El staging es el lugar donde se guardan temporalmente los cambios, para luego ser llevados definitivamente al repositorio. El repositorio es el lugar donde se guardan todos los registros de los cambios realizados a los archivos.

Para iniciar un repositorio, o sea, activar el sistema de control de versiones de [Git](https://platzi.com/clases/1557-git-github/20215-que-es-git/) en tu proyecto, solo debes ejecutar el comando git init.

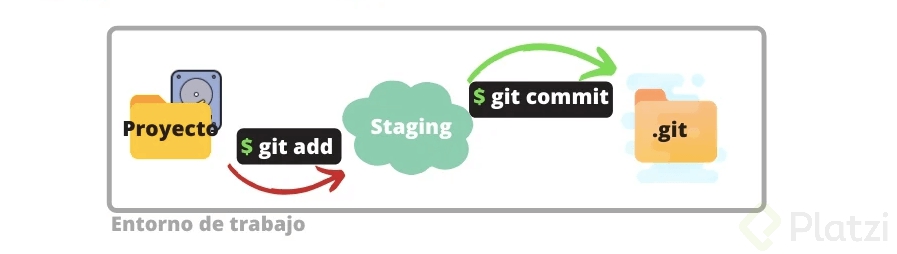
**¿Qué es el área de staging?**

El área de staging se puede ver como un limbo donde nuestros archivos están por ser enviados al repositorio o ser regresados a la carpeta del proyecto.

**¿Qué es git init?**

git inites el comando que activa git en nuestro proyecto creando un espacio en memoria RAM llamado staging y una carpeta .git.

Este comando se encargará de dos cosas: primero, crear una carpeta .git, donde se guardará toda la base de datos con cambios atómicos de nuestro proyecto; segundo, crear un área que conocemos como **staging**, que guardará temporalmente nuestros archivos (cuando ejecutemos un comando especial para eso) y nos permitirá, más adelante, guardar estos cambios en el repositorio (también con un comando especial).



**Cómo funciona el staging y el repositorio: ciclo básico de trabajo en git:**

El flujo de trabajo básico en git es algo así:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiéndolos a tu área de preparación (staging).
3. Confirmas los cambios (commit), lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de git.

Veamos a detalle las 3 secciones principales que tiene un proyecto en git.

**Working directory**

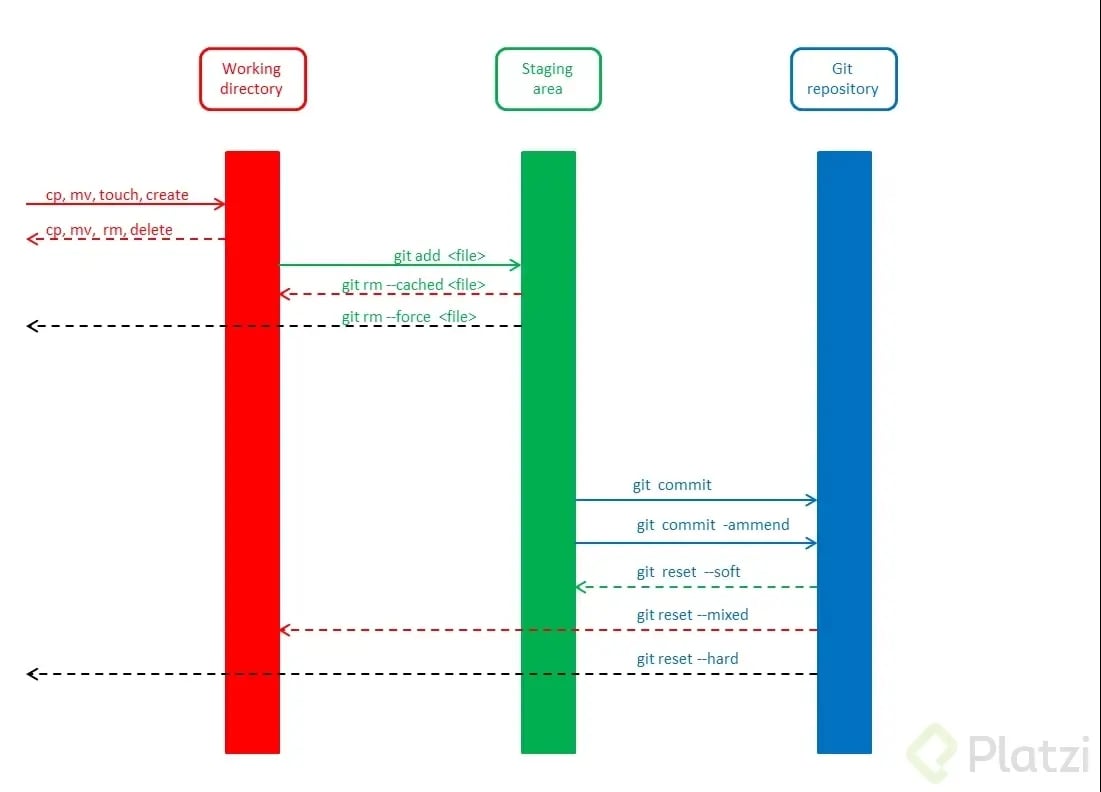
El *working directory* es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de git y se colocan en el disco para que los puedas usar o modificar.

**Staging area**

Es un área que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice (*index*).

**.git directory (repository)**

En el *repository* se almacenan los metadatos y la base de datos de los objetos para tu proyecto. Es la parte más importante de git (carpeta .git) y es lo que se copia cuando clonas un repositorio desde otra computadora.



**Ciclo de vida o estados de los archivos en git**

Cuando trabajamos con git, nuestros archivos pueden vivir y moverse entre 4 diferentes estados (cuando trabajamos con repositorios remotos pueden ser más estados, pero lo estudiaremos más adelante):

**Archivos tracked**

Son los archivos que viven dentro de git, no tienen cambios pendientes y sus últimas actualizaciones han sido guardadas en el repositorio gracias a los comandos git add y git commit.

**Archivos staged**

Son archivos en staging. Viven dentro de git y hay registro de ellos porque han sido afectados por el comando git add, aunque no sus últimos cambios. Git ya sabe de la existencia de estos últimos cambios, pero todavía no han sido guardados definitivamente en el repositorio porque falta ejecutar el comando git commit.

[Git stash: guarda cambios temporalmente](https://platzi.com/clases/1557-git-github/19984-stash/)

**Archivos unstaged**

Entiéndelos como archivos *“tracked* pero *unstaged”*. Son archivos que viven dentro de git pero no han sido afectados por el comando git add ni mucho menos por git commit. Git tiene un registro de estos archivos, pero está desactualizado, sus últimas versiones solo están guardadas en el disco duro.

**Archivos untracked**

Son archivos que NO viven dentro de git, solo en el disco duro. Nunca han sido afectados por git add, así que git no tiene registros de su existencia.

Recuerda que hay un caso muy raro donde los archivos tienen dos estados al mismo tiempo: staged y untracked. Esto pasa cuando guardas los cambios de un archivo en el área de staging (con el comando git add), pero antes de hacer commit para guardar los cambios en el repositorio haces nuevos cambios que todavía no han sido guardados en el área de staging.

**Comandos para mover archivos entre los estados de Git**

Estos son los comandos más importantes que debes conocer:

**Git status**

git status nos permite ver el estado de todos nuestros archivos y carpetas.

**Git add**

git add nos ayuda a mover archivos del untracked o unstaged al estado staged. Podemos usar git nombre-del-archivo-o-carpeta para añadir archivos y carpetas individuales o git add -A para mover todos los archivos de nuestro proyecto (tanto untrackeds como unstageds).

**Git reset HEAD**

Nos ayuda a sacar archivos del estado staged para devolverlos a su estado anterior. Si los archivos venían de unstaged, vuelven allí. Y lo mismo se venían de untracked.

**Git commit**

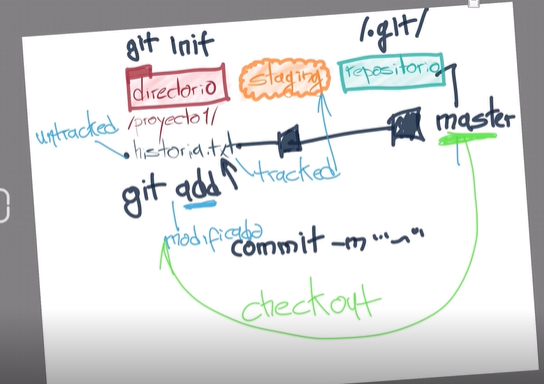
Nos ayuda a mover archivos de unstaged a tracked. Esta es una ocasión especial, los archivos han sido guardados o actualizados en el repositorio. Git nos pedirá que dejemos un mensaje para recordar los cambios que hicimos y podemos usar el argumento m para escribirlo (git commit -m "mensaje").

**Git rm**

Este comando necesita alguno de los siguientes argumentos para poder ejecutarse correctamente:

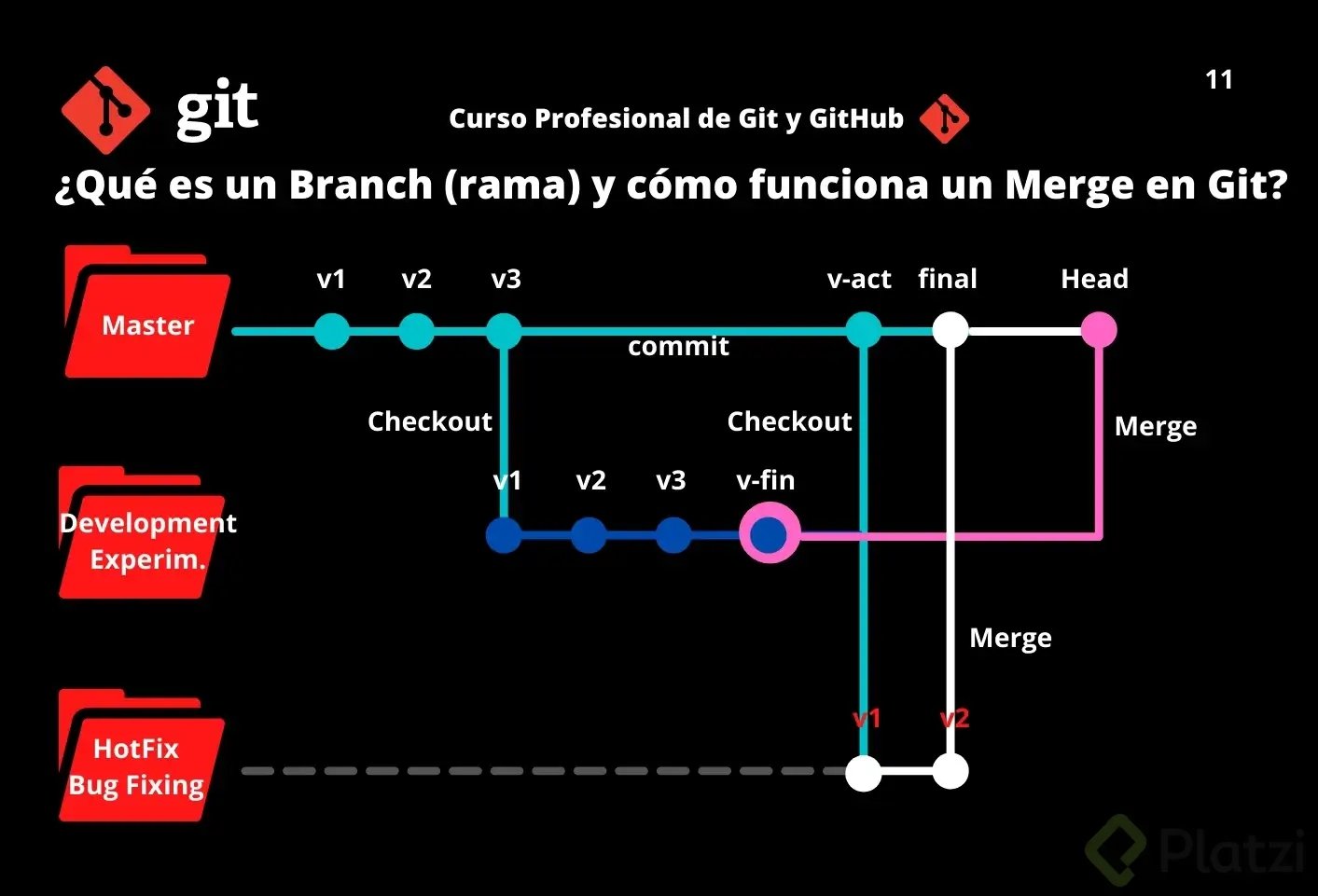
* git rm --cached: mueve los archivos que le indiquemos al estado untracked.
* git rm --force: elimina los archivos de git y del disco duro. Git guarda el registro de la existencia de los archivos, por lo que podremos recuperarlos si es necesario (pero debemos usar comandos más avanzados).

**Contribución creada con los aportes de:** *Diego Camacho, Jesús Sarabia y Angelo Yenque.*



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.

En otras palabras, un branch o rama en [Git](https://platzi.com/clases/1557-git-github/20215-que-es-git/) es una rama que proviene de otra. Imagina un árbol, que tiene una rama gruesa, y otra más fina, en la rama más gruesa tenemos los commits principales y en la rama fina tenemos otros commits que pueden ser de hotfix, devlopment entre otros.ㅤ



## Clases de branches o ramas en Git

Estas son las ramas base de un proyecto en Git:

### 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.

### 2. 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.

### 3. 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.

**Todos los commits se aplican sobre una rama**. Por defecto, siempre empezamos en la rama Main (pero puedes cambiarle el nombre si no te gusta) y generamos nuevas ramas, a partir de esta, para crear flujos de trabajo independientes.

## Cómo crear un branch o rama 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.

## Cómo hacer merge

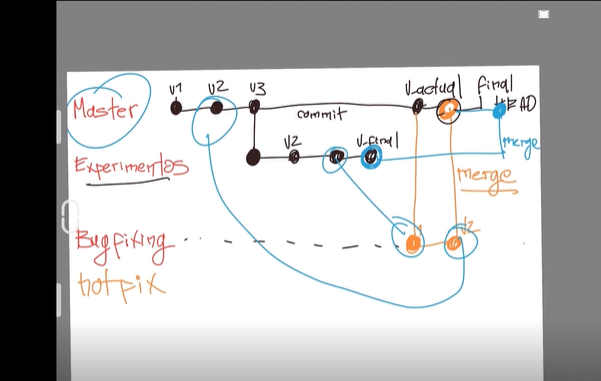
Producir una nueva rama se conoce como **Checkout**. Unir dos ramas lo conocemos como **Merge**.

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.

Descubre qué son los [git tags](https://platzi.com/clases/1557-git-github/19952-tags-y-versiones-en-git-y-github/)

Solo ten en cuenta que combinar estas ramas ([hacer “merge”](https://platzi.com/clases/1557-git-github/19939-funcion-de-ramas-con-git-mer-7/)) 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.



Los comandos git reset y git rm tienen utilidades muy diferentes, pero pueden confundirse fácilmente.

Git reset

El comando git reset es una herramienta poderosa que te permite deshacer o revertir cambios en tu repositorio de Git. Lo puedes ejecutar de tres maneras diferentes, con las líneas de comando --soft, --mixed y --hard.

Pero no como git checkout que nos deja ir, mirar, pasear y volver. Con git reset volvemos al pasado sin la posibilidad de volver al futuro. Borramos la historia y la debemos sobreescribir. No hay vuelta atrás.

Tres árboles en Git

Para entender lo anterior, recordemos que los “tres árboles” de Git son estructuras de datos basadas en nodos y punteros que Git utiliza para hacer seguimiento a un cronograma de ediciones, aunque no sean estructuras en forma de árbol en el sentido tradicional.

La mejor forma de entender estos mecanismos es creando un conjunto de cambios en un repositorio y siguiéndolos a través de los tres árboles. Averigüémoslo.

$ mkdir git\_reset\_test

$ cd git\_reset\_test/

$ git init .

Initialized empty Git repository in /git\_reset\_test/.git/

$ touch reset\_lifecycle\_file

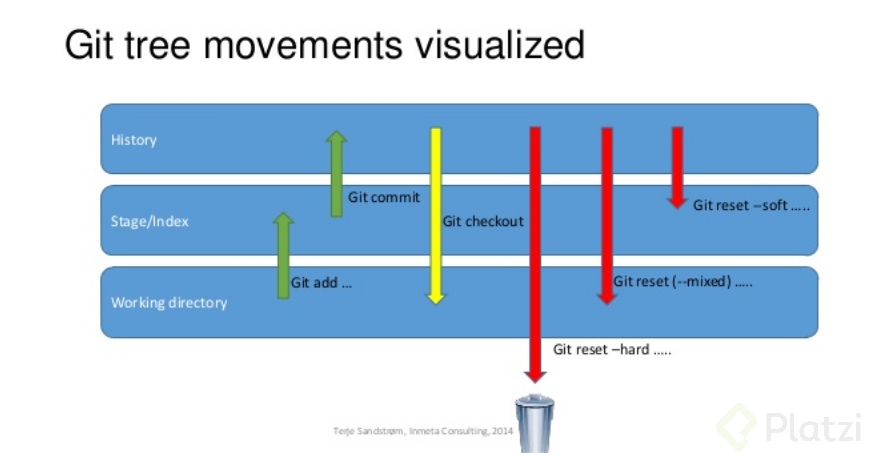
$ git add reset\_lifecycle\_file

$ git commit -m"initial commit"

[main (root-commit) d386d86] initial commit

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 reset\_lifecycle\_file



¿Cómo funciona Git Reset en tu flujo de trabajo?

git reset permite moverte entre diferentes commits para deshacer o rehacer cambios. Git guarda todos lo nuevo del repositorio como commits, que son instantáneas del estado del código en un momento dado y existen variaciones de este comando.

Variaciones de Git Reset

* git reset --soft: Borra el historial y los registros de Git de commits anteriores, pero guarda los cambios en Staging para aplicar las últimas actualizaciones a un nuevo commit.
* git reset --hard: Deshace todo, absolutamente todo. Toda la información de los commits y del área de staging se elimina del historial.
* git reset --mixed: Borra todo, exactamente todo. Toda la información de los commits y del área de staging se elimina del historial.
* git reset HEAD: El comando git reset saca archivos del área de staging sin borrarlos ni realizar otras acciones. Esto impide que los últimos cambios en estos archivos se envíen al último commit. Podemos incluirlos de nuevo en staging con git add si cambiamos de opinión.

Ten en cuenta que, si deshaces commits en un repositorio compartido en GitHub, estarás cambiando su historia y esto puede causar problemas de sincronización con otros colaboradores.

¿Qué es git reset HEAD?

git reset HEAD es un comando que te permite revertir los cambios que ya habías preparado para subir, y moverlos de vuelta a tu proyecto. Con este comando puedes cancelar los cambios que ya habías agregado, para que puedas revisarlos, modificarlos o deshacerlos antes de confirmarlos con un commit.

Git rm

Por otro lado, git rm es un comando que nos ayuda a eliminar archivos de Git sin eliminar su historial del sistema de versiones. Para recuperar el archivo eliminado, necesitamos retroceder en la historia del proyecto, recuperar el último commit y obtener la última confirmación antes de la eliminación del archivo.

Es importante tener en cuenta que git rm no puede usarse sin evaluarlo antes. Debemos usar uno de los flags siguientes para indicarle a Git cómo eliminar los archivos que ya no necesitamos en la última versión del proyecto.

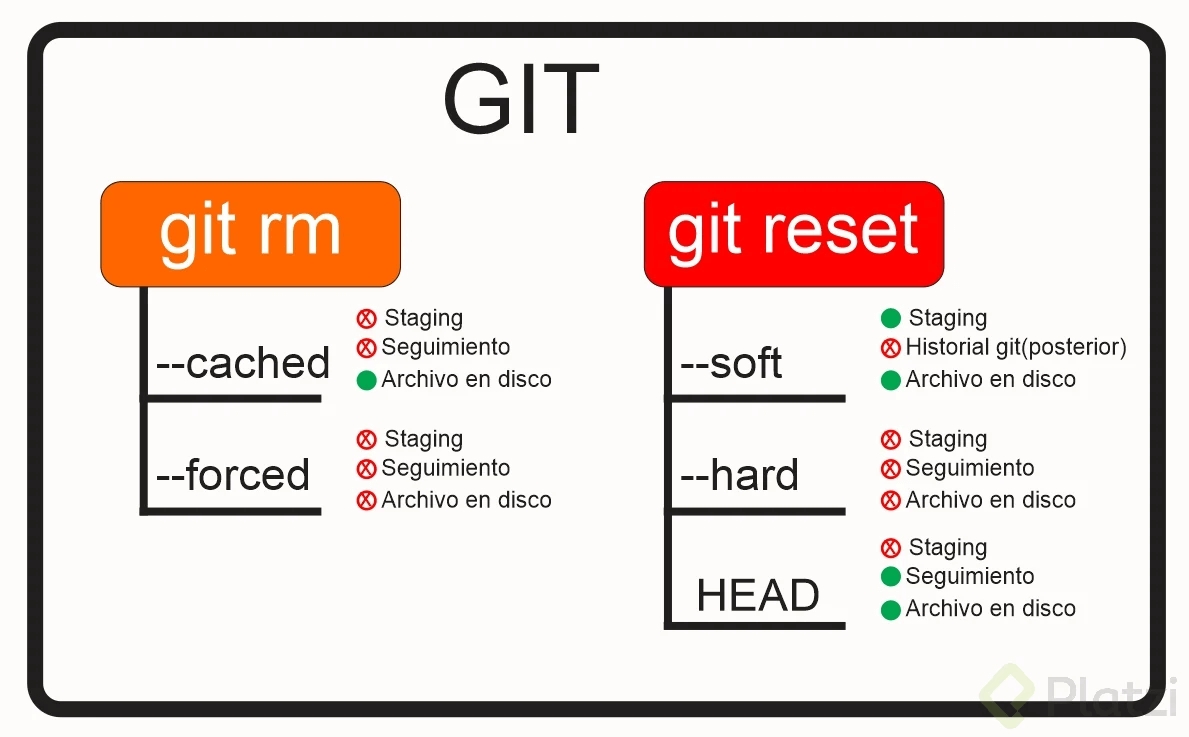
Variaciones de Git rm

* git rm --cached: Elimina archivos del repositorio local y del área de staging, pero los mantiene en el disco duro. Deja de trackear el historial de cambios de estos archivos, por lo que quedan en estado untracked.
* git rm --force: Elimina los archivos de Git y del disco duro. Git guarda todo, por lo que podemos recuperar archivos eliminados si es necesario (empleando comandos avanzados).

¡Al usar git rm lo que haremos será eliminar este archivo completamente de git!

¿Cuál es la diferencia entre git rm y git reset Head?

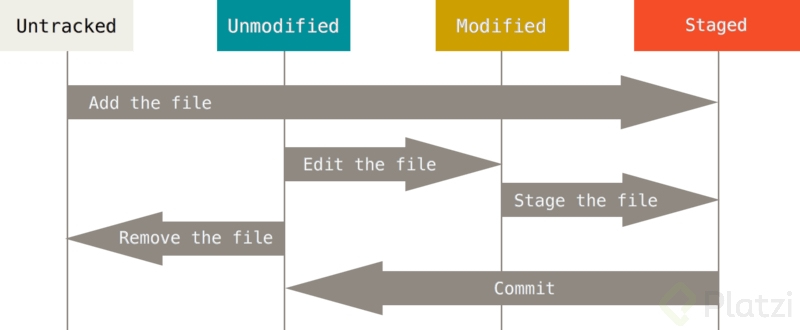
La diferencia principal entre git rm y git reset HEAD radica en que git rm elimina archivos del repositorio y de la historia del proyecto, mientras que git reset saca los cambios del área de preparación y los mueve del espacio de trabajo, sin afectar la historia del repositorio.



Es importante tener en cuenta el efecto que cada comando tiene en el proyecto y usarlos según tus necesidades y objetivos específicos.

¿Cuándo utilizar git reset en lugar de git revert?

Para reescribir la historia del repositorio y eliminar confirmaciones anteriores, se utiliza git reset. Para deshacer cambios de confirmaciones anteriores de forma segura sin modificar la historia del repositorio, se emplea git revert.



Resumen

Para evitar problemas en el trabajo, es valioso entender las implicaciones y riesgos de cada comando y elegir el enfoque adecuado según las necesidades y el flujo de trabajo del proyecto.

Con git rm eliminamos un archivo de Git, pero mantenemos su historial de cambios. Si no queremos borrar un archivo, sino dejarlo como está y actualizarlo después, no debemos usar este comando en este commit.

Empleando git reset HEAD, movemos los cambios de Staging a Unstaged, pero mantenemos el archivo en el repositorio con los últimos cambios en los que hicimos commit. Así, no perdemos nada relevante.

Siguientes pasos

Bueno, todos los cambios están en el área de Staging, incluido el archivo con los cambios que no están listos. Esto significa que debemos sacar ese archivo de Staging para poder hacer commit de todos los demás.