ ls
archivo2.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/archivo1
$ cp archivo2.txt archivo3.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/archivo1
$ ls
archivo2.txt  archivo3.txt
```

Creamos un `archivo3.txt` que es la copia de `archivo2.txt`

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cd archivo1

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/archivo1
$ ls
archivo2.txt  archivo3.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/archivo1
$ cp archivo2.txt ~/desktop/nuevo_archivo

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/archivo1
$ cd ..

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cd nuevo_archivo

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/nuevo_archivo
$ ls
archivo2.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/nuevo_archivo
$
```

En aquí copiamos un archivo archivo2.txt de la carpeta archivo1 a otra carpeta llamada nuevo\_archivo.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ mkdir nueva_carpeta

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cd nueva_carpeta

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/nueva_carpeta
$ touch texto1.txt texto2.txt texto3.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/nueva_carpeta
$ ls
texto1.txt  texto2.txt  texto3.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/nueva_carpeta
$ cp -r nueva_carpeta carpeta_copia
cp: cannot stat 'nueva_carpeta': No such file or directory

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/nueva_carpeta
$ cd ..

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cp -r nueva_carpeta carpeta_copia

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cd carpeta_copia

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta_copia
$ ls
texto1.txt  texto2.txt  texto3.txt
```

En este ultimo lo que hicimos es copiar todo el contenido de una carpeta a otra carpeta llamada carpeta\_copia.

mv: Sirve para mover o renombrar archivos.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cd nueva_carpeta

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/nueva_carpeta
$ ls
texto1.txt  texto2.txt  texto3.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/nueva_carpeta
$ mv texto1.txt texto_renombrado.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/nueva_carpeta
$ ls
texto2.txt  texto3.txt  texto_renombrado.txt
```

Renombramos al archivo texto1.txt a texto\_renombrado.txt

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta_origen
$ ls
texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta_origen
$ mv texto.txt ~/desktop/carpeta_destino

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta_origen
$ ls

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta_origen
$ cd ..

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cd carpeta_destino

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta_destino
$ ls
texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta_destino
$

```

En este segundo ejemplo vemos que tenemos un archivo texto.txt en una carpeta llamada carpeta\_origen, lo que hacemos es escribir en la terminal mv texto.txt /carpeta\_destino, esto mueve nuestro archivo a la nueva carpeta.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cd carpeta1

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta1
$ ls
texto1.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta1
$ mv texto1.txt ~/desktop/carpeta2/texto_renombrado.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta1
$ cd ..

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cd carpeta2

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta2
$ ls
texto_renombrado.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta2
$

```

Podemos mover y renombrar al mismo tiempo.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ mv carpeta1 ~/desktop/carpeta2

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cd carpeta2

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta2
$ ls
carpeta1/  texto_renombrado.txt
```

Y también podemos mover una carpeta a otra.

Son estos algunos de los comandos básicos que tenemos que nos ayudaran a movernos e interactuar entre carpetas y archivos de nuestra máquina.

Si quieres un pdf similar a este enfocando en comandos bash y uso de la terminal, envíame un mensaje a mi correo [javier.alan.ventura12@gmail.com](mailto:javier.alan.ventura12@gmail.com) para enviarte o preparar una pequeña guía.

## Configuración git y guardar el usuario y correo electronico:

Cuando hay mas de una persona trabajando en un mismo proyecto, es necesario saber quien es la persona que hace cambios, es decir que las configuraciones git son la **identidad del usuario** que se asociará a los *commits* (cambios que haces en tu proyecto).

Al trabajar con Git, cada usuario debe poseer su propio identificador, que se corresponderá con su nombre y dirección de correo electrónico. Estos identificadores se utilizan para etiquetar cada cambio realizado en el proyecto, lo que permite una fácil identificación de cada acción registrada en el sistema.

**Git global:** Es darle una instrucción a git de que use toda la información es nuestros proyectos, el nombre y el correo electrónico.

Para hacer eso escribimos en nuestra terminal:

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~
$ git config --global user.name "JavierV"

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~
$ git config --global user.email "javier.alan.ventura12@gmail.com"
```

Los valores que están dentro de comillas son donde va tu nombre de usuario y tu correo electrónico. Para usar git este paso es indispensable.

## Inicialización de un repositorio

Cuando tengas la carpeta con la que vas a trabajar en tu proyecto, lo primero que debes hacer es ejecutar el siguiente comando: **git init**, esto crea un repositorio git vacío **.git**, dentro de esa carpeta guarda toda la información necesaria para llevar el historial de tu proyecto.

Es un paso necesario para trabajar con git y el sistema lo reconozca como un directorio con git operativo.

Vamos a entender que es un repositorio, un repositorio es una carpeta donde git guarda tu proyecto y su historial de cambios, es como una aplicación en donde sacan la ultima versión, pero esta presenta errores, entonces nos permite volver a versiones anteriores.

El primer paso es en nuestra carpeta en la cual estemos trabajando, hacemos

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ejemplo
$ git init
Initialized empty Git repository in C:/Users/JAVIER/Desktop/ejemplo/.git/
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ejemplo (master)
$
```

Notaremos la palabra **master**, significa que estás trabajando en una **rama principal del repositorio**.

## Ramas

Seguramente muchos de ustedes vieron la serie de *Loki*. En ella, hay escenas donde se muestran líneas o ramas temporales: diferentes caminos que surgen a partir de cambios en la línea principal del tiempo.

En Git sucede algo similar. Las ramas son líneas de trabajo independientes que no afectan a las demás, ni a la rama principal.

Esto nos permite hacer modificaciones, experimentar, y si es necesario, volver en el tiempo: corregir errores o regresar el proyecto a una versión anterior.

## Guardado

**Commits:** Se podría interpretar como una fotografía del estado de nuestro proyecto en determinado momento, por ejemplo, en varios videojuegos puedes guardar la partida, y en otro momento volver al lugar en la que dejaste la partida.

El siguiente comando a aprender es el **git add**, cuando usamos este comando estamos seleccionando los archivos a los cuales los incluiremos en el commit, ojo que no los estamos guardando, solo lo estamos seleccionando para hacer el commit con el siguiente comando.



**git commit -m "Descripción del cambio":** Este comando lo que hace es guardar todos los cambios o selecciones que hicimos con `git add`, notemos que lo que le decimos a bash al usar **-m** es que usaremos un mensaje corto que describa el cambio que hicimos, ese mensaje va dentro de las comillas dobles.

Otro comando importante es el de **git status**, el cual te muestra el estado actual de tus proyectos. Que información de da:

- Qué archivos han cambiado
- Cuáles ya están listos para guardar (`git add`)
- Cuáles aún no has añadido
- En qué rama estás trabajando

Hagamos un ejemplo completo, presta mucha atención

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ mkdir ej_commits

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cd ej_commits

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ej_commits
$ git init
Initialized empty Git repository in C:/Users/JAVIER/Desktop/ej_commits/.git/

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ej_commits (master)
$ touch texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ej_commits (master)
$ ls
texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ej_commits (master)
$ echo "este es un texto" >> texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ej_commits (master)
$ cat texto.txt
este es un texto

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ej_commits (master)
$ git add texto.txt
warning: in the working copy of 'texto.txt', LF will be replaced by CRLF the next time Git touches it

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ej_commits (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   texto.txt
```

Primeramente, creamos una carpeta llamada `ej_commits`, en esa carpeta lo que hicimos fue crear un archivo `texto.txt`, en el cual agregamos un texto “esto es un texto”.

Fíjese que después de haber hecho esos pasos, usamos el comando git add texto.txt, el cual selecciona ese archivo para hacerle un commit, y por ultimo usamos el comando git status, el cual nos menciona en la última línea que tenemos un archivo preparado para hacerle un commit.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ej_commits (master)
$ git commit -m "esta es la primera commit que hago"
[master (root-commit) 0934704] esta es la primera commit que hago
1 file changed, 1 insertion(+)
create mode 100644 texto.txt
```

Acá ya creamos el commit a nuestro archivo preseleccionado, e imprimimos el mensaje de “esta es la primera commit que hago”

Ahora que ya creamos nuestro commit, podemos ver los commits con los siguientes comandos:

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ej_commits (master)
$ git log
commit 0934704cd02888817b87f82b8535a82704b21fcc (HEAD -> master)
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Thu May 8 19:43:59 2025 -0400

    esta es la primera commit que hago

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ej_commits (master)
$ git log --oneline
0934704 (HEAD -> master) esta es la primera commit que hago
```

Estos nos muestran el estado general de nuestro proyecto, los archivos que tenemos preseleccionados y los que hicimos un commit, además de que nos muestran el nombre y correo de la persona que hizo el cambio.

## Comandos de inspección y seguimiento de cambios en Git

### Git diff

Prestemos total atención a los siguientes, que son importantes para entender el manejo y funcionamiento de los comandos que nos ayudan a ver los cambios que realizamos en nuestro proyecto a través de commits y hacerles un seguimiento.

Aplicaremos todos los pasos que hasta ahora hemos aprendido:

1. Primeramente crearemos un directorio con el nombre de “carpeta”, luego nos dirigimos a esa carpeta

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ mkdir carpeta

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cd carpeta

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta
```

2. Luego en esa carpeta crearemos un archivo texto.txt, en el cual insertaremos un texto, el cual esta entre comillas, luego verificamos que efectivamente ese texto esta en el archivo txt.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta
$ touch texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta
$ echo "primera fila de texto que tenemos en nuestro archivo" >> texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta
$ cat texto.txt
primera fila de texto que tenemos en nuestro archivo
```

3. El comando git init inicializa un nuevo repositorio de Git en la carpeta actual. Esto le indica a Git que comience a rastrear los cambios en esa carpeta y crea una subcarpeta oculta llamada .git, donde se almacenará todo el historial, la configuración y las versiones de los archivos. A partir de ese momento, esa carpeta se convierte en un repositorio Git.

También es importante aclarar que la palabra master indica que estamos trabajando en la rama principal del proyecto. Esta rama representa la línea base o raíz sobre la cual se desarrollan y organizan las demás versiones del código

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta
$ git init
Initialized empty Git repository in C:/Users/JAVIER/Desktop/carpeta/.git/

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
```

4. Ahora seleccionamos el archivo texto.txt para incluirlo en el próximo commit. En este punto, aún no se ha creado el commit; simplemente hemos añadido el archivo a una **zona de preparación** (staging area). Podemos pensar en esta etapa como un “pre-commit”, donde los archivos están en una lista de espera hasta que confirmemos los cambios definitivamente con el comando git commit.

En la salida de la consola hay un **warning**, no entiendo muy bien a lo que se refiere, creo que es algo que tiene que ver con los espacios o saltos de línea, pero que no afecta en nada nuestro archivo, creo xd.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git add texto.txt
warning: in the working copy of 'texto.txt', LF will be replaced by CRLF the next time Git touches it
```

5. Hasta este punto hemos logrado hacer un precommit, que aun no es un commit, ejecutaremos el siguiente comando: **git diff**

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git diff texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
```

Como podemos observar no hizo nada, ya que **diff** significa “diferencia”, y en nuestro archivo texto.txt no modificamos nada, ahora intentemos nuevamente modificando nuestro texto.txt

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ echo "segunda linea de texto que tenemos en nuestro archivo" >> texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ cat texto.txt
primera fila de texto que tenemos en nuestro archivo
segunda linea de texto que tenemos en nuestro archivo
```

Agregamos una segunda línea de texto, en el cual podemos ver exactamente que si cambio nuestro texto.

Recordemos que estamos en un precommit, que aun no hicimos un commit, pero sin embargo modificamos nuestro archivo. Para ver todas las modificaciones que se han hecho en un archivo pero que **aún no han sido registradas en un commit**, es decir, que no están guardadas en la carpeta .git, usamos el comando git diff

```
$ git diff texto.txt
warning: in the working copy of 'texto.txt', LF will be replaced by CRLF the next time Git touches it
diff --git a/texto.txt b/texto.txt
index 721cbc4..7ed6598 100644
--- a/texto.txt
+++ b/texto.txt
@@ -1,2 @@
 primera fila de texto que tenemos en nuestro archivo
+segunda linea de texto que tenemos en nuestro archivo
```

Enfoquémonos en las 2 ultimas líneas, penúltima línea es el estado de nuestro archivo antes de hacer los cambios, y la línea verde es la modificación que hicimos, en este caso aparece un +, que nos indica que agregamos la línea de texto verde.

6. Ahora vamos a reemplazar ambas líneas por un texto nuevo

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ echo "texto que reemplaza ambas lineas de codigo" > texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ cat texto.txt
texto que reemplaza ambas lineas de codigo
```

Notemos que hemos eliminado dos líneas de texto y agregamos una nueva, ¿o no?

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git diff texto.txt
warning: in the working copy of 'texto.txt', LF will be replaced by CRLF the next time Git touches it
diff --git a/texto.txt b/texto.txt
index 721cbc4..8d8a70c 100644
--- a/texto.txt
+++ b/texto.txt
@@ -1,1 @@
-primera fila de texto que tenemos en nuestro archivo
+texto que reemplaza ambas lineas de codigo
```

¿Por qué solo aparece que eliminé una sola línea de texto? ¿Dónde está la segunda?

En realidad, sí eliminamos ambas líneas, pero git diff compara los archivos línea por línea. Si las nuevas líneas no se parecen a las anteriores, Git no puede relacionarlas directamente.

Por eso, si eliminamos dos líneas y agregamos dos nuevas, git diff nos mostrará las cuatro: dos líneas en rojo (las eliminadas) y dos en verde (las nuevas).

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ echo -e "texto3\ntexto4" > texto2.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ cat texto2.txt
texto3
texto4
```

En un nuevo archivo de texto agregamos dos líneas de texto.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git add texto2.txt
warning: in the working copy of 'texto2.txt', LF will be replaced by CRLF the next time Git touches it
```

Como estamos trabajando en un nuevo archivo de texto, no olvidemos que para usar el comando diff, este debe estar en un precommit, es decir un git add o un commit para comparar entre estos.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ echo -e "texto5\ntexto6" > texto2.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git diff texto2.txt
warning: in the working copy of 'texto2.txt', LF will be replaced by CRLF the next time Git touches it
diff --git a/texto2.txt b/texto2.txt
index e09dd03..d360df1 100644
--- a/texto2.txt
+++ b/texto2.txt
@@ -1,2 +1,2 @@
-texto3
-texto4
+texto5
+texto6
```

Ahora reemplazamos las anteriores líneas de texto por dos nuevas, y al momento de usar git diff nos mostrara todos los cambios en nuestro archivo.

Si tenemos cambios que no son línea por línea podemos usar el comando **git diff --word-diff**, el cual de manera mas clara y directa nos da toda la información.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git diff --word-diff texto2.txt
warning: in the working copy of 'texto2.txt', LF will be replaced by CRLF the next time Git touches it
diff --git a/texto2.txt b/texto2.txt
index e09dd03..d360df1 100644
--- a/texto2.txt
+++ b/texto2.txt
@@ -1,2 +1,2 @@
[-texto3-]
[-texto4-]{+texto5+}
{+texto6+}
```

**Plus:** cuando al ejecutar el comando git diff no aparece nada, entonces lo mas probable es que no hayas hecho un precommit, es decir usar el comando git add, y otra de las razones es que ya has commitado ese archivo, es decir que previamente ya has hecho un git commit y no has hecho cambios desde entonces.

## Git show:

Creamos otra carpeta y un archivo de texto, y hacemos los pasos hasta el precommit:

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta
$ git init
Initialized empty Git repository in C:/Users/JAVIER/Desktop/carpeta/.git/

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git add com.txt
warning: in the working copy of 'com.txt', LF will be replaced by CRLF the next time Git touches it

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   com.txt
```

En la consola creamos el commit con git commit

```
$ git commit -m "commit numero 1"
[master (root-commit) a4c1daf] commit numero 1
1 file changed, 1 insertion(+)
create mode 100644 com.txt
```

```
$ git show
commit a4c1daf095c512c595eac74ee4b440c9d4338c3f (HEAD -> master)
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Sat May 10 08:37:34 2025 -0400

    commit numero 1

diff --git a/com.txt b/com.txt
new file mode 100644
index 0000000..3379fd4
--- /dev/null
+++ b/com.txt
@@ -0,0 +1 @@
+primera línea de parrafo
```

Con el comando **git show** lo que hicimos fue que se nos mostrara en la salida el commit en el que estamos, es decir que te muestra tu ultimo commit.

Como vemos en la imagen, nos da información del autor y la fecha en la que se hizo la modificación, seguido a eso nos muestra el mensaje con el que asociamos el commit, en nuestro caso **commit numero 1**, y al final nos muestra la modificación de nuestro archivo, que como es nuevo agregamos una línea de texto.

Aclaremos que git show muestra información detallada de un commit específico en nuestro directorio (rama master). Como solo tenemos un commit, nos muestra los cambios realizados en ese archivo solamente

Si tuvieras dos archivos txt, a los que hiciéramos un commit, al ejecutar el comando de git show nos mostraría información de ambos, detalles de ambos como los autores y fecha de modificación, los cambios que se hicieron, y nos mostraría en el orden en el cual guardamos cada commit independientemente del archivo.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git show
commit 7a26740db6c778cbf6703c98c1118b6a15afa904 (HEAD -> master)
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Sat May 10 10:55:35 2025 -0400

    commit numero2

diff --git a/com.txt b/com.txt
index 3379fd4..0fc9230 100644
--- a/com.txt
+++ b/com.txt
@@ -1,2 @@
 primera línea de parrafo
+segunda línea de código
```

Ahora hicimos un segundo commit, el cual llamamos **commit numero 2**, y agregamos una línea de texto segunda línea de texto.

```
$ git show
commit 7a26740db6c778cbf6703c98c1118b6a15afa904 (HEAD -> master)
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Sat May 10 10:55:35 2025 -0400

    commit numero2

diff --git a/com.txt b/com.txt
index 3379fd4..0fc9230 100644
--- a/com.txt
+++ b/com.txt
@@ -1,2 @@
 primera línea de parrafo
+segunda línea de código

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ |
```

Git show nos muestra nuestro ultimo commit, en este caso **commit número 2**.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ echo "tercera y unica línea de texto" > com.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ cat com.txt
tercera y unica línea de texto

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git add com.txt
warning: in the working copy of 'com.txt', LF will be replaced by CRLF the next time Git touches it

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git commit -m "commit numero 3"
[master 1f1af0b] commit numero 3
1 file changed, 1 insertion(+), 2 deletions(-)
```



Creamos un tercer commit eliminando lo que teníamos y reemplazando por la tercera línea de texto.

Como podemos ver git show es muy importante para hacer referencia a nuestro commit, ver los cambios y la información que esta nos proporciona, ahora veremos sus variaciones.

Git show HEAD: Al igual que git show muestra el ultimo commit que hicimos

```
$ git show HEAD
commit 1f1af0bd60ff5c25c2161859bc3cc639ae4f2e14 (HEAD -> master)
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Sat May 10 11:56:51 2025 -0400

    commit numero 3

diff --git a/com.txt b/com.txt
index 0fc9230..2bd4d6d 100644
--- a/com.txt
+++ b/com.txt
@@ -1,2 +1 @@
-primer línea de parrafo
-segunda línea de código
+tercera y única línea de texto
```

### Git show de un archivo commitado en específico:

Para ver el commit de un archivo en específico usamos el comando **git show HEAD – archivo**

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git show HEAD -- com.txt
commit 1f1af0bd60ff5c25c2161859bc3cc639ae4f2e14 (HEAD -> master)
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Sat May 10 11:56:51 2025 -0400

    commit numero 3

diff --git a/com.txt b/com.txt
index 0fc9230..2bd4d6d 100644
--- a/com.txt
+++ b/com.txt
@@ -1,2 +1 @@
-primer línea de parrafo
-segunda línea de código
+tercera y única línea de texto
```

Como vemos acá nos muestra el ultimo commit que hicimos a nuestro com.txt.

### Git show print

Si queremos ver el contenido completo de un archivo commitado usamos el comando **git show HEAD:archivo**

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git show HEAD:com.txt
tercera y unica linea de texto
```

### Ir a un commit en especifico

Para poder dirigirnos a un commit en específico, primero debemos aprender un comando, **git log**.

**Git log** sirve para ver el historial de commits en un repositorio. Te muestra una lista de commits detalladamente.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git log
commit 1f1af0bd60ff5c25c2161859bc3cc639ae4f2e14 (HEAD -> master)
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Sat May 10 11:56:51 2025 -0400

    commit numero 3

commit 7a26740db6c778cbf6703c98c1118b6a15afa904
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Sat May 10 10:55:35 2025 -0400

    commit numero2

commit a4c1daf095c512c595eac74ee4b440c9d4338c3f
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Sat May 10 08:37:34 2025 -0400

    commit numero 1
```

Centrémonos en las líneas amarillas que empiezan con la palabra 'commit', se los denomina **hash del commit**, es un identificador único que asignamos a cada commit en el repositorio.

Cuando queramos buscar un commit en específico, debemos usar el comando **git show hash\_del\_commit**.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git show 7a26740
commit 7a26740db6c778cbf6703c98c1118b6a15afa904
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Sat May 10 10:55:35 2025 -0400

    commit numero2

diff --git a/com.txt b/com.txt
index 3379fd4..0fc9230 100644
--- a/com.txt
+++ b/com.txt
@@ -1,2 @@
 primera linea de parrafo
+segunda liena de codigo
```

Al hacer git log, el identificador de nuestro segundo commit es: 7a26740db6c778cbf6703c98c1118b6a15afa904, afortunadamente también tenemos el copia y pega, pero podemos usar los primeros caracteres del identificador, lo recomendable son de 6 a 10, en mi caso usamos los primeros 7.

Solo tenemos un archivo en ese commit, y es ese el que nos muestra, si quisiéramos trabajar en un archivo en específico cuando tuviéramos varios usamos el **git show hash\_del\_commit - archivo**

### **git show hash\_del\_commit - - archivo**

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git show 7a26740 -- com.txt
commit 7a26740db6c778cbf6703c98c1118b6a15afa904
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Sat May 10 10:55:35 2025 -0400

    commit numero2

diff --git a/com.txt b/com.txt
index 3379fd4..0fc9230 100644
--- a/com.txt
+++ b/com.txt
@@ -1,2 @@
 primera linea de parrafo
+segunda liena de codigo
```

### **Como ver el print o contenido de un commit en específico**

Usando el comando git show hash\_del\_commit:archivo, podemos acceder al contenido en un determinado commit.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/carpeta (master)
$ git show 7a26740:com.txt
primera linea de parrafo
segunda liena de codigo
```

Acá vemos que en nuestro commit número 2, teníamos un contenido de dos líneas de texto.

### Git show head

Git show head muestra información de nuestro ultimo commit, pero podemos modificarlo para que muestre el penúltimo, antepenúltimo, etc. Con el comando de **git show HEAD~Posicion**

El penúltimo lo llamamos son el comando **git show HEAD~1**

```
$ git show HEAD~1
commit 7a26740db6c778cbf6703c98c1118b6a15afa904
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Sat May 10 10:55:35 2025 -0400

    commit numero2

diff --git a/com.txt b/com.txt
index 3379fd4..0fc9230 100644
--- a/com.txt
+++ b/com.txt
@@ -1,2 @@
 primera linea de parrafo
+segunda liena de codigo
```

Y el penúltimo con el **git show HEAD~2**

```
$ git show HEAD~2
commit a4c1daf095c512c595eac74ee4b440c9d4338c3f
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Sat May 10 08:37:34 2025 -0400

    commit numero 1

diff --git a/com.txt b/com.txt
new file mode 100644
index 0000000..3379fd4
--- /dev/null
+++ b/com.txt
@@ -0,0 +1 @@
+primera linea de parrafo
```

Y así sucesivamente.

## Operaciones con ramas

Como habíamos mencionado, una **rama** más conocida como **branch** es como una **copia del proyecto** donde puedes trabajar sin dañar el original.

### Git branch

El comando `git branch` muestra todas las ramas que existen en nuestro proyecto y nos indica en cuál estamos actualmente mediante un asterisco (\*) al lado del nombre.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ramass (master)
$ git branch
* master
```

Como se observa solo tenemos una rama (`master`) que es nuestra rama principal, ahora vamos a crear otra rama

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ramass (master)
$ git branch rama1

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ramass (master)
$ git branch
* master
  rama1
```

Se creo una rama llamada **rama1**, lo hicimos con el comando **git branch nombre**, y con el comando **git branch** podemos ver la **rama1** creada.

### Checkout

Para movernos entre ramas, es decir cambiar de una rama a otra, usamos el comando **git checkout nombre\_rama** o **git checkout nombre\_hash**

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ramass (master)
$ git checkout rama1
Switched to branch 'rama1'

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ramass (rama1)
$ git branch
  master
* rama1
```

El asterisco se sitúa al lado de rama1, lo que significa que esa es la rama sobre la cual estamos parados.

### Renombrar una rama

Para renombrar una rama usamos de igual manera el comando git branch, seguido de -m, el nombre de tu rama a modificar y el nuevo nombre de esa rama.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ramass (rama1)
$ git branch -m rama1 rama_1.2

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ramass (rama_1.2)
$ git branch
  master
* rama_1.2
```

### Eliminar una rama

Para eliminar una rama usamos el comando git branch -d nombre\_rama

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ramass (master)
$ git branch -d rama_1.2
Deleted branch rama_1.2 (was aa859c7).

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ramass (master)
$ git branch
* master
```

Ojo que nunca puedes eliminar algo sobre lo cual estas parado, para eliminar una rama debes ir a la rama principal o sus predecesores.

### ¿Como funciona una rama en tu proyecto?

Debemos diferenciar el trabajo en una rama secundaria del trabajo en la rama master. Como mencionamos antes, **una rama es una versión alternativa del proyecto que parte de un punto específico del historial**. Cuando creamos una nueva rama a partir de master, todo el contenido existente en master hasta el último commit se copia a la nueva rama. A partir de ahí, cualquier cambio realizado en esa nueva rama no afectará a master, a menos que lo fusionemos explícitamente. Por lo tanto, cuando dejamos esa rama y regresamos a master, los cambios hechos en la rama secundaria ya no serán visibles desde master.

Veamos cómo funciona, presta atención y disculpa lo largo que será esto.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~  
$ cd desktop  
  
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop  
$ mkdir arch_ramas  
  
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop  
$ cd arch_ramas  
  
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas  
$ mkdir rama_pri  
  
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas  
$ cd rama_pri  
  
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri  
$ touch archivo1.txt  
  
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri  
$ echo "55+45=100" > archivo1.txt  
  
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri  
$ cat archivo1.txt  
55+45=100
```

Nos dirigimos a nuestro escritorio, creamos una carpeta llamada **arch\_ramas**, en esa carpeta creamos otra carpeta llamada **rama\_pri**.

En la ultima carpeta que creamos, creamos un archivo **archivo1.txt**, al cual le agregamos un texto de una suma,  $55+45=100$ .

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri  
$ git init  
Initialized empty Git repository in C:/Users/JAVIER/Desktop/arch_rama
```

Al hacer **git init** en nuestra ultima carpeta, esa carpeta se convierte en un repositorio git y crea una carpeta **.git**.

Además de que por defecto nuestra **rama principal** o (master) estará en esa carpeta.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (master)
$ git add .
warning: in the working copy of 'archivo1.txt', LF will be replaced b

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (master)
$ git commit "primer commit"
error: pathspec 'primer commit' did not match any file(s) known to gi

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (master)
$ git commit -m "primer commit"
[master (root-commit) 1d75a7b] primer commit
1 file changed, 1 insertion(+)
create mode 100644 archivo1.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (master)
$ git log
commit 1d75a7bc3d9778323b9b176fec61f404c67d325f (HEAD -> master)
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Mon May 12 12:29:04 2025 -0400

```

Es recomendable realizar un commit antes de crear una nueva rama, ya sea el primero del proyecto o el último de una serie. Esto se debe a que, al crear una rama, Git toma el estado del proyecto según el último commit realizado. De esta manera, todos los archivos confirmados estarán disponibles en la nueva rama y se podrán trabajar sin afectar la rama principal.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (master)
$ git branch
* master

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (master)
$ git branch rama_1

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (master)
$ git branch
* master
  rama_1

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (master)
$ git checkout rama_1
Switched to branch 'rama_1'

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git branch
  master
* rama_1

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ ls
archivo1.txt

```



Como se observa, solo tenemos una rama, nuestra rama **master**, creamos una nueva rama, **rama\_1**, nos movemos a esa rama y hacemos un **ls**, el cual nos dice que tenemos un archivo `archivo1.txt`, que se copio de nuestra rama principal.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ ramas/rama_pri (rama_1)
$ touch nuevo_arch_1.txt nuevo_arch_2.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ ramas/rama_pri (rama_1)
$ ls
archivo1.txt  nuevo_arch_1.txt  nuevo_arch_2.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ ramas/rama_pri (rama_1)
$ echo "55+45=110" > archivo1.txt
```

Vamos a crear dos nuevos archivos de texto en nuestra rama secundaria, hacemos un **ls** y en efecto se crearon. Y además vamos a modificar nuestro `archivo1.txt` con una suma incorrecta.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ ramas/rama_pri (rama_1)
$ cat archivo1.txt
55+45=110

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ ramas/rama_pri (rama_1)
$ git checkout master
Switched to branch 'master'

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ ramas/rama_pri (master)
$ cat archivo1.txt
55+45=100
```

Notemos que podemos verificar que ambos archivos son diferentes en ambas ramas, el error que cometimos en nuestra rama secundaria no esta en la rama (master).

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ ramas/rama_pri (rama_1)
$ git checkout master
M      archivo1.txt
Switched to branch 'master'

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ ramas/rama_pri (master)
$ ls
archivo1.txt  nuevo_arch_1.txt  nuevo_arch_2.txt
```

Si volvemos a la rama principal, notemos una M, significa que el archivo **archivo1.txt** ha sido modificado, pero que aun no ha sido commiteado y que no pertenece a ninguna rama.

Ahora podremos notar un error, hacemos un **ls**, y nos aparecen los archivos de la rama secundaria, **rama\_1**, cosa que no debería pasar, pero la razón por la que sucede esto es que no hemos commiteado esos archivos en nuestra rama secundaria, y por eso nos aparecen en nuestra rama principal.

```
$ git checkout rama_1
M       archivo1.txt
Switched to branch 'rama_1'

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git add .
warning: in the working copy of 'archivo1.txt', LF will be replaced
by CRLF the next time Git touches it

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git commit -m "commit de la rama"

[rama_1 01c237a] commit de la rama
 3 files changed, 1 insertion(+), 1 deletion(-)
 create mode 100644 nuevo_arch_1.txt
 create mode 100644 nuevo_arch_2.txt
```

Volvemos a nuestra rama secundaria, donde aun nos muestra la M, que es ese archivo que modificamos, pero no commiteamos.

Usamos el comando **git add .** para seleccionar y preparar todos los archivos modificados y nuevos del directorio actual, dejándolos listos para el próximo commit en la rama en la que nos encontramos, en nuestro caso, **rama\_1**.

Una vez ya los commiteamos, se nos muestra que un archivo a sido modificado, y dos que han sido creados.

```
$ git checkout master
Switched to branch 'master'

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (master)
$ ls
archivo1.txt
```

Ahora volvemos a nuestra rama principal, y al hacer un **ls** nos muestra que solo tenemos un archivo, y que los otros dos no existen en ella, y que están en una rama secundaria.

```
commit 1d75a7bc3d9778323b9b176fec61f404c67d325f (master)
Author: JavierV <javier.alan.ventura12@gmail.com>
Date:   Mon May 12 12:29:04 2025 -0400

    primer commit

diff --git a/archivo1.txt b/archivo1.txt
new file mode 100644
index 0000000..8f5d829
--- /dev/null
+++ b/archivo1.txt
@@ -0,0 +1 @@
+55+45=100
```

El archivo archivo1.txt no se ve afectado por nada en nuestra rama principal.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git log -p archivo1.txt
commit 01c237aeb21628df1ea27e1c0f74f38126ff6e2d (HEAD -> rama_1)
Author: JavierV <javier.alan.ventura12@gmail.com>
Date:   Mon May 12 12:46:21 2025 -0400

    commit de la rama

diff --git a/archivo1.txt b/archivo1.txt
index 8f5d829..344c236 100644
--- a/archivo1.txt
+++ b/archivo1.txt
@@ -1,1 +1,1 @@
-55+45=100
+55+45=110
```

En cambio, en la rama secundaria, a partir del archivo original de master, eliminamos una línea de texto y agregamos otra.

**Conociendo más a fondo git checkout**

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ cat archivo1.txt
55+45=110

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ echo "1+1=3" >> archivo1.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ cat archivo1.txt
55+45=110
1+1=3

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git checkout archivo1.txt
Updated 1 path from the index

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ cat archivo1.txt
55+45=110
```

Al hacer **git checkout** a un archivo en específico lo que hace es regresar a su commit anterior.

Aunque lo más recomendado es usarlo de la siguiente manera

```
$ git checkout -- archivo1.txt
```

## Reset

Sirve para retroceder cambios, quitar commits, archivos precommitados (git add), o borrar todos tus cambios.

**Git reset - - soft:** Este comando elimina uno o más commits recientes, pero **sin borrar los cambios en tus archivos**.

Los archivos modificados quedan listos para volver a hacer git commit, como si ya hubieras hecho git add.

A continuación, crearemos más commits en nuestra rama secundaria, modificando cambios en cada uno, como podemos ver estamos parados sobre nuestro tercer commit, si queremos eliminar el último commit para volver a commitear nuevamente usamos el comando git reset de la siguiente manera.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git log --oneline
ba2b549 (HEAD -> rama_1) tercer commit de la rama
996569a segundo commit de la rama
01c237a commit de la rama
1d75a7b (master) primer commit

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git reset --soft HEAD~1

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git log --oneline
996569a (HEAD -> rama_1) segundo commit de la rama
01c237a commit de la rama
1d75a7b (master) primer commit
```

Podemos ver que eliminamos nuestro ultimo commit y nos situamos en el penúltimo, HEAD~1.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git commit -m "tercer commit corregido"
[rama_1 74265fa] tercer commit corregido
1 file changed, 2 insertions(+)

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git log --oneline
74265fa (HEAD -> rama_1) tercer commit corregido
996569a segundo commit de la rama
01c237a commit de la rama
1d75a7b (master) primer commit
```

Y podemos volver a commitar sin necesidad de nuevamente hacer **git add**.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git commit -m "tercer commit corregido"
[rama_1 74265fa] tercer commit corregido
1 file changed, 2 insertions(+)

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git log --oneline
74265fa (HEAD -> rama_1) tercer commit corregido
996569a segundo commit de la rama
01c237a commit de la rama
1d75a7b (master) primer commit
```

**Git reset - -mixed:** Lo que hace es borrar algún commit que especifiquemos, y además quita el git add, es decir que también quita su precommit.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git reset --mixed HEAD~1
Unstaged changes after reset:
M   archivo1.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git log --oneline
996569a (HEAD -> rama_1) segundo commit de la rama
01c237a commit de la rama
1d75a7b (master) primer commit

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git add archivo1.txt
warning: in the working copy of 'archivo1.txt', LF will be replaced by CRLF the next time Git touches it

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git commit -m "commit corregido por tercera ves"
[rama_1 7f86ad9] commit corregido por tercera ves
1 file changed, 2 insertions(+)

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git log --oneline
7f86ad9 (HEAD -> rama_1) commit corregido por tercera ves
996569a segundo commit de la rama
01c237a commit de la rama
1d75a7b (master) primer commit

```

Como mencionamos, el comando eliminó nuestro último commit y también lo que estaba preparado para commit (precommit).  
Vemos nuevamente la letra **M**, que indica que el archivo ha sido **modificado** con respecto al último commit.

Esto significa que es diferente al commit anterior y que **no está en el área de preparación** (es decir, no ha pasado por git add).

**Git reset - -hard:** elimina commits, precommmits y contenido, es el mas fuerte de los comandos ya que borras absolutamente todo.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git log --oneline
7f86ad9 (HEAD -> rama_1) commit corregido por tercera ves
996569a segundo commit de la rama
01c237a commit de la rama
1d75a7b (master) primer commit

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git reset --hard 01c237a
HEAD is now at 01c237a commit de la rama

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git log --oneline
01c237a (HEAD -> rama_1) commit de la rama
1d75a7b (master) primer commit

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ cat archivo1.txt
55+45=110

```

Hemos eliminado todos los commits posteriores y regresado al primer commit en la rama `rama_1`, restaurando su contenido original y borrando los cambios que habíamos hecho después.

## Alias en git

Nos sirve para abreviar muchos de los comandos que escribimos en la terminal.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git config --global alias.lg "log --oneline --graph --decorate --all"

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ lg
bash: lg: command not found

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git lg
* 01c237a (HEAD -> rama_1) commit de la rama
* 1d75a7b (master) primer commit
```

La manera de configurar uno es **git config --global alias.(alias\_abrev)**  
**comando**

Podremos configurar los que necesitemos o los que nos cuesten memorizar.

## Git ignore

Al momento de hacer commits, puede haber cierta información que no deseamos subir, ya sea información personal, contraseñas, o archivos que sean irrelevantes para nuestro proyecto, para eso usamos git ignore, para ignorar esos archivos.

El funcionamiento es simple, se crea un archivo **.gitignore** en la raíz, y en ella podremos agregar y modificar todos los archivos que queremos que git pase de ellos.

Podemos crearlo con el siguiente comando:

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ touch .gitignore
```

Para agregar archivos o carpetas que deben ser ignorados, lo hacemos editando el archivo **.gitignore** que acabamos de crear. Para acceder al editor de texto y modificar este archivo, utilizamos el comando **nano**.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ nano .gitignore
```

Una vez escrito ese comando se nos abrirá el editor, en el cual agregamos el nombre de los archivos, carpetas u extensiones de todo lo que se necesite ignorar en nuestro proyecto.

```
GNU nano 8.3 .gitignore
archivo1.txt
```

Para guardar cambios y salir del editor, presionamos Ctrl + o, Enter y Ctrl + x.

Para verificar los archivos que están siendo ignorados correctamente ejecutamos el comando git status - ignored.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git status --ignored
On branch rama_1
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore

nothing added to commit but untracked files present (use "git add" to track)
```

Epaaaaa..... hicimos los pasos para ignorar un archivo, pero en la terminal nos indica que no hay archivos ignorados. Esto se debe a que el archivo que intentamos ignorar ya fue **commiteado previamente**, lo que significa que Git ya está **rastreando** ese archivo, por lo tanto, no será ignorado automáticamente. Para que Git lo ignore, necesitamos eliminarlo del control de versiones con **git rm -cached nombre\_archivo**.

Lo que hace este ultimo comando, es que elimina del archivo los commits, los git add, y git ya no lo rastreara para futuros commits.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git rm --cached archivo1.txt
rm 'archivo1.txt'
```



```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git add .
warning: in the working copy of '.gitignore', LF will be replaced by CRLF

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git commit -m "commit sin archivo1.txt"
[rama_1 0831b10] commit sin archivo1.txt
2 files changed, 1 insertion(+), 1 deletion(-)
create mode 100644 .gitignore
delete mode 100644 archivo1.txt
```

Volvemos a usar **git add .** y **git commit**. Esta vez, `archivo1.txt` ya no será incluido en futuros commits. La salida de la terminal nos indica que se ha creado el archivo **.gitignore** y que **archivo1.txt** ha sido eliminado del control de versiones de Git.

Además, al hacer `git add .`, también se incluyen en el *stage* los nuevos archivos rastreables, como `.gitignore`. Luego, al hacer `git commit`, se guarda ese archivo en el historial del repositorio.

Es importante destacar que **solo se commitea el archivo .gitignore, no los archivos listados dentro de él**. Sin este commit, `.gitignore` no tendría ningún efecto, ya que Git necesita conocer y rastrear ese archivo para poder aplicar correctamente las reglas de exclusión que contiene.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/arch_ramas/rama_pri (rama_1)
$ git status --ignored
On branch rama_1
Ignored files:
  (use "git add -f <file>..." to include in what will be committed)
    archivo1.txt

nothing to commit, working tree clean
```

Y lo comprobamos mediante el comando que ya usamos, y en este se incluye por fin ese archivo.

### Usar git ignore con nuestros archivos y carpetas de manera correcta

Además de usar archivos sueltos como lo hicimos, podemos hacer muchas cosas interesantes en nuestro proyecto.

No lo hare en terminal por flojera, pero veamos algunos casos.

```
MINGW64:/c/Users/JAVIER/desktop/arch_ramas/rama_pri
GNU nano 8.3 .gitignore Modified
*.txt
Ignora todos los archivos que terminan
con la extension .txt, podemos replicar_
lo con varios lenguajes de programacion
y archivos con extensiones como css, js
```

```
MINGW64:/c/Users/JAVIER/desktop/arch_ramas/rama_pri
GNU nano 8.3 .gitignore Modified
nombre_carpeta/
Este instruccion ignora carpetas
enteras, debemos poner la / al final
de cada carpeta que le indiquemos
```

```
MINGW64:/c/Users/JAVIER/desktop/arch_ramas/rama_pri
GNU nano 8.3 .gitignore Modified
/nombre_archivo.txt
indicamos que el archivo txt unicamente
se elimina de la raiz de nuestro
proyecto, mas no de subcarpetas que
puedan contener el mismo nombre y
extension
```

```
MINGW64:/c/Users/JAVIER/desktop/arch_ramas/rama_pri
GNU nano 8.3 .gitignore Modified
/subcarpeta/*.txt
eliminamos todos los archivos con
extensiones .txt dentro de una sub_
carpeta, solo afecta a la ruta final de
esa subcarpeta
```

```
MINGW64:/c/Users/JAVIER/desktop/arch_ramas/rama_pri
GNU nano 8.3 .gitignore Modified
**/archivo1.txt o **/carpeta
Ignora ya sea un archivo o una carpeta
en cualquier parte del proyecto, es
decir en cualquier carpeta o subcarpeta
del proyecto
```

```
MINGW64:/c/Users/JAVIER/desktop/arch_ramas/rama_pri
GNU nano 8.3 .gitignore Modified
carpeta_10/**/*.*.txt
dentro de la carpeta_10 elimina todos
los archivos con extension .txt
```

## Git tag

Nos sirve para resaltar commits importantes de nuestro proyecto, como tal son etiquetas de marcaje.

Crear un tag:

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas (master)
$ git tag version1.0

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas (master)
$ git tag
version1.0
```

La manera en que creamos un tag es mediante el comando **git tag nombre\_tag**, lo que hace es crearnos una etiqueta en nuestro ultimo commit.

Verificamos con **git tag** que se haya creado nuestra etiqueta.

Crear un tag anotada o con un mensaje:

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas (master)
$ git tag -a version2.0 -m "version mas estable"
```

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas (master)
$ git tag
version1.0
version2.0

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas (master)
$ git show version2.0
tag version2.0
Tagger: JavierV <javier.alan.ventura12@gmail.com>
Date: Tue May 13 16:15:40 2025 -0400

version mas estable

commit 723e4274581af620d98ca884b32f719f891b6c75 (HEAD -> master, tag: version2.0, tag: version1.0)
Author: JavierV <javier.alan.ventura12@gmail.com>
Date: Tue May 13 15:58:54 2025 -0400

    commit 1

diff --git a/ejemplo.txt b/ejemplo.txt
new file mode 100644
index 0000000..020eb14
--- /dev/null
+++ b/ejemplo.txt
@@ -0,0 +1 @@
+ejemplo1
```

Hemos creado una **etiqueta anotada** (version2.0) que apunta a un commit en la rama master, y ese mismo commit también tiene otra etiqueta llamada version1.0. La etiqueta contiene un mensaje personalizado: "versión más estable".

Crear un tag en un commit en específico:

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas (master)
$ git tag -a version3.0 8760ac1 -m "crear un tag en commit especifico"
```

Eliminar etiquetas:

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/ejer_1.txt (master)
$ git tag -d v3.0
Deleted tag 'v3.0' (was 959c441)
```

## Git merge

Hasta ahora hemos visto cómo crear ramas y trabajar de forma independiente en cada una. Sin embargo, llegará un momento en que necesitaremos fusionarlas. Es decir, combinar los cambios de dos ramas en una sola, generando un nuevo commit que represente esa unión.

Una regla importante al usar el comando git merge es recordar que debes posicionarte en la rama donde quieres que se apliquen los cambios. Por ejemplo, si deseas fusionar la rama secundaria **rama\_sec** con la rama principal master, primero debes situarte en master y luego ejecutar git merge **rama\_sec**,

esto traerá los cambios realizados en **rama\_sec** hacia master, fusionando el trabajo de ambas ramas dentro de master.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas (master)
$ mkdir fusion

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas (master)
$ cd fusion

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (master)
$ touch ejercicio1.py

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (master)
$ nano ejercicio1.py
```

```
GNU nano 8.3 ejercicio1.py
def mensaje():
    return "Hola desde la rama principal (main)"

print(mensaje())
```

Ctrl + o, enter y ctrl + x

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (master)
$ python ejercicio1.py
Hola desde la rama principal (main).
```

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (master)
$ git init
Initialized empty Git repository in C:/Users/JAVIER/Desktop/etiquetas/fusion/.git/

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (master)
$ git add ejercicio1.py
warning: in the working copy of 'ejercicio1.py', LF will be replaced by CRLF the next time
it

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (master)
$ git commit -m "commit de master"
[master (root-commit) 717e27a] commit de master
1 file changed, 4 insertions(+)
create mode 100644 ejercicio1.py
```

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (master)
$ git branch rama_sec

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (master)
$ git checkout rama_sec
Switched to branch 'rama_sec'

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (rama_sec)
$ git branch
  master
* rama_sec

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (rama_sec)
$ ls
ejercicio1.py

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (rama_sec)
$ nano ejercicio1.py

```

```

GNU nano 8.3 ejercicio1.py
def mensaje():
    return "Hola desde la rama_sec"

def nueva_funcion():
    return "Aca podemos agregar cambios y nuevas funcionalidades"

print(mensaje())
print(nueva_funcion())

```

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (rama_sec)
$ python ejercicio1.py
Hola desde la rama_sec
Aca podemos agregar cambios y nuevas funcionalidades

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (rama_sec)
$ git add ejercicio1.py
warning: in the working copy of 'ejercicio1.py', LF will be replaced by CRLF
it

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (rama_sec)
$ git commit -m "commit de la rama_sec"
[rama_sec f3d0eec] commit de la rama_sec
1 file changed, 5 insertions(+), 1 deletion(-)

```

Hasta acá supongo que ya entendimos maso menos lo que hice, ahora fusionaremos mi **rama\_sec** a mi rama principal **master**.

Recuerda que la fusión se realiza desde la rama en la que queremos aplicar los cambios. Al hacerlo, los cambios de ambas ramas se combinan y se guardan en la rama en la que nos encontramos. Por eso nos vamos a nuestra rama **master** para hacer el git **merge**.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (rama_sec)
$ git checkout master
Switched to branch 'master'

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (master)
$ git merge rama_sec
Updating 717e27a..f3d0eec
Fast-forward
 ejercicio1.py | 6 +++++-
 1 file changed, 5 insertions(+), 1 deletion(-)
```

**Updating** nos hace referencia a la fusión de ambas ramas

**Fast-forward** nos dice que ha sido una fusión sin conflictos

Hemos eliminado nuestro código y agregado 5 nuevas, recordemos que teníamos 3 líneas de código, pero solo se toma como uno, y como agregamos 6, menos “1”, que se eliminó, entonces tenemos 5+ y 1-.

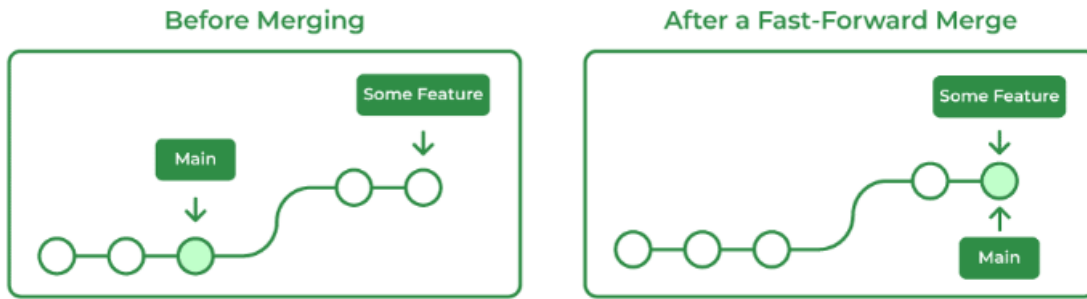
```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/etiquetas/fusion (master)
$ git diff 717e27a f3d0eec
diff --git a/ejercicio1.py b/ejercicio1.py
index 66e0a1e..13b03a1 100644
--- a/ejercicio1.py
+++ b/ejercicio1.py
@@ -1,4 +1,8 @@
 def mensaje():
-     return "Hola desde la rama principal (main)."
+     return "Hola desde la rama_sec"
+
+def nueva_funcion():
+     return "Aca podemos agregar cambios y nuevas funcionalidades"
+
 print(mensaje())
+print(nueva_funcion())
```

Y algo importante es que esta fusión ya está commiteado en nuestro master, y no hace falta commitarla, pero en caso de conflictos u otras situaciones se deberán hacer commits, pero como tenemos un **fast-forward** todo bien.

Las ramas permanecen intactas luego de la fusión, y podemos volver a trabajar con ellas.

## Tipos de fusión

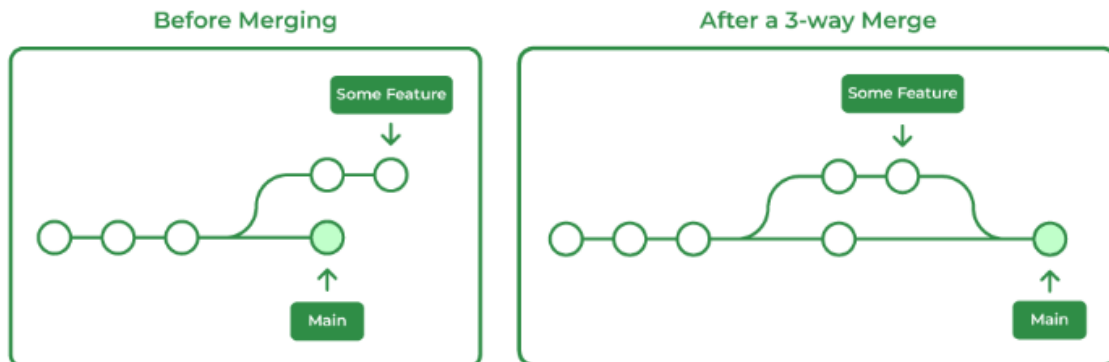
### Fast forward:



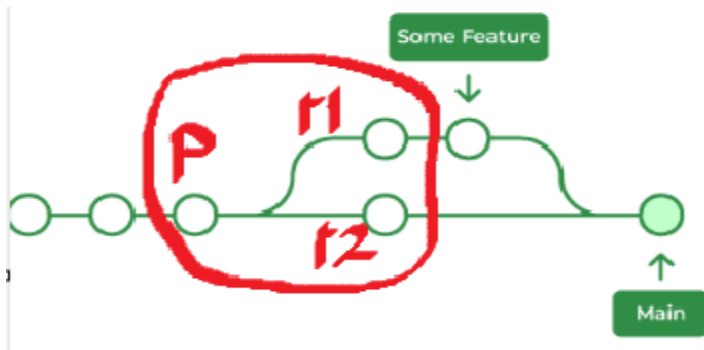
En la imagen de la izquierda, observamos la rama principal main (nuestro master). A partir de esta rama se creó una rama secundaria. Al ejecutar git merge, ambas ramas se unieron y se generó un nuevo commit de combinación (merge commit) que integró los cambios realizados en la rama secundaria dentro de main. En la imagen vemos como es una ruta **lineal** entre las dos ramas que se van a fusionar. Pero como tal no es una fusión si no que mueve a **master** a la rama secundaria.

Este es el ejemplo que realizamos en el último ejercicio. El merge se realizó automáticamente mediante un *fast-forward*, lo que significa que no hubo conflictos ni necesidad de crear un commit adicional, ya que la rama **master** no había recibido ningún cambio desde que se creó la rama secundaria. Si hubiéramos hecho modificaciones en **master** antes de fusionar, Git no habría podido hacer un *fast-forward* y habría sido necesario generar un commit de fusión para integrar los cambios de ambas ramas.

### Three-way marge:



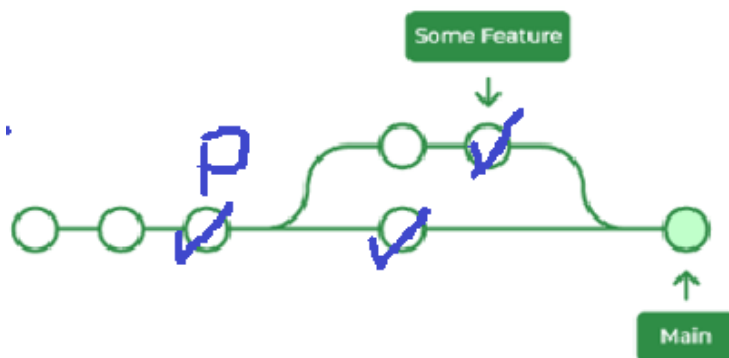




Este tipo de combinación se denomina *fusión tripartita* (three-way merge). Para entenderlo, imaginemos que P es el último commit de la rama **master** (nuestra rama principal). A partir de ese punto, creamos una rama secundaria llamada some feature (r1), en la cual realizamos dos nuevos commits. Hasta aquí, el escenario es similar al ejemplo anterior.

Sin embargo, si luego regresamos a la rama **master**, realizamos algunos cambios y los confirmamos con un nuevo commit, se genera otra línea de desarrollo, que podemos llamar r2. En este punto, existen dos ramas que han evolucionado de forma independiente desde un ancestro común (P): una corresponde a los cambios hechos en some feature y la otra a los cambios hechos directamente en master. Como resultado, la historia del proyecto deja de ser lineal y se ramifica.

Al hacer un git merge sucede lo siguiente:



La fusión se hace de 3 participantes, el ultimo commit de master, ultimo commit de la rama secundaria, y de su ancestro común, en este caso P, que es el punto de donde nacen ambas líneas r1 y r2.

Este commit P es el punto de partida desde el cual ambas ramas comenzaron a desarrollarse de forma independiente. Al realizar un git merge, git compara los cambios realizados en ambas ramas a partir de ese ancestro común (P) y trata de integrarlos en un nuevo commit de combinación, porque git analiza tres

versiones del proyecto para determinar cómo unirlos: el estado original (P), los cambios en master y los cambios en la rama secundaria.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ mkdir fusion_trip

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cd fusion_trip
bash: cd: too many arguments

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cd fusion_trip

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip
$ git init
Initialized empty Git repository in C:/Users/JAVIER/Desktop/fusion_trip/.git/

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master)
$ touch texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master)
$ echo "123456789" > texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master)
$ git add texto.txt
warning: in the working copy of 'texto.txt', LF will be replaced by CRLF the next time Git

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master)
$ git commit -m "commit P"
[master (root-commit) bf385bc] commit P
1 file changed, 1 insertion(+)
create mode 100644 texto.txt
```

Creamos un directorio, en el directorio iniciamos git, luego de eso creamos un texto.txt, en el cual ponemos “123456789”, ponemos en stage y commiteamos, ahora esta es nuestra rama master padre(P).

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master)
$ git branch some_feature

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master)
$ git branch
* master
  some_feature

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master)
$ git checkout some_feature
Switched to branch 'some_feature'

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ echo "11111111" >> texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ cat texto.txt
123456789
111111111

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git add texto.txt
warning: in the working copy of 'texto.txt', LF will be replaced by CRLF

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git commit -m "commit1 en R1-somefeature"
[some_feature 6d23d18] commit1 en R1-somefeature
1 file changed, 1 insertion(+)

```

Creamos una nueva rama, llamada **some\_feature**, nos movemos a esa rama, y agregamos texto “11111111”, hacemos un stage y commit.

Esta es nuestra línea de trabajo R1 en nuestra rama secundaria.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ echo "22222222" >> texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git add texto.txt
warning: in the working copy of 'texto.txt', LF will be replaced by CRLF

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git commit -m "commit2 en R1-somefeature"
[some_feature 7d480d2] commit2 en R1-somefeature
1 file changed, 1 insertion(+)

```

Creamos un segundo commit, en la línea **R1** que es nuestra rama secundaria **some\_feature**, agregamos más texto “22222222”, hacemos otro stage y commit.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git checkout master
Switched to branch 'master'

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master)
$ echo "333333333" >> texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master)
$ git add texto.txt
warning: in the working copy of 'texto.txt', LF will be replaced by CRLF

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master)
$ git commit -m "commit1 en master"
[master b3ad7df] commit1 en master
1 file changed, 1 insertion(+)
```

Volvemos a nuestra rama **master**, y en esa rama creamos otro commit, agregando texto “333333333”, hacemos stage y un commit, ahora a partir de **P** se han creado dos ramas, una en **main r2** y otra en **some\_feature r1**.

Si hacemos un **merge** usara a nuestro ancestro **P**, y sus ramas descendientes.

Resaltemos que el **merge** lo hacemos desde master, donde queremos que los cambios se conserven.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master)
$ git merge some_feature
Auto-merging texto.txt
CONFLICT (content): Merge conflict in texto.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Un conflicto, no te preocupes ahora aprendemos lo que son.

## Conflictos

Ocurren cuando se intentan fusionar cambios de dos ramas diferentes, pero estos cambios se realizan en las mismas líneas de código o en archivos que han sido modificados de forma contradictoria. En estos casos, el sistema no puede determinar automáticamente qué cambios deben prevalecer y requiere la intervención manual del usuario para resolver la situación.

Es decir que en nuestros dos archivos hay líneas diferentes de texto en una misma línea, y git nos dice que no sabe que hacer, así que te pide que lo hagas tú, total es tu trabajo y un algoritmo no puede decidir por ti xd.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master|MERGING)
$ nano texto.txt
```

Entramos al archivo mediante nano.

```
GNU nano 8.3          texto.txt
123456789
<<<<<<< HEAD
333333333
=====
111111111
222222222
>>>>>> some_feature
```

Aquí nos dice que HEAD, es decir, la línea "**123456789**", es igual en ambos archivos, por lo que no hay ningún problema en esa parte. Sin embargo, el conflicto se presenta en las líneas "**333333333**" y "**111111111**" / "**222222222**", que ocupan el mismo espacio en el archivo. Git no sabe si debe:

- **Mantener ambas líneas de texto (de las dos ramas),**
- **Eliminar la línea de texto que está en el commit de master, o**

Por lo tanto, Git nos deja la decisión de elegir cuál de las opciones aplicar. Es ahí donde entra nuestra intervención para resolver el conflicto.

Para solucionarlo debemos tomar una de 3 posibles soluciones que te menciones, o modificar todo, pero eso seria haber hecho todo por nada jaja, buen en mi caso conservare lo de ambas ramas, y como lo hago, pues en el mismo nano elimino esos caracteres de igualdad y los nombres de las ramas, y guardo.

```
GNU nano 8.3          texto.txt
123456789

111111111
222222222

333333333
```

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master|MERGING)
$ git add texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master|MERGING)
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
  modified:   texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master|MERGING)
$ git commit -m "commit merge final"
[master 6490e52] commit merge final

```

Notemos que estamos haciendo un merge por las letras azules en el lado derecho, **merging**, y volvemos a hacer un stage y commit de los cambios.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master)
$ git lg
*   6490e52 (HEAD -> master) commit merge final
| \
|  * 7d480d2 (some_feature) commit2 en R1-somefeature
|  * 6d23d18 commit1 en R1-somefeature
* | b3ad7df commit1 en master
| /
* bf385bc commit P

```

Veamos las ramas mediante esas líneas, fusionamos dos líneas de trabajo en una, y tenemos el commit merge final.

Otras dos posibilidades de solucionar conflictos son:

- `git merge --theirs <archivo>`: Esta opción conserva los cambios de la *rama* que se está fusionando (rama a combinar), y descarta los cambios de la *rama local* (rama actual). Es decir, toma *su versión* (theirs) en lugar de la *nuestra* (mine o ours).
- `git merge --mine` (o `--ours`) `<archivo>`: Esta opción conserva los cambios de la *rama local* (rama actual), y descarta los cambios de la *rama* que se está fusionando (rama a combinar). Es decir, toma *nuestra versión* (mine o ours) en lugar de *su versión* (theirs).

## Git stash

Nos permite guardar temporalmente todos los cambios que hemos hecho en nuestra rama actual, sin necesidad de hacer un commit. Es útil cuando queremos:

- Cambiar de rama sin perder nuestro progreso.
- Atender tareas urgentes en otra rama.
- Detenernos sin dejar commits incompletos o desordenados.

De esta forma, podemos pausar nuestro trabajo, enfocarnos en otra cosa y luego retomar exactamente donde lo dejamos. O también es posible llevar un stash a otra rama y fusionarla allí, sin necesidad de un commit.

Ahora aprendamos dos conceptos importantes:

**Archivos tracked:** Son archivos a los que git ya les hace un seguimiento ya sea mediante stage o un commit.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git status
On branch some_feature
nothing to commit, working tree clean
```

Acá nos dice que todos los archivos están siendo rastreados.

**Archivos untracked:** No tienen seguimiento de git, no tienen un stage ni un commit.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ touch stash.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ ls
stash.txt  texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ echo "archivo untracked aqui" > stash.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git status
On branch some_feature
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    stash.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Acá tenemos un archivo untracked, ya que recién lo creamos y esta pendiente a que hacer con él. No podemos dejarlo allí y cambiar de rama.

Una vez aclarado esto continuemos con git stash:

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash
No local changes to save
```

WTF, y ahora que paso? Nos dice que no hay cambios que hacer a pesar de crear un nuevo archivo, al ser un comando de git, necesariamente debe estar rastreado por git, al hacer un git stash a un archivo que no esta siendo seguido por git pues pasa esto, pero como todo en la vida tiene solución, haz esto:

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash -u
warning: in the working copy of 'stash.txt', LF will be replaced by CRLF the next time Git touches it
Saved working directory and index state WIP on some_feature: 7d480d2 commit2 en R1-somefeature
```

Se uso el comando **git stash** seguido de **-u**, untracked, lo que hace es guardar temporalmente los archivos que git no les hace un seguimiento

**Git stash list:** nos sirve para ver una lista de stash que tenemos almacenados, el 0 es el mas reciente siempre es el mas reciente o tu ultimo stash



```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash list
stash@{0}: WIP on some_feature: 7d480d2 commit2 en R1-somefeature
```

## Entendamos mejor stash con un ejercicio

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git add stash.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git commit -m "commit de stash"
[some_feature 901a66e] commit de stash
1 file changed, 1 insertion(+)
create mode 100644 stash.txt
```

Normalmente en todos los archivos que trabajemos van a ser de tipo **tracked**, así que vamos a hacer un stage y commit a nuestro archivo.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git status
On branch some_feature
nothing to commit, working tree clean
```

Git status nos dice que no hay nada pendiente que resolver, que no tenemos archivos no modificaciones para guardar, ya que hicimos un commit que guardo todo, básicamente git nos dice que todo esta bien y que podemos apagar nuestra computadora o podemos ir a otro lugar (podemos ir a otra rama), es como si nos dijera, no tienes ningún trabajo pendiente, así que te puedes largar.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ echo "11111111" > stash.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git status
On branch some_feature
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   stash.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Pero imagina que haces un cambio, y no lo guardas ni en stage ni en un commit, git status nos dice que tenemos un archivo que ha sido modificado que está pendiente de que le hagamos un stage y/o un commit, pero nos vale verga lo que nos diga y queremos ir a otra rama.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git checkout master
error: Your local changes to the following files would be overwritten by checkout:
    stash.txt
Please commit your changes or stash them before you switch branches.
Aborting
```

Pues no nos dan permiso, tienes que resolver ese pendiente, ya sea por un commit o un stash.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash save "stash en some_feature/stash.txt"
warning: in the working copy of 'stash.txt', LF will be replaced by CRLF the next time Git
saved working directory and index state On some_feature: stash en some_feature/stash.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash list
stash@{0}: On some_feature: stash en some_feature/stash.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git checkout master
Switched to branch 'master'

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master)
```

Creamos un stash con **git stash save “mensaje”**, y podemos ver con **git stash list** que se creo nuestro stash, asi que nuevamente le pedimos permiso a nuestro jefe git para ir a otra rama, y nos concede ese permiso ya que no tenemos trabajo pendiente.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (master)
$ git checkout some_feature
Switched to branch 'some_feature'

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ ls
stash.txt  texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git status
On branch some_feature
nothing to commit, working tree clean
```

Volvemos a nuestra rama secundaria.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ echo "22222222" > stash.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash save "stash de los 2s"
warning: in the working copy of 'stash.txt', LF will be replaced by CRLF
Saved working directory and index state On some_feature: stash de los 2s

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ echo "33333333" > stash.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash save "stash de los 3s"
warning: in the working copy of 'stash.txt', LF will be replaced by CRLF
Saved working directory and index state On some_feature: stash de los 3s

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ echo "44444444" > stash.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash save "stash de los 4s"
warning: in the working copy of 'stash.txt', LF will be replaced by CRLF
Saved working directory and index state On some_feature: stash de los 4s

```

Y creamos 3 nuevos stash, modificando 3 veces nuestro archivo stash.txt.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash list
stash@{0}: On some_feature: stash de los 4s
stash@{1}: On some_feature: stash de los 3s
stash@{2}: On some_feature: stash de los 2s
stash@{3}: On some_feature: stash en some_feature/stash.txt

```

Podremos notar que stash@{0} siempre será nuestro stash más reciente, que no es algo fijo esos números, y que se van modificando, así que ojo con eso.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash show -p 3
diff --git a/stash.txt b/stash.txt
index d44985a..bb81b3c 100644
--- a/stash.txt
+++ b/stash.txt
@@ -1,1 @@
-archivo untracked aqui
+111111111

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash show -p stash@{3}
diff --git a/stash.txt b/stash.txt
index d44985a..bb81b3c 100644
--- a/stash.txt
+++ b/stash.txt
@@ -1,1 @@
-archivo untracked aqui
+111111111

```

Podemos ver el contenido de cada stash con los dos comandos que acabamos de hacer.

**Regresar a nuestros stash:** Para mover, aplicar y continuar trabajando en donde dejamos algún archivo o proyecto, tenemos dos comandos, **git apply** y **git pop**.

**Git stash apply:** como habíamos visto usando el comando **git stash list** podemos ver el historial de los stash que tenemos, si queremos conservar ese historial y no borrar el stash al que estamos yendo, usamos **apply**

```
$ git stash list
stash@{0}: On some_feature: stash de los 4s
stash@{1}: On some_feature: stash de los 3s
stash@{2}: On some_feature: stash de los 2s
stash@{3}: On some_feature: stash en some_feature/stash.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git apply 2
error: can't open patch '2': No such file or directory

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash apply 2
On branch some_feature
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   stash.txt

no changes added to commit (use "git add" and/or "git commit -a")

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ cat stash.txt
222222222

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash list
stash@{0}: On some_feature: stash de los 4s
stash@{1}: On some_feature: stash de los 3s
stash@{2}: On some_feature: stash de los 2s
stash@{3}: On some_feature: stash en some_feature/stash.txt
```

En primera instancia podemos ver nuestro historial de stash, ahora en nuestro segundo comando ejecutamos el comando **git stash apply 2**, el cual restaura el archivo y su estado y contenido en el momento de hacerle un stash, como podemos observar en el comando **cat**, y si nuevamente vemos nuestro historial aún conserva ese stash.

**Git stash pop:** Es como apply, restaura nuestro stash, pero con la diferencia de que **elimina** al stash y, por ende, lo saca del historial.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash pop stash@{3}
error: Your local changes to the following files would be overwritten by merge:
    stash.txt
Please commit your changes or stash them before you merge.
Aborting
On branch some_feature
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   stash.txt

no changes added to commit (use "git add" and/or "git commit -a")
The stash entry is kept in case you need it again.
```

Antes de usar pop, veamos este problema, ojo que es algo que también me esta costando entender, así que busca una solución tu también y no te fíes de mí.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ echo "555555555" > stash.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash save "'stash de los 5s'"
warning: in the working copy of 'stash.txt', LF will be replaced by CRLF
Saved working directory and index state On some_feature: stash de los 5s

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ cat stash.txt
archivo untracked aqui
```

Agregamos un nuevo stash y comprobamos que el archivo stash.txt no se modifica, ya que los *stash* solo guardan temporalmente los cambios sin aplicarlos directamente.

Stash no cambia ni modifica nuestro directorio actual.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash list
stash@{0}: On some_feature: stash de los 5s
stash@{1}: On some_feature: stash de los 4s
stash@{2}: On some_feature: stash de los 3s
stash@{3}: On some_feature: stash de los 2s
stash@{4}: On some_feature: stash en some_feature/stash.txt
```

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash apply stash@{3}
On branch some_feature
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   stash.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git diff
diff --git a/stash.txt b/stash.txt
index d44985a..79a79f1 100644
--- a/stash.txt
+++ b/stash.txt
@@ -1,1 @@
-archivo untracked aqui
+222222222
```

Atento, en nuestro directorio y nuestro archivo `stash.txt` estaba sin modificaciones con el texto “archivo untracked aqui”, pero una vez que aplicamos **git stash apply stash@{3}** (stash de los 2), modifica nuestro archivo, eliminando la primera línea de texto “archivo untracked aqui”, y agregando la línea de texto “222222222”, se ha hecho una **modificación** de nuestro archivo **stash.txt** con el cual estamos trabajando, el cual necesita ser commiteado o stasheado.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash apply stash@{0}
error: Your local changes to the following files would be overwritten by merge:
      stash.txt
Please commit your changes or stash them before you merge.
Aborting
On branch some_feature
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
      modified:   stash.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Si intentamos agregar un stash adicional, nos saldría un mensaje de commitea o stashea antes de hacer un cambio como merge o stash, nos dice que tenemos pendientes, y mientras no resolvamos ese pendiente, no podremos hacer cambios ya sea con stash o merge.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash push -m "stash modificado"
Saved working directory and index state On some_feature: stash modificado
```

Creamos un nuevo stash, `git stash push -m ""`, es lo mismo que `git stash save ""`

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ cat stash.txt
archivo untracked aqui

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git status
On branch some_feature
nothing to commit, working tree clean
```

Como podemos ver los nuevos stash no modifican nada de nuestro stash.txt, y al hacer un nuevo stash no tenemos pendientes.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash apply stash@{1}
On branch some_feature
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   stash.txt

no changes added to commit (use "git add" and/or "git commit -a")

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git diff
diff --git a/stash.txt b/stash.txt
index d44985a..da65642 100644
--- a/stash.txt
+++ b/stash.txt
@@ -1,1 @@
-archivo untracked aqui
+555555555
```

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash list
stash@{0}: On some_feature: stash modificado
stash@{1}: On some_feature: stash de los 5s
stash@{2}: On some_feature: stash de los 4s
stash@{3}: On some_feature: stash de los 3s
stash@{4}: On some_feature: stash de los 2s
stash@{5}: On some_feature: stash en some_feature/stash.txt
```

Entonces podemos ya hacer un nuevo stash, en nuestro caso git **stash apply stash@{1}** (stash de los 5s), **y modifica nuestro archivo actual**, el cual nuevamente esta pendiente ya sea de un commit o stage, volviendo al mismo problema de antes.

Resaltar esto, cuando en nuestro directorio actual traemos un stash ya sea por medio de apply o pop, **nuestro directorio actual se modifica con el nuevo stash**, pero si **creamos un nuevo stash**, ese archivo modificado se va al nuevo stash, y nuestro directorio actual vuelve a su estado antes de stash. es decir, al último commit.

Si no deseamos crear otro stash podemos hacer un restore, que elimina el ultimo stash que hicimos y lo devuelve a su estado original o ultimo commit.



```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git status
On branch some_feature
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   stash.txt

no changes added to commit (use "git add" and/or "git commit -a")

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ cat stash.txt
5555555555

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git restore stash.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ cat stash.txt
archivo untracked aqui

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git status
On branch some_feature
nothing to commit, working tree clean

```

Vemos que con el stash que hicimos nuestro archivo se modificó, "555555555", con el comando **restore** lo restauramos al ultimo commit, es decir al texto "archivo untracked aqui", y nos quitamos los pendientes para seguir trabajando.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash apply stash@{1}
On branch some_feature
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   stash.txt

no changes added to commit (use "git add" and/or "git commit -a")

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ cat stash.txt
5555555555

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash push -m "stash modificado2"
Saved working directory and index state On some_feature: stash modificado2

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ cat stash.txt
archivo untracked aqui

```

Podemos confirmar lo que dijimos antes, si creamos un nuevo stash todos nuestros cambios irán allí, y nuestro directorio actual vuelve a su estado original, a su último commit.



```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash list
stash@{0}: On some_feature: stash modificado2
stash@{1}: On some_feature: stash modificado
stash@{2}: On some_feature: stash de los 5s
stash@{3}: On some_feature: stash de los 4s
stash@{4}: On some_feature: stash de los 3s
stash@{5}: On some_feature: stash de los 2s
stash@{6}: On some_feature: stash en some_feature/stash.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash pop stash@{5}
On branch some_feature
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   stash.txt

no changes added to commit (use "git add" and/or "git commit -a")
Dropped stash@{5} (3ee3bfb80279c5c9d3bb51355517f71b58aa975a)

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash list
stash@{0}: On some_feature: stash modificado2
stash@{1}: On some_feature: stash modificado
stash@{2}: On some_feature: stash de los 5s
stash@{3}: On some_feature: stash de los 4s
stash@{4}: On some_feature: stash de los 3s
stash@{5}: On some_feature: stash en some_feature/stash.txt

```

El uso de pop, en este ejemplo usamos **pop stash@{5}** (stash de los 2s), elimina el stash de los 2s, y ya no aparece en la lista.

### Eliminar un stash:

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash list
stash@{0}: On some_feature: stash modificado2
stash@{1}: On some_feature: stash modificado
stash@{2}: On some_feature: stash de los 5s
stash@{3}: On some_feature: stash de los 4s
stash@{4}: On some_feature: stash de los 3s
stash@{5}: On some_feature: stash en some_feature/stash.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash drop stash@{0}
Dropped stash@{0} (18071ce998855ddbdb609fae4ad546099a238a0)

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash list
stash@{0}: On some_feature: stash modificado
stash@{1}: On some_feature: stash de los 5s
stash@{2}: On some_feature: stash de los 4s
stash@{3}: On some_feature: stash de los 3s
stash@{4}: On some_feature: stash en some_feature/stash.txt

```

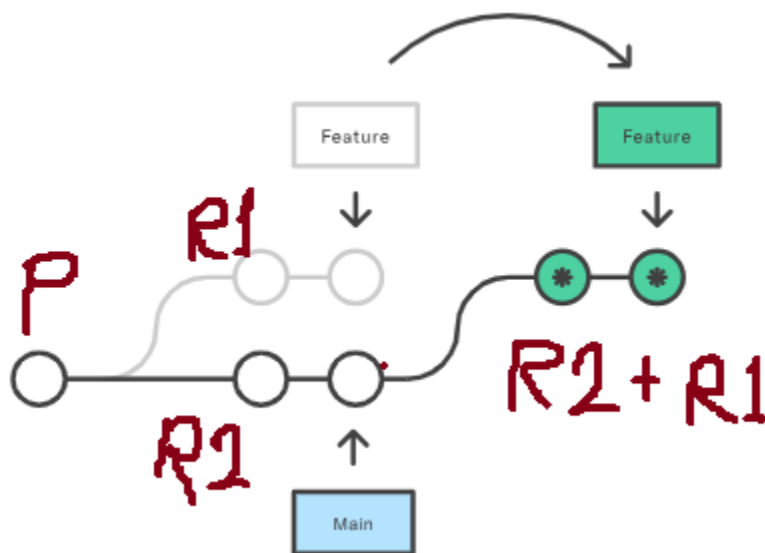
Usando el comando **drop** como lo hicimos podemos eliminar stashes, en nuestro caso eliminamos el ultimo que creamos.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash clear

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/fusion_trip (some_feature)
$ git stash list
```

Y con **git stash clear** eliminamos todos los stash.

## Git rebase



En el caso del rebase, partimos de un commit base (P) desde el cual se crean dos ramas: la rama **feature**, que tiene dos commits (línea r1), y la rama master, que también avanza con dos commits propios (línea r2).

Cuando aplicamos un rebase de **feature** sobre **master**, lo que sucede es que Git toma los commits de **feature** y los vuelve a aplicar encima del último commit de **master**. Es decir, reescribe el historial de **feature** como si hubiera partido desde el punto actual de master, logrando así una línea de tiempo más limpia y lineal.

A diferencia de un merge, el **rebase no crea un commit de fusión, sino que reorganiza el historial para que parezca que todo se desarrolló secuencialmente**. El resultado es que el trabajo de **feature** queda "pegado" después del último commit de **master**.

En el siguiente ejemplo te muestro como lo aplique yo, aunque en teoría lo aplique bien, me es difícil explicarme cómo funciona.

```

$ mkdir rebase_

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop
$ cd rebase_

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_
$ git init
Initialized empty Git repository in C:/Users/JAVIER/Desktop/rebase_/.git/

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (master)
$ touch texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (master)
$ echo "111111111" > texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (master)
$ git add .
warning: in the working copy of 'texto.txt', LF will be replaced by CRLF the next time Git touches it

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (master)
$ git commit -m "commit original de master"
[master (root-commit) 4d5759d] commit original de master

```

Creamos un directorio, **rebase\_**, creamos un archivo **texto.txt**, y hacemos nuestro primer commit, en nuestro caso este será el commit Padre o P.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (master)
$ git branch feature_

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (master)
$ git checkout feature_
Switched to branch 'feature_'

```

Creamos una rama secundaria feature\_, y nos movemos allí, en este caso sería la línea r1 donde vamos a hacer más commits y cambios.

```
feature_)
$ echo "222222222" >> texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (
feature_)
$ git add .
warning: in the working copy of 'texto.txt', LF wi
ll be replaced by CRLF the next time Git touches i
t

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (
feature_)
$ git commit -m "commit 1 FEATURE"
[feature_ 361b0b4] commit 1 FEATURE
1 file changed, 1 insertion(+)

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (
feature_)
$ echo "333333333" >> texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (
feature_)
$ git add .
warning: in the working copy of 'texto.txt', LF wi
ll be replaced by CRLF the next time Git touches i
t

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (
feature_)
$ git commit -m "commit 2 FEATURE"
[feature_ bf959b4] commit 2 FEATURE
```

Agregamos dos cambios más, y creamos commits de ellos, en este caso 2 commits.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (feature_)
$ git checkout master
Switched to branch 'master'

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (master)
$ echo "4444444444" >> texto.txt

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (master)
$ git add .
warning: in the working copy of 'texto.txt', LF will be replaced by CRLF the next time Git touches it

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (master)
$ git commit -m "commit 2 master"
[master ba45ca5] commit 2 master
1 file changed, 1 insertion(+)

```

Volvemos a nuestra rama **master**, en el cual hacemos un cambio adicional y commiteamos, donde creamos la línea de trabajo **r2**.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (master)
$ git checkout feature_
Switched to branch 'feature_'

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (feature_)
$ git rebase master
Auto-merging texto.txt
CONFLICT (content): Merge conflict in texto.txt
error: could not apply b71ca9b... commit 1 FEATURE
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
hint: Disable this message with "git config set advice.mergeConflict false"
Could not apply b71ca9b... commit 1 FEATURE

```

Si volvemos a nuestra rama secundaria, y hacemos un rebase, nos dará errores, por un conflicto que hay entre los archivos, que debemos arreglar manualmente.

```

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (feature_|REBASE 1/2)
# nano texto.txt

```

Nótese que el texto azul nos dice que nos encontramos en medio de un rebase, y que son **2 conflictos a solucionar**.

```
GNU nano 8.3      texto.txt
1111111111
<<<<<<< HEAD
4444444444
=====
2222222222
>>>>>>> b71ca9b (commit 1 FEATURE)
```

Al ejecutar el **comando git rebase - -continue**, se abrirá un editor de texto con el mensaje del commit actual.

Este mensaje corresponde al mismo commit que estamos re-aplicando durante el rebase.

Git nos da la opción de modificar el mensaje o mantenerlo tal cual, nosotros lo modificaremos.

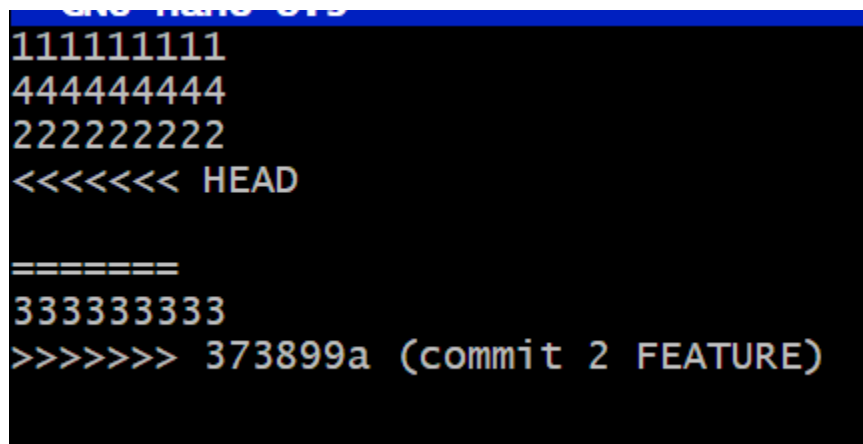
1. teclea la letra “i”, ya que este significa insert y nos permite modificar nuestro mensaje, el de color amarillo oscuro

2. modificamos el nombre del commit, agregamos un “\_modificado1”

3. para guardar cambios, tecleamos **Esc**, y en la parte de abajo escribimos0 “:wq”, que signigica **write and quite** (escribir y salir)

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (feature_|REBASE 2/2)
```

Aun tenemos un conflicto, volvemos a abrir el editor con nano.



```
1111111111
4444444444
2222222222
<<<<<<< HEAD

=====
3333333333
>>>>>>> 373899a (commit 2 FEATURE)
```

```
1111111111
```

```
4444444444
```

```
2222222222
```

```
<<<<<<< HEAD
```

Es lo que tenemos en nuestro archivo en nuestra rama base, y

3333333333 indica el contenido que viene del commit que estás re-aplicando en el rebase, commit 2 FEATURE

Decidimos mantener ambos, así que solo borramos los signos y nombres que hay.



```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (feature_|REBASE 2/2)
$ git add .

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (feature_|REBASE 2/2)
$ git rebase --continue|
```

Y volvemos a hacer lo de cambiar el nombre, no le tome captura, pero digamos que sí.

```
commit 1 FEATURE_modificado1

# Conflicts:
#   texto.txt

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# interactive rebase in progress; onto 371e6e3
# Last command done (1 command done):
#   pick b71ca9b commit 1 FEATURE
# Next command to do (1 remaining command):
#   pick 373899a commit 2 FEATURE
# You are currently rebasing branch 'feature_' on '371e6e3'.
#
# Changes to be committed:
#   modified:   texto.txt
#
```

Le cambie el nombre a “commit 1 FEATURE\_modificado2”

```
[detached HEAD 84089cf] commit 2 FEATURE_modificado2
1 file changed, 1 insertion(+)
Successfully rebased and updated refs/heads/feature_.

JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (feature_)
```

Ya no tenemos rebases pendientes. Como te mostré al principio, rebase sirve para hacer que la historia de los commits sea lineal, eliminando la ramificación extensa que genera un merge.

```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (feature_|REBASE 2/2)
$ git log --oneline --graph --all
* a899312 (HEAD) commit 1 FEATURE_rebase
* ba45ca5 (master) commit 2 master
| * bf959b4 (feature_) commit 2 FEATURE
| * 361b0b4 commit 1 FEATURE
|/
* 4d5759d commit original de master
```

Cuando usamos merge tenemos una ramificación similar



```
JAVIER@LAPTOP-VJRJVLQT MINGW64 ~/desktop/rebase_ (feature_)
$ git lg
* 84089cf (HEAD -> feature_) commit 2 FEATURE_modificado2
* 1821cd5 commit 1 FEATURE_modificado1
* 371e6e3 (master) commit 2 master
* c975126 commit original de master
```

Con rebase, la historia queda lineal: los commits de la rama **feature** se integran sobre **master** de forma **secuencial**, como si se hubieran creado directamente a partir de master, sin multiramas.

## Conclusión

Espero te haya servido, gracias por leer o descargar, complementa la información con libros y documentos de internet, si quieres invitarme un cafecito: <https://ko-fi.com/javieralanventura>

Y mi correo [javier.alan.ventura12@gmail.com](mailto:javier.alan.ventura12@gmail.com), por si tienes preguntas, sugerencias o quieres corregirme en algo del texto.

## Bibliografía

GeeksforGeeks. (s.f.). *Git merge*. Recuperado el 15 de mayo de 2025, de <https://www.geeksforgeeks.org/git-merge/>

Atlassian. (s.f.). *Git rebase*. Recuperado el 15 de mayo de 2025, de <https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase>

Moure, Brais. *Git y GitHub desde cero: Guía de estudio teórico-práctica paso a paso más curso en vídeo*. 2.<sup>a</sup> ed., Leanpub, 2024. <https://leanpub.com/git-github>