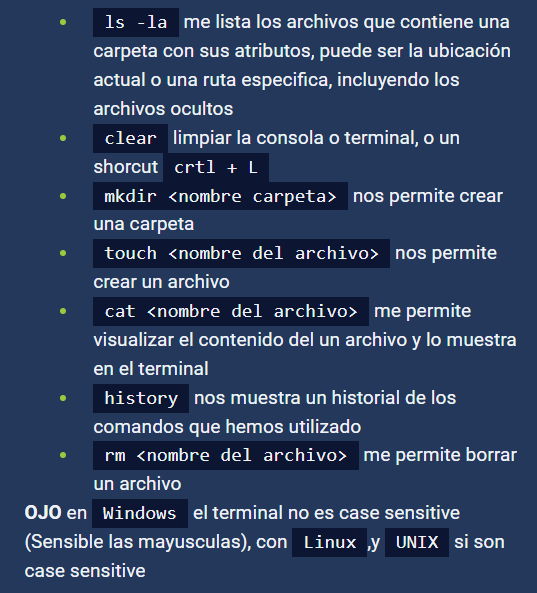
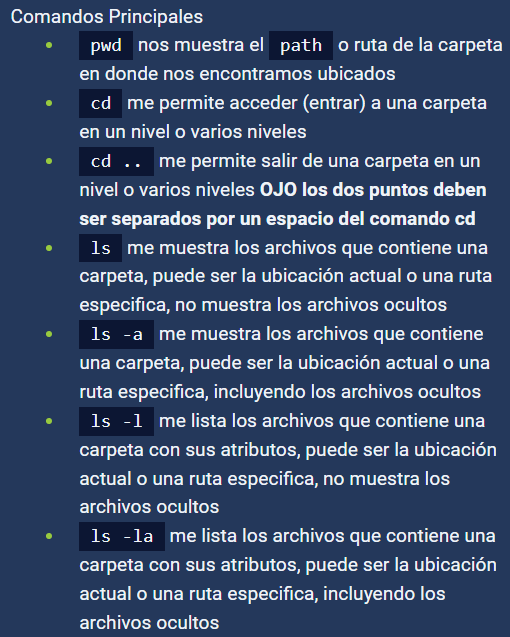
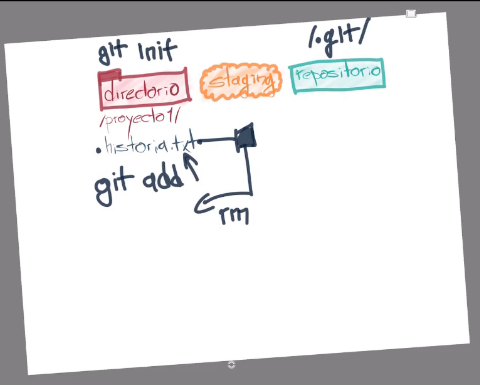
**GIT**

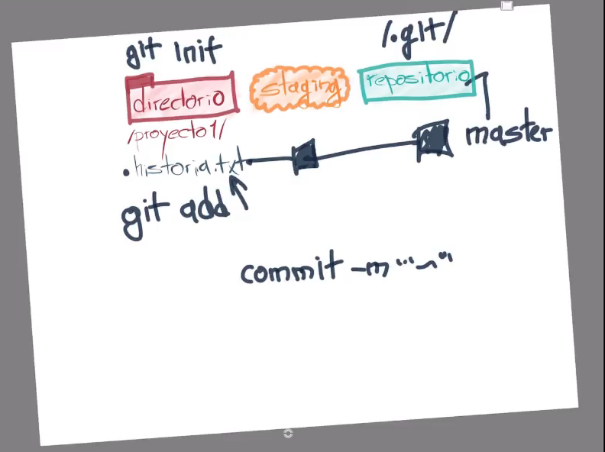


Tenemos una carpeta o “directorio”, en este caso proyecto 1, en el cual creamos un archivo (vacio.txt), en la consola de git lo inicializamos usando el código “git init” hace 2 cosas, se crea un área en memoria RAM desconectada llamada “staging” que es donde en un principio se van a ir guardando los cambios que hagamos temporalmente y crea el “repositorio”.

El “repositorio” conocida como la carpeta oculta en el directorio llamada “. git” que es donde se quedaran todos los cambios del archivo al final.

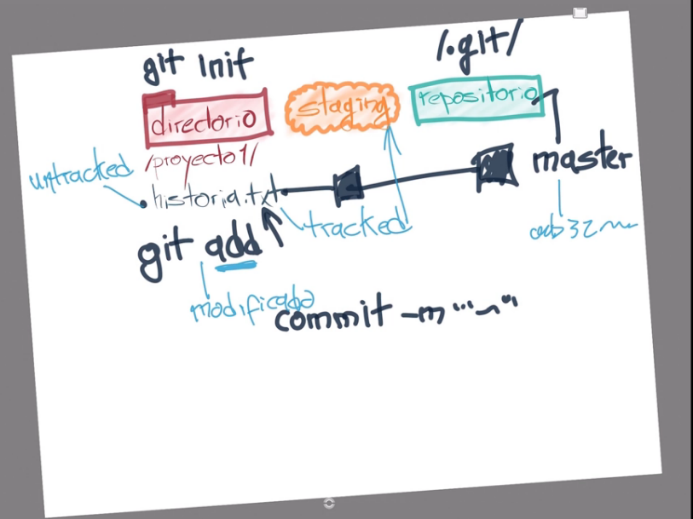
Una vez hechos los cambios, guardas los avances en el “staging” usando el comando

“git add (nombreDelArchivo)”, al hacer esto el archivo se encuentra “esperando” a que lo lleves al repositorio, si no lo deseas puedes removerlo utilizando el comando

“git rm (nombreDelArchivo)”

Si no lo remueves el archivo, lo que se hace es crear un “commit”, podes agregarle un mensaje al commit escribiendo “commit -m “mensaje” ”

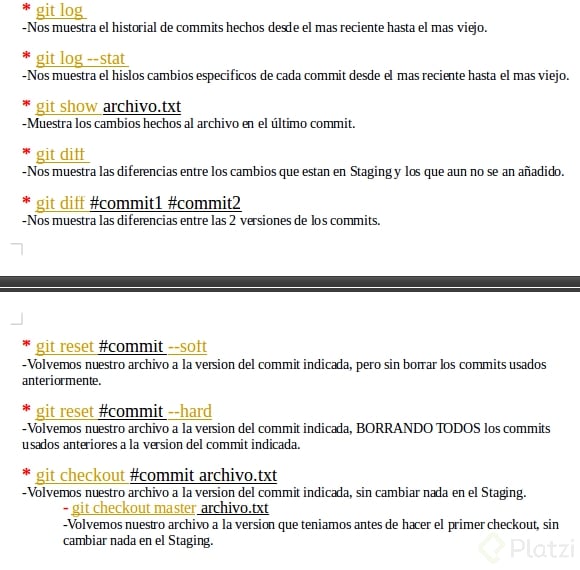
Esto es altamente recomendado al hacerlo, el archivo se va al repositorio que es llamado “master” por defecto

**Estados de los archivos**

Si todavía no has utilizado el comando “git add” en el archivo, este queda sin rastrear o “untracked”

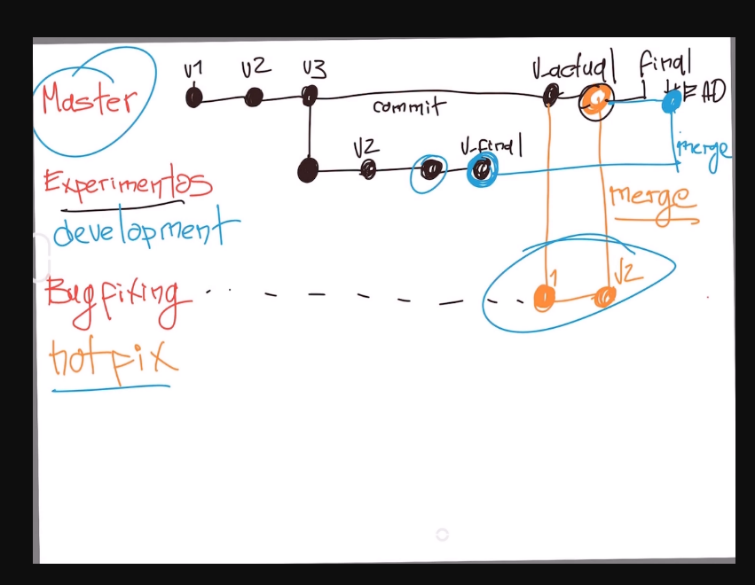
Al modificar el archivo y guardarlo, luego usando “git add” es cuando el archivo queda como “tracked” es decir que es rastreado y pasa a formar parte del “staging”

Luego del “commit” pasa a formar parte del repositorio y se le asigna un nombre a esa versión del archivo (nombres largos tales como “0c1a2713819d76f5f30a9289570db14fdfffc136”)



**RAMAS**

Una rama o “Branch” es una versión del código del proyecto sobre el que estás trabajando. Estas ramas ayudan a mantener el orden en el control de versiones y manipular el código de forma segura.



Estas son las ramas base de un proyecto en Git:

* **Rama main** (Master)
* **Rama development**
* **Rama hotfix**

1. **Rama Main (Master)**

Por defecto, el proyecto se crea en una rama llamada Main (anteriormente conocida como Master). Cada vez que añades código y guardas los cambios, estás haciendo un “commit”, que es añadir el nuevo código a una rama. Esto genera nuevas versiones de esta rama o “Branch”, hasta llegar a la versión actual de la rama Main.

1. **Rama Development**

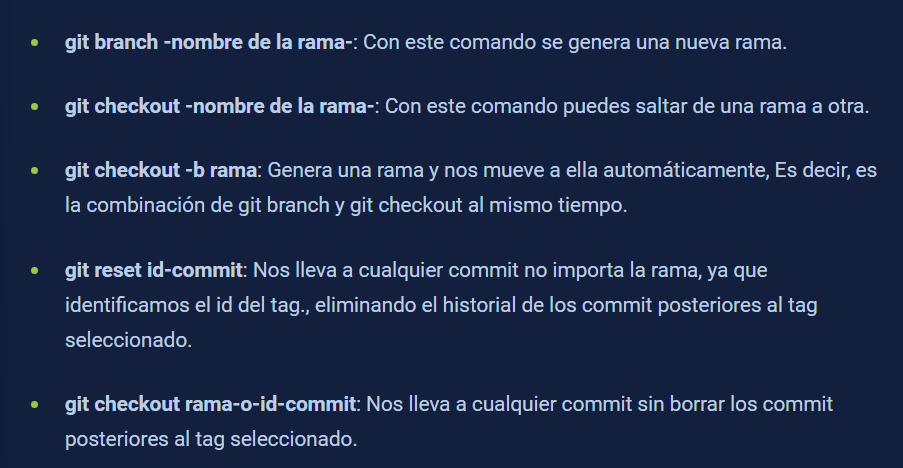
Cuando decides hacer experimentos, puedes generar ramas experimentales (usualmente llamadas development), que están basadas en alguna rama main, pero sobre las cuales puedes hacer cambios a tu gusto sin necesidad de afectar directamente al código principal.

1. **Rama Hotfix**

En otros casos, si encuentras un bug o error de código en la rama Main (que afecta al proyecto en producción), tendrás que crear una nueva rama (que usualmente se llaman bug fixing o hot fix) para hacer los arreglos necesarios. Cuando los cambios estén listos, los tendrás que fusionar con la rama Main para que los cambios sean aplicados. Para esto, se usa un comando llamado “Merge”, que mezcla los cambios de la rama que originaste a la rama Main.

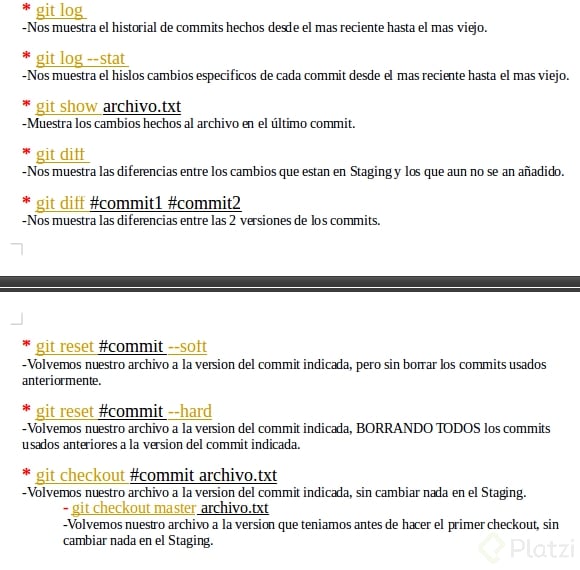
**Cómo crear un “Branch” en Git**

El comando “git branch” permite crear una rama nueva. Si quieres empezar a trabajar en una nueva función, puedes crear una rama nueva a partir de la rama master con “git branch new\_branch”. Una vez creada, puedes usar “git checkout new\_branch” para cambiar a esa rama.



Recuerda que todas tus versiones salen de la rama principal o Master y de allí puedes tomar una versión específica para crear otra rama de versiones.

RESET Y CHEKOUT



Para volver a “commits” previos, borrando los cambios realizados desde ese “commit”, podemos utilizar:

* git reset --soft [IC]: elimina los cambios hasta el staging area
* git reset --mixed [IC]: elimina los cambios hasta el working area
* git reset --hard [IC]: regresa hasta el commit del [SHA-1]

**Donde el IC es el identificador del commit!!!**

Para navegar entre versiones por medio del “commit” se utiliza el “checkout” de la siguiente manera:

* git checkout [IC] nombreDelArchivo: sirve para volver hacia el commit ingresado
* git checkout master nombreDelArchivo: sirve para volver a la versión final

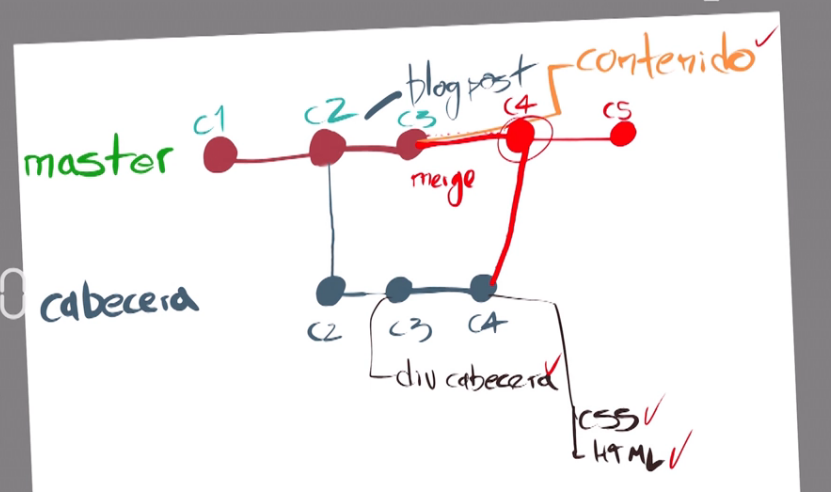
**Cómo hacer “Merge”**

Producir una nueva rama se conoce como Checkout. Unir dos ramas lo conocemos como Merge.

Cuando haces “merge” de estas ramas con el código principal, su código se fusiona originando una nueva versión de la rama master (o main) que ya tiene todos los cambios que aplicaste en tus experimentos o arreglos de errores.

Podemos generar todas las ramas y “commits” que queramos. De hecho, podemos aprovechar el registro de cambios de Git para producir ramas, traer versiones viejas del código, arreglarlas y combinarlas de nuevo para mejorar el proyecto.

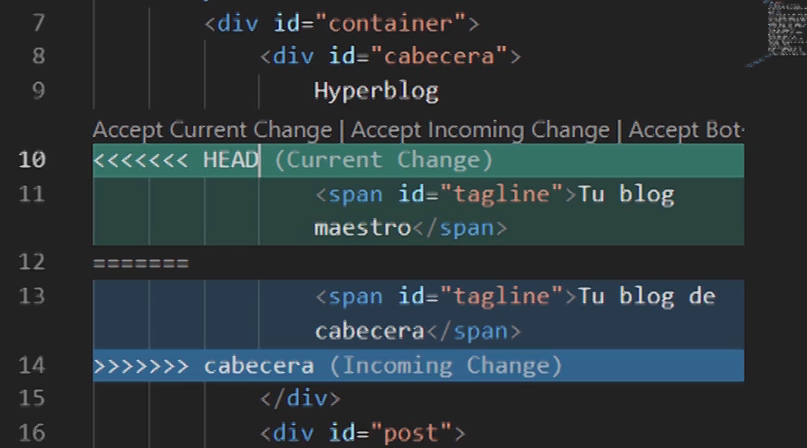
Solo ten en cuenta que combinar estas ramas (hacer “merge”) puede generar conflictos. Algunos archivos pueden ser diferentes en ambas ramas. Git es muy inteligente y puede intentar unir estos cambios automáticamente, pero no siempre funciona. En algunos casos, somos nosotros los que debemos resolver estos conflictos a mano



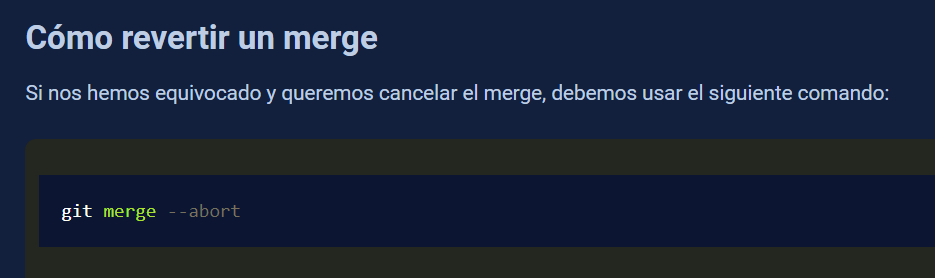
Git no borra nada, a menos que nosotros se lo indiquemos. Cuando usamos los comandos git merge o git checkout estamos cambiando de rama o creando un nuevo commit, no borrando ramas ni commits (recuerda que puedes borrar commits con git reset y ramas con git branch -d).

Git puede resolver algunos conflictos automáticamente, pero algunas veces no sabe cómo resolver estas diferencias, por ejemplo, cuando dos ramas diferentes hacen cambios distintos a una misma línea.

Esto lo conocemos como conflicto y lo podemos resolver manualmente. Solo debemos hacer el merge, ir a nuestro editor de código y elegir si queremos quedarnos con alguna de estas dos versiones o algo diferente. Algunos editores de código como Visual Studio Code nos ayudan a resolver estos conflictos sin necesidad de borrar o escribir líneas de texto.

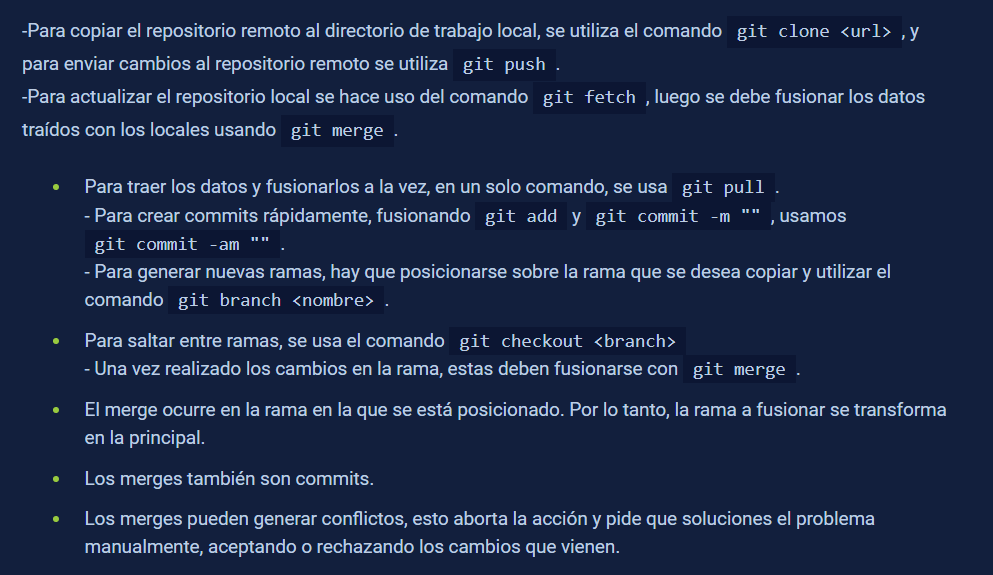
Recuerda que siempre debemos crear un nuevo commit para aplicar los cambios del merge. Si Git puede resolver el conflicto, hará commit automáticamente. Pero, en caso de no pueda resolverlo, debemos solucionarlo y hacer el commit.

Los archivos con conflictos por el comando git merge entran en un nuevo estado que conocemos como **Unmerged.**

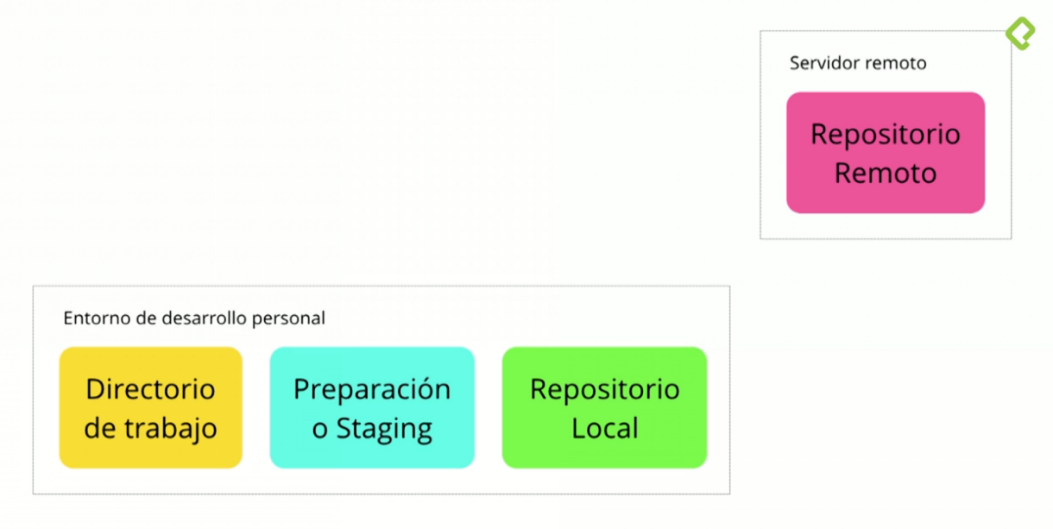


**Conflictos en repositorios remotos**

Al trabajar con otras personas, es necesario utilizar un repositorio remoto.



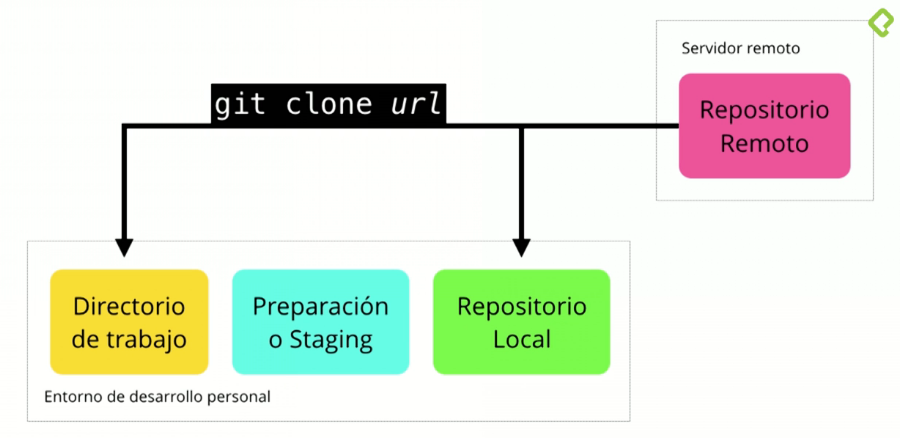
**Git Clone, Fetch and Pull**



El repositorio remoto es el mismo repositorio que estás trabajando, pero online, lo que hace que otras personas puedan verlo y modificarlo también.

Estamos hablando de GitHub, GitLab, BigPocket, o un servidor propio montado.

Para traernos datos de un servidor remoto, lo que hacemos es clonar (con el url) el repositorio remoto trae una copia del **main** al directorio de trabajo y crea una base de datos con todos los cambios históricos en el repositorio local propio.

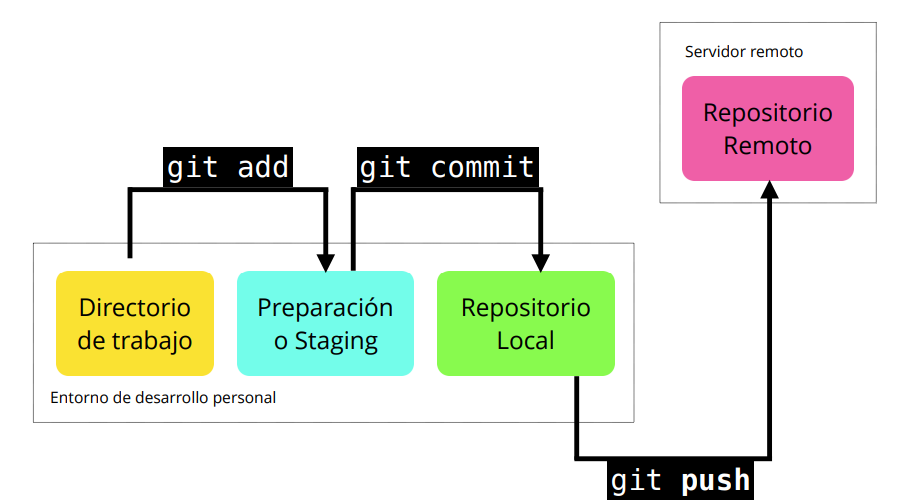


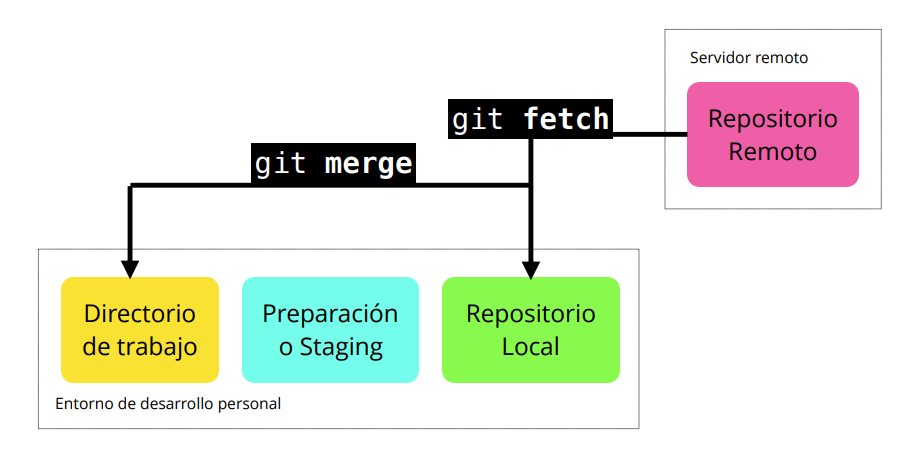
**Git Push:** Se envía la última versión del repositorio local hacia el repositorio remoto. Si surgen problemas, se lidian con ellos.

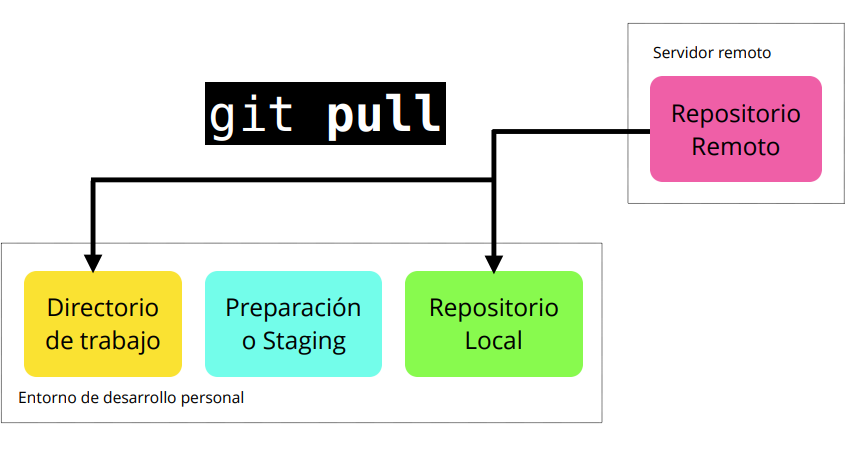
**Git Fetch:** Si quiero traer una actualización del repositorio remoto hacia mi repositorio local, se utiliza un git fetch **PERO NO** lo copia en los archivos

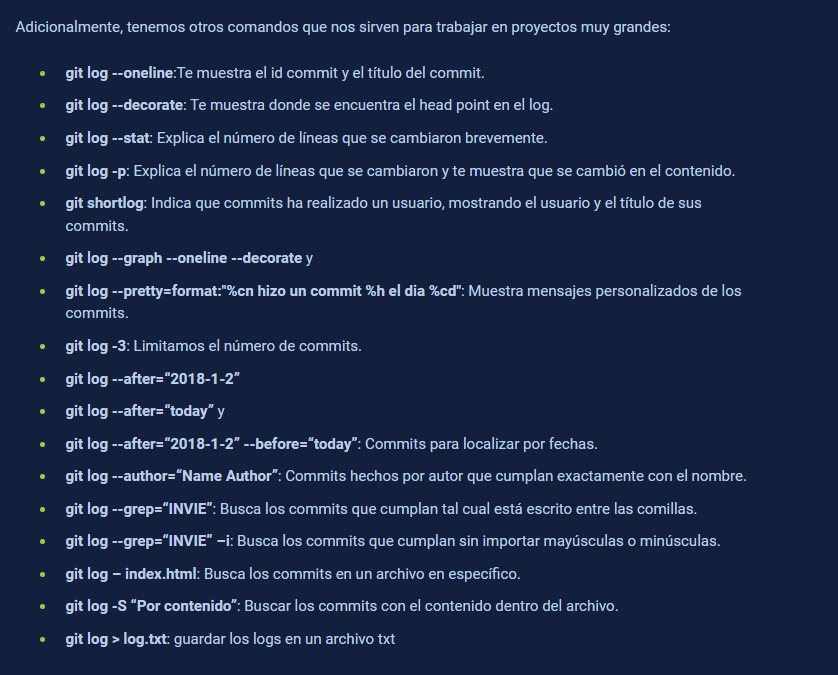
**Git Merge:** Para que lo copie en los archivos, hay que fusionar la última versión del repositorio local con la versión actual.

**Git Pull:** Fusión de git merge y git fetch.





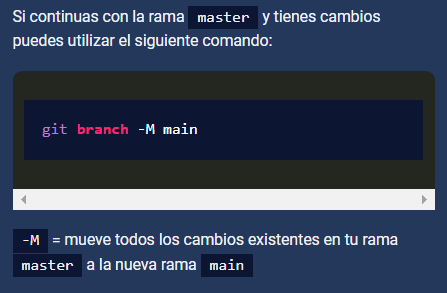




**¡¡¡Importante!!!**

**Cambios en GitHub: de master a main**

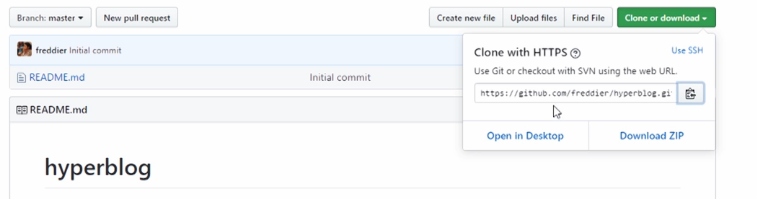
Ya no se utilizará el nombre de la cabecera por defecto “master” sino “main”



**GitHub:**

GitHub es una plataforma que nos permite guardar repositorios de Git que podemos usar como servidores remotos y ejecutar algunos comandos de forma visual e interactiva (sin necesidad de la consola de comandos).

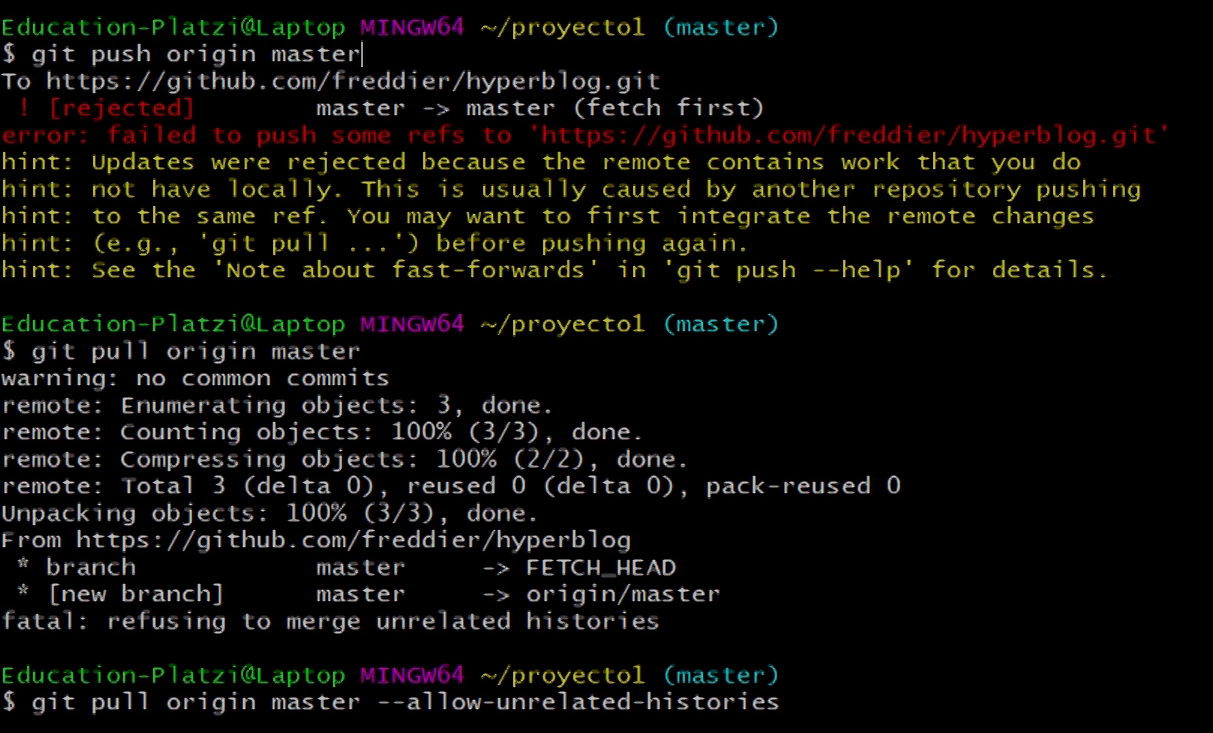
Para conectar un repositorio remoto creado en GitHub con nuestro repositorio local, utilizamos un URL en código HTTPS.



Luego se agrega un origen remoto de nuestros archivos con “git remote add origin URL”

PUSH: código para hacer un push hacia el escritorio remoto “git push origin main” (main como la “cabecera que queremos subir a GitHub”).

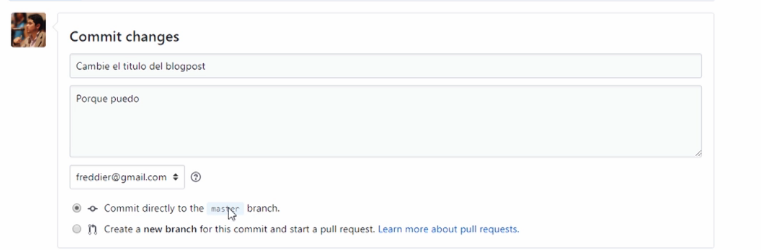
Si esto se realiza por primera vez, es posible que el programa no nos deje hacerlo, ya que el remoto contiene trabajo que no tienes en el repositorio local, antes hay que integrar estos cambios remotos desde el repositorio remoto hacia el repositorio local antes de hacer el PUSH.



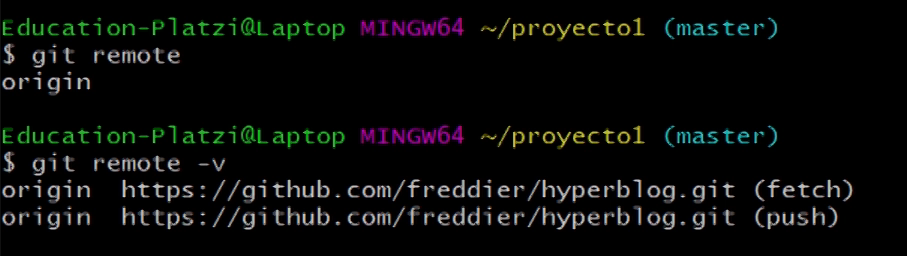
Cuando hagamos el PULL para traer esos archivos desde el repositorio remoto, se creará un error fatal que diga que el programa se rehúsa a traer los archivos con historias (grupo de commits) no relacionadas, para forzar esto se utiliza el siguiente código.

git pull origin main --allow-unrelated-histories

Al actualizar el repositorio remoto en GitHub se agregará de manera automática TODO lo que subimos con git pull mostrándonos los archivos, los cambios históricos (commits) y el usuario que los realizo, permitiéndonos también, editar los archivos desde el mismo GitHub a forma de commit.



**Otras acciones:**



remote nos muestra nuestro origin y remote -v nuestros URL para hacer un fetch y un push.

**README.md:**

El README.md es el archivo que veremos por defecto al entrar a un repositorio. Es una muy buena práctica configurarlo para describir el proyecto, los requerimientos y las instrucciones que debemos seguir para contribuir correctamente.



**RAW:** Muestra el código plano que creo ese texto

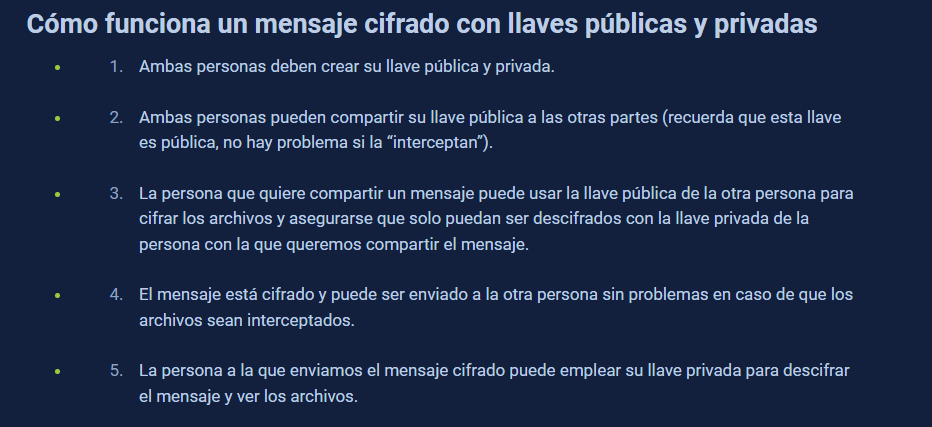
**BLAME:** Muestra quien tiene la “culpa” de quien hizo algún cambio o creo un nuevo archivo

**HISTORY:** Muestra la historia del archivo (al igual que git log)

**Llaves Públicas y Privadas**

Las llaves públicas y privadas, conocidas también como cifrado asimétrico de un solo camino, sirven para mandar mensajes privados entre varios nodos con la lógica de que firmas tu mensaje con una llave pública vinculada con una llave privada que puede leer el mensaje.

Las llaves públicas y privadas nos ayudan a cifrar y descifrar nuestros archivos de forma que los podamos compartir sin correr el riesgo de que sean interceptados por personas con malas intenciones.



En el entorno local, se crea una llave privada y una publica, le asignas al repositorio una llave publica a través de GitHub y el protocolo SSH (Secure Shield), mientras que te quedas la llave privada. Al hacerlo GitHub te envía una llave publica con tu cifrado de llave pública.



Con este comando comprobamos que los servidores de las llaves estén funcionando.

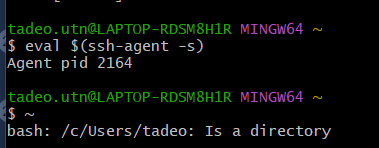
Agent significa que el servidor esta corriendo.

PID Process ID

El numero simboliza que el proceso está corriendo

**Dentro del GIT BASH**

El símbolo ~ sirve para no tener que escribir toda la dirección nuevamente de donde este creada la llave. Esto es solamente una variable.



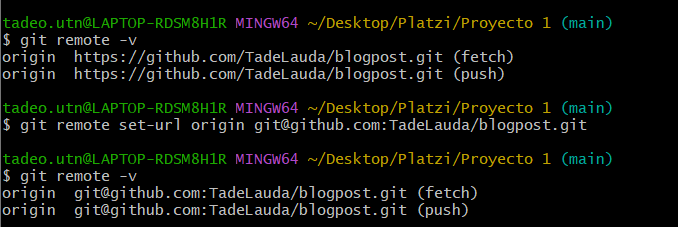
$cd ~/.ssh/ es un atajo para encontrar la carpeta donde está la llave

Con ls -al se puede ver la lista de llaves públicas y privadas.

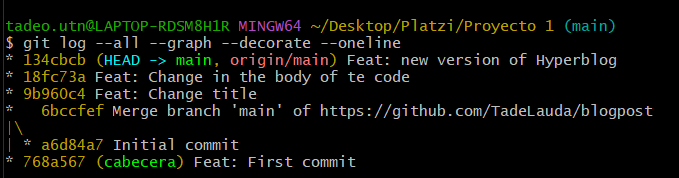
Con el ultimo comando se añade la llave, siempre es la id\_rsa, NUNCA la id\_rsa.pub ya que esa es la llave privada.



Comando para cambiar URL del repositorio de HTTPS a SSH.



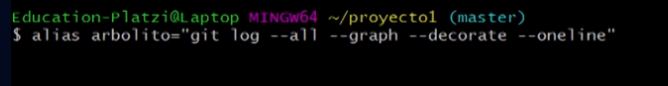
TAGS, ALIAS y Versiones



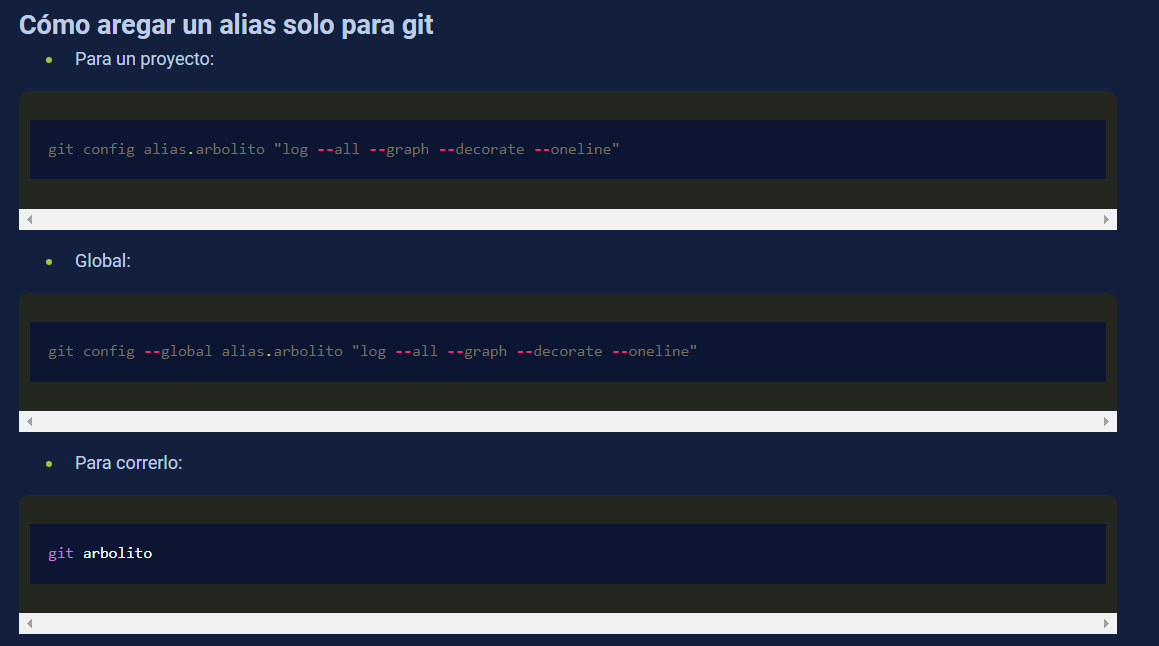
Comando para conocer TODOS los cambios hechos sobre un repositorio expresados a modo de grafico de historia lineal (árbol)

ALIAS:

Sirve para crear un “atajo” de un comando, cada programador usa sus propios alias

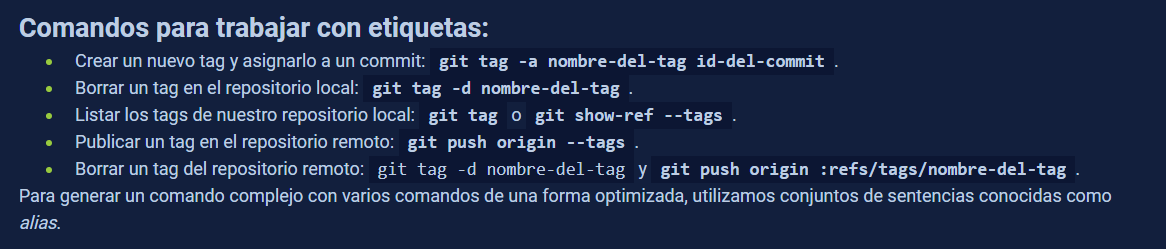


Se escribe $alias arbolito = “comando”



TAGS:

Los tags o etiquetas nos permiten asignar versiones a los commits con cambios más importantes o significativos de nuestro proyecto.

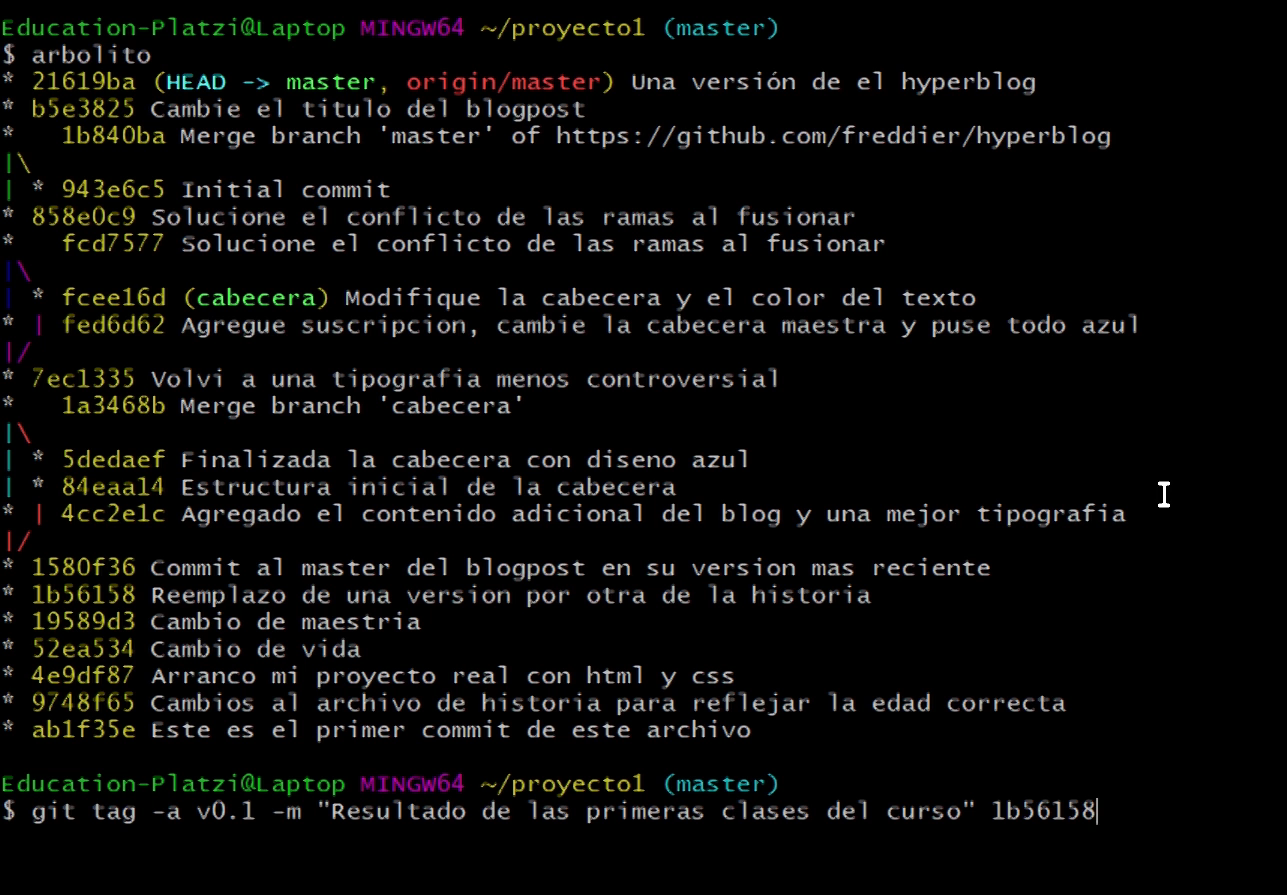


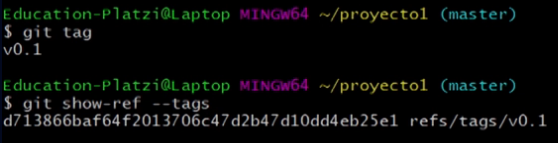
Para generar un comando complejo con varios comandos de una forma optimizada, utilizamos conjuntos de sentencias conocidas como alias.

¡¡¡IMPORTANTE!!!

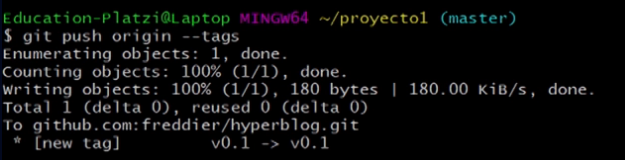
$gitk abre un software ya instalado donde se aprecia de manera mas visual los cambios.

Seleccionamos una commit especifico y le asignamos un tag de “versión”

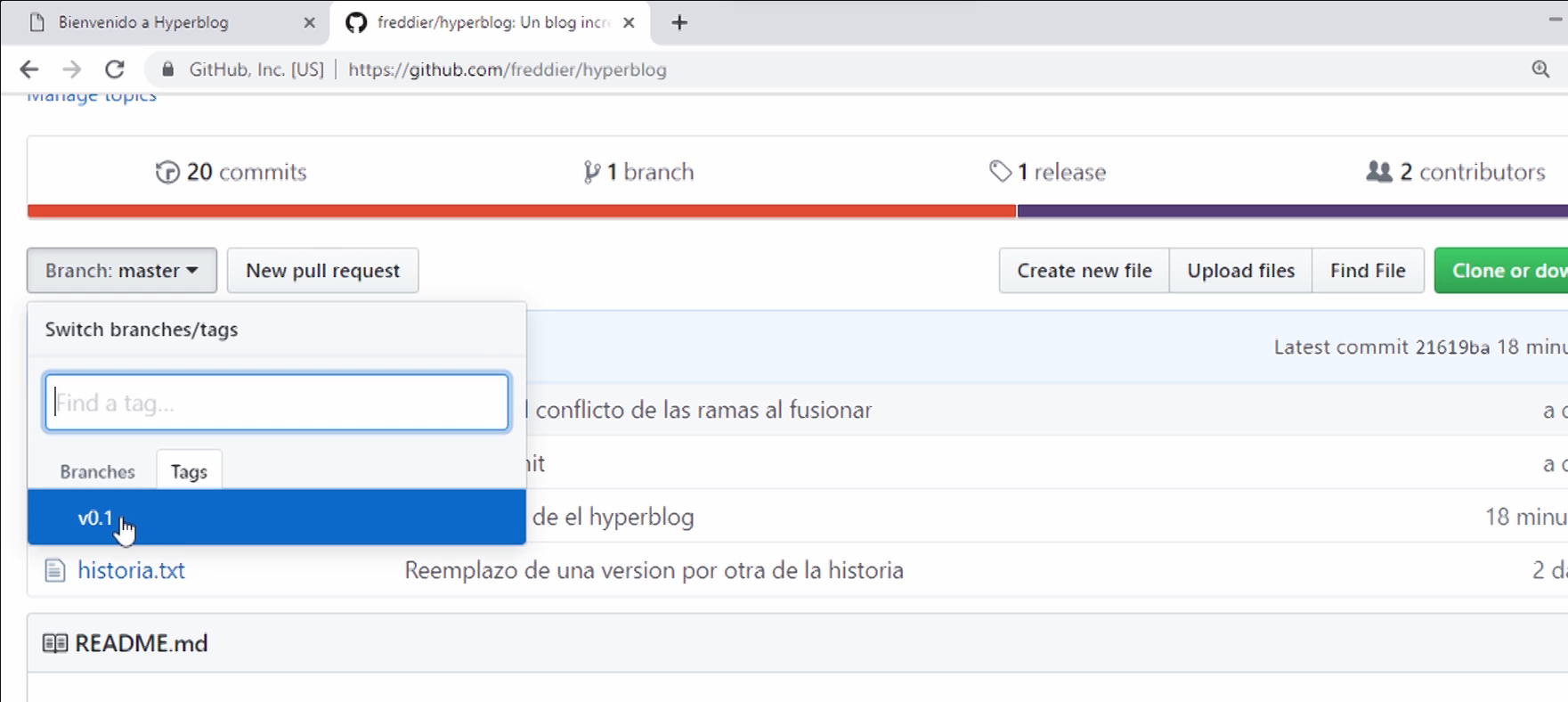




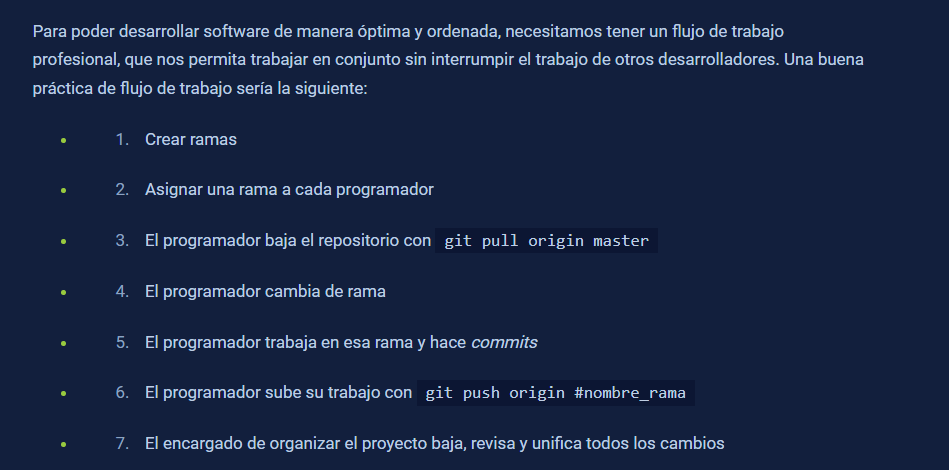
Luego de hacer esto se pushean los nuevos tags hacia GitHub con git push.



Estas etiquetas se verán reflejadas también en GitHub en la parte de “Tags”.



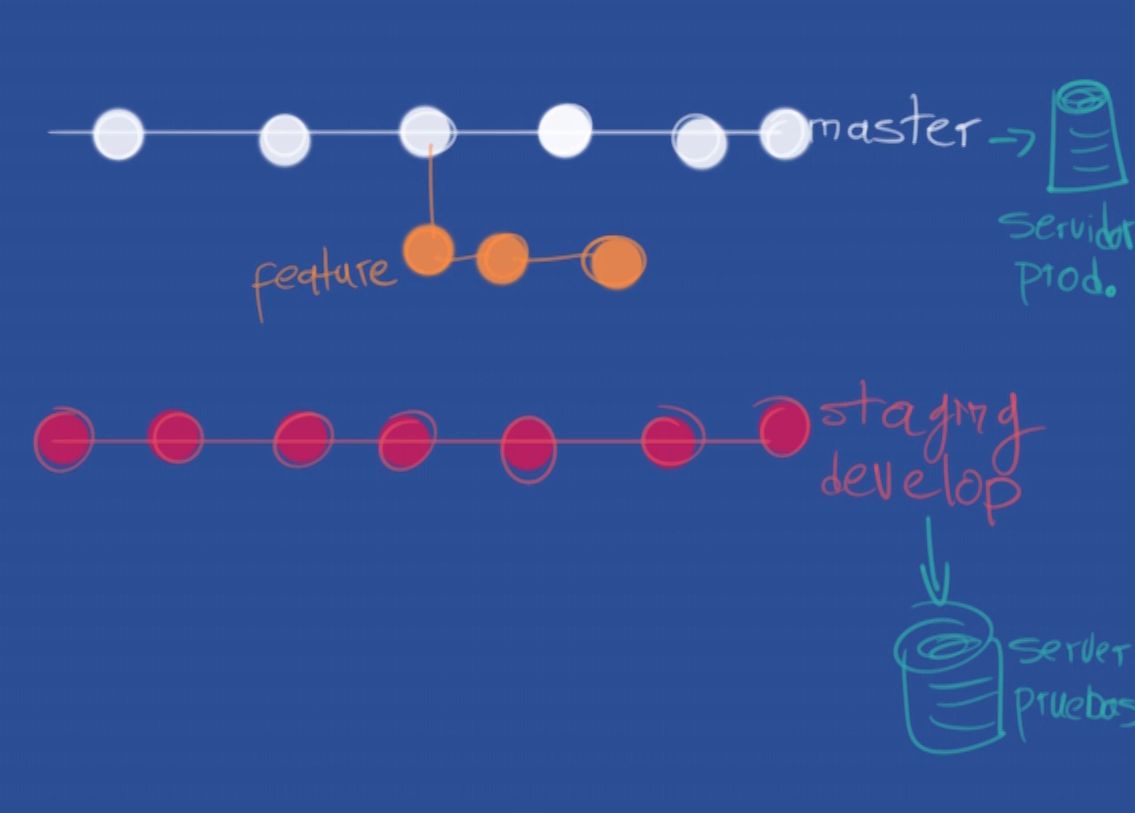
**Flujo de Trabajo:**

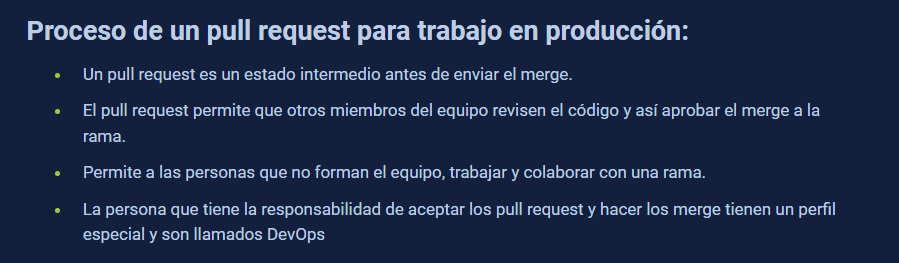


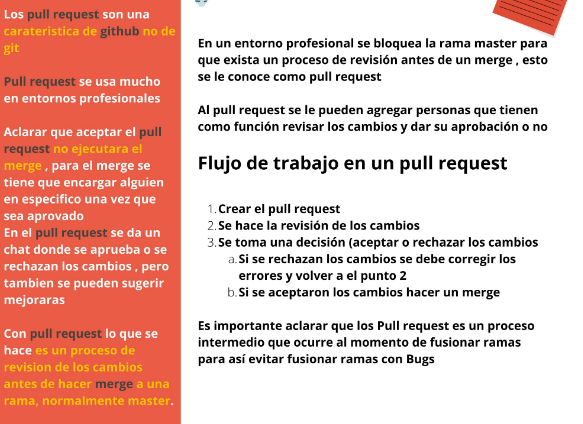
**Pull Request:**

Pull request es una funcionalidad de GitHub (en Gitlab llamada merge request y en Bitbucket push request), en la que un colaborador pide que revisen sus cambios antes de hacer merge a una rama, normalmente master (ahora conocida como main).

Al hacer un pull request, se genera una conversación que pueden seguir los demás usuarios del repositorio, así como autorizar y rechazar los cambios.









**Fork:**

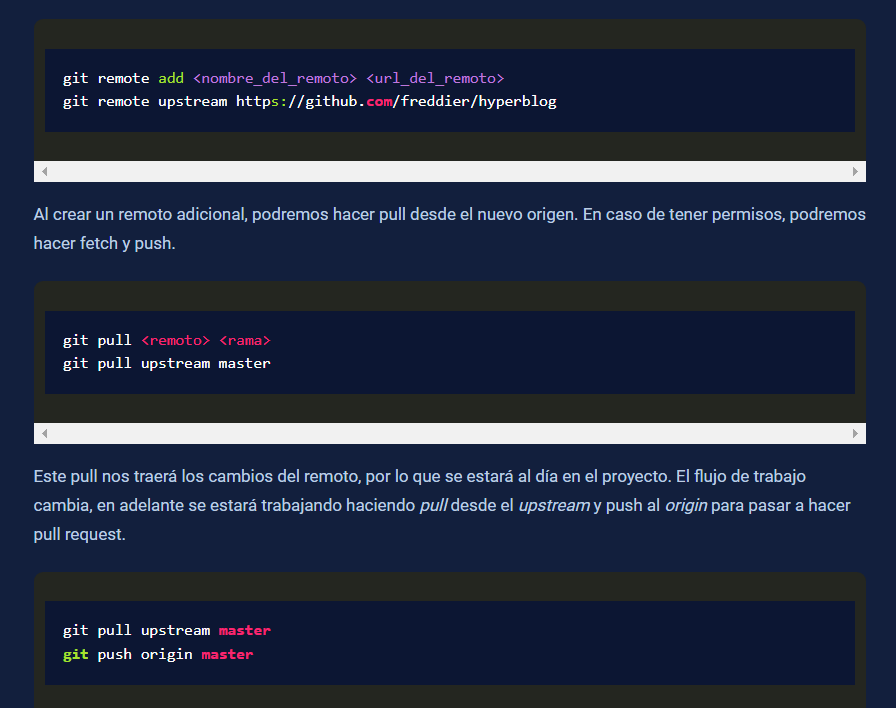
Los forks o bifurcaciones son una característica única de GitHub en la que se crea una copia exacta del estado actual de un repositorio directamente en GitHub. Este repositorio podrá servir como otro origen y se podrá clonar (como cualquier otro repositorio). En pocas palabras, lo podremos utilizar como un nuevo repositorio git cualquiera

Un fork es como una bifurcación del repositorio completo. Comparte una historia en común con el original, pero de repente se bifurca y pueden aparecer varios cambios, ya que ambos proyectos podrán ser modificados en paralelo y para estar al día un colaborador tendrá que estar actualizando su fork con la información del original.

Los forks son importantes porque es la manera en la que funciona el open source, ya que, una persona puede no ser colaborador de un proyecto, pero puede contribuir al mismo, haciendo mejor software que pueda ser utilizado por cualquiera.

**Cómo se hace un fork remoto desde consola en GitHub:**

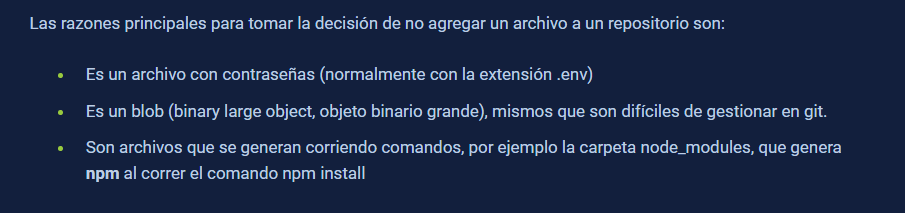
Al hacer un fork, GitHub sabe que se hizo el fork del proyecto, por lo que se le permite al colaborador hacer pull request desde su repositorio propio.



**Gitignore:**

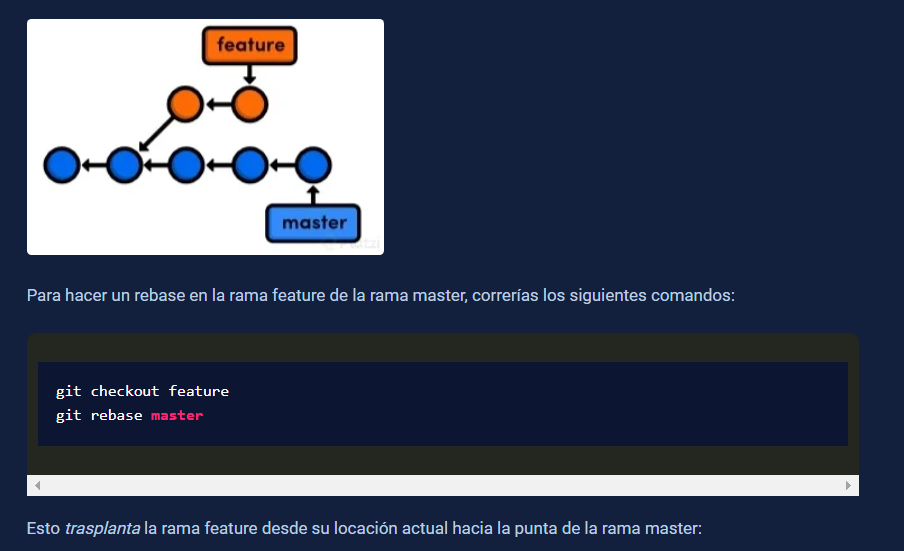
No todos los archivos que agregas a un proyecto deberían guardarse en un repositorio. Esto es porque hay archivos que no todo el mundo debería de ver, y hay archivos que al estar en el repositorio ralentizan el proceso de desarrollo.

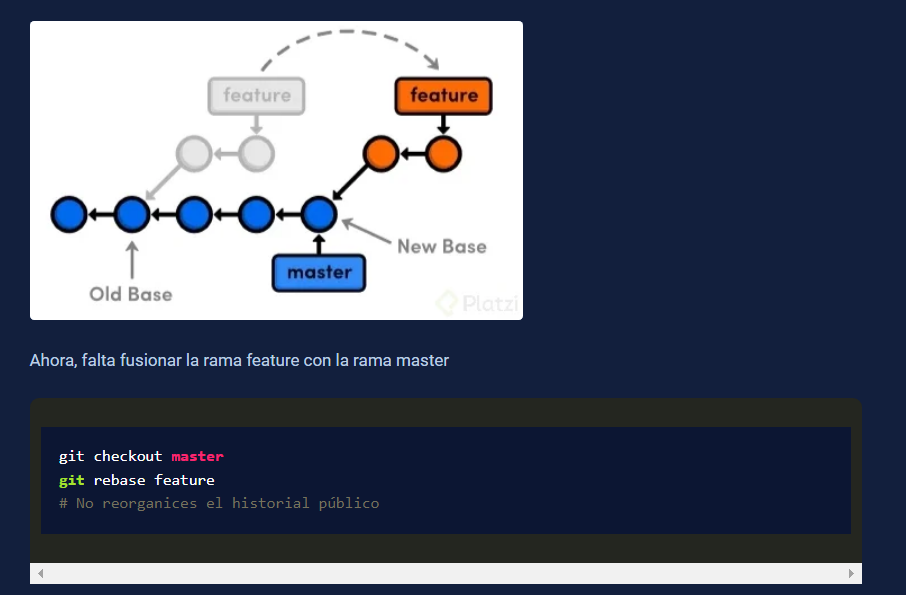
Para que no se suban estos archivos no deseados se puede crear un archivo con el nombre .gitignore en la raíz del repositorio con las reglas para los archivos que no se deberían subir: Aquí puedes ver la sintaxis de los .gitignore.



**Git Rebase: reorganizando el trabajo realizado**

Rebase es el proceso de mover o combinar una secuencia de confirmaciones en una nueva confirmación base. La reorganización es muy útil y se visualiza fácilmente en el contexto de un flujo de trabajo de ramas de funciones. El proceso general se puede visualizar de la siguiente manera.





Nunca debes reorganizar las confirmaciones una vez que se hayan enviado a un repositorio público. La reorganización sustituiría las confirmaciones antiguas por las nuevas y parecería que esa parte del historial de tu proyecto se hubiera desvanecido de repente.

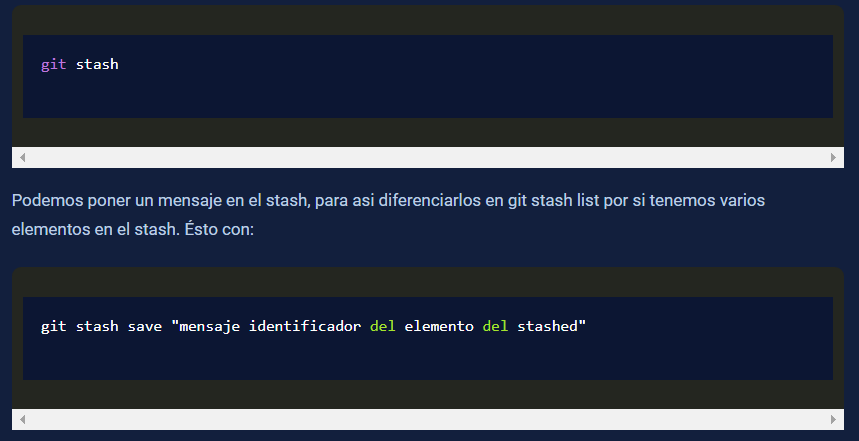
El comando rebase es \*\*\_una mala práctica, sobre todo en repositorios remotos. Se debe evitar su uso, pero para efectos de práctica te lo vamos a mostrar, para que hagas tus propios experimentos. Con rebase puedes recoger todos los cambios confirmados en una rama y ponerlos sobre otra.



**Git stash**

El comando git stash guarda el trabajo actual del Staging en una lista diseñada para ser temporal llamada Stash, para que pueda ser recuperado en el futuro.

Para agregar los cambios al stash se utiliza el comando:



**Comandos con Git stash:**

git stash : Guarda el trabajo actual de manera temporal. (Archivos modificados o eliminados)

git stash -u : Crea un stash con todos los archivos. (Añadiendo los creados Untracked)

git stash save “mensaje” : Crea un stash con el mensaje especificado.

git stash list : Permite visualizar todos los stash existentes.

git stash clear : Elimina todos los stash existentes.

git stash drop : Elimina el stash más reciente. El que tiene num\_stash=0.

git stash drop stash@{num\_stash} : Elimina un stash específico.

git stash apply : Aplica el stash más reciente. El que tiene num\_stash=0.

git stash apply stash@{num\_stash} : Aplica los cambios de un stash específico.

git stash pop : Aplica el stash más reciente y lo elimina. El que tiene num\_stash=0.

git stash pop stash@{num\_stash} : Aplica los cambios de un stash específico y elimina lo stash.

git stash branch nombre\_de\_rama : Crea una rama y aplica el stash mas reciente.

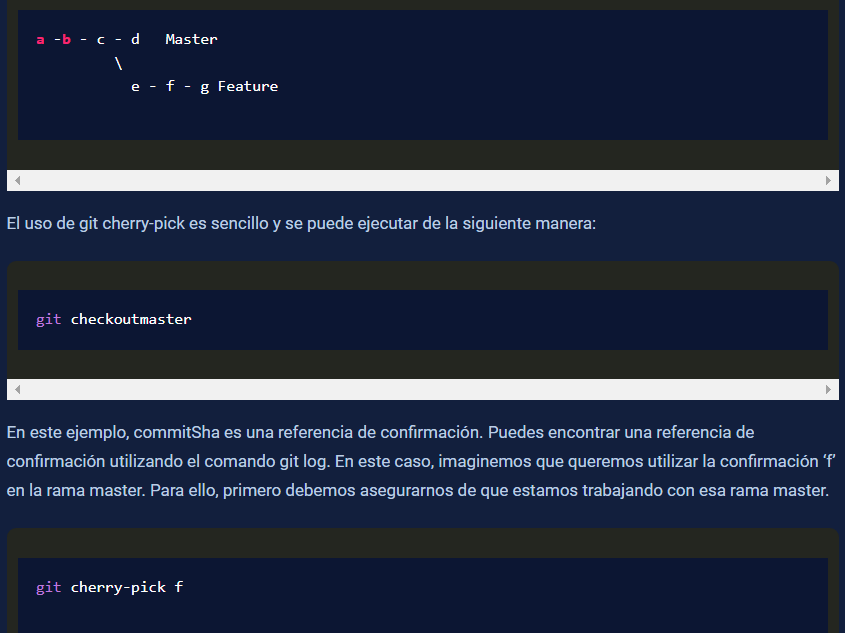
git stash branch nombre\_de\_rama stash@{num\_stash} : Crea una rama y aplica el stash especificado.

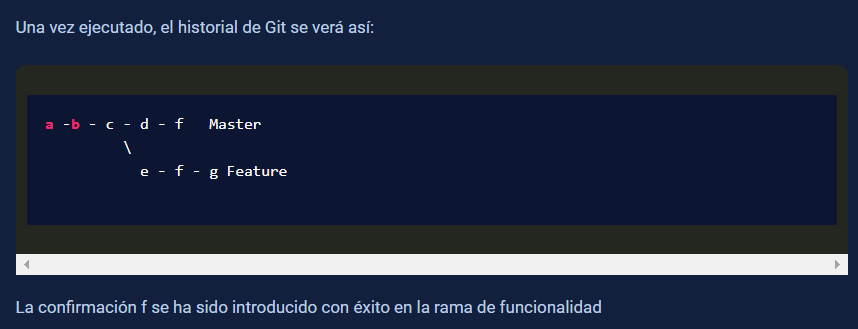
**Consideraciones:**

* El cambio más reciente (al crear un stash) SIEMPRE recibe el valor 0 y los que estaban antes aumentan su valor.
* Al crear un stash tomará los archivos que han sido modificados y eliminados. Para que tome un archivo creado es necesario agregarlo al Staging Area con git add [nombre\_archivo] con la intención de que git tenga un seguimiento de ese archivo, o también utilizando el comando git stash -u.
* Al aplicar un stash este no se elimina, es buena práctica eliminarlo.

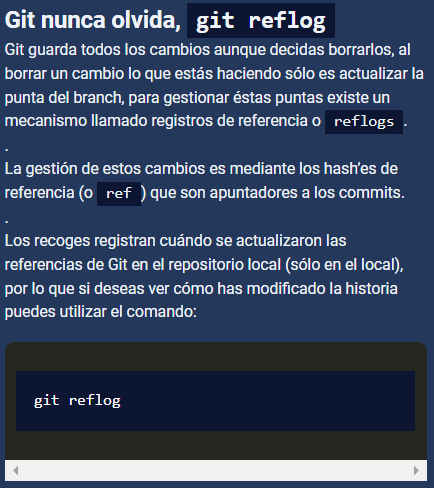
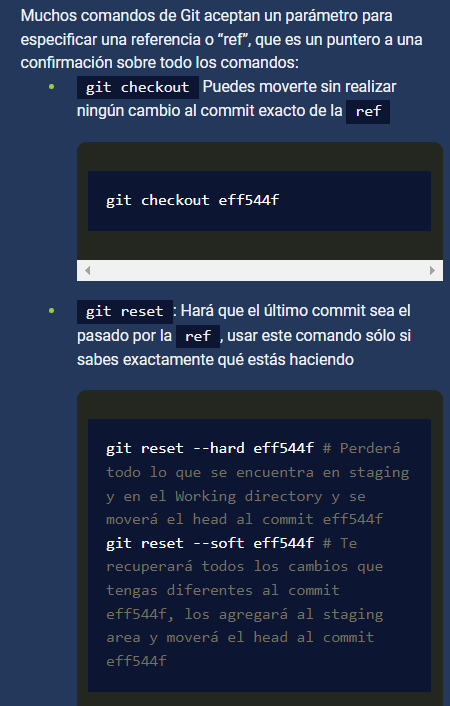
**Cherry-pick:**

Git Cherry-pick es un comando que permite tomar uno o varios commits de otra rama o branch sin tener que hacer un merge completo. Así, gracias a cherry-pick, podríamos aplicar los commits relacionados con nuestra funcionalidad en la rama master sin necesidad de hacer un merge.





**Git Reset y Reflog: úsese en caso de emergencia:**

* git reset --soft HashDelHEAD te mantiene lo que tengas en staging ahí.
* git reset --hard HashDelHEAD resetea absolutamente todo incluyendo lo que tengas en staging.

**Git amend:**

Remendar un commit con amend puede modificar el commit más reciente (enmendar) en la misma rama. Lo ejecutamos así:

