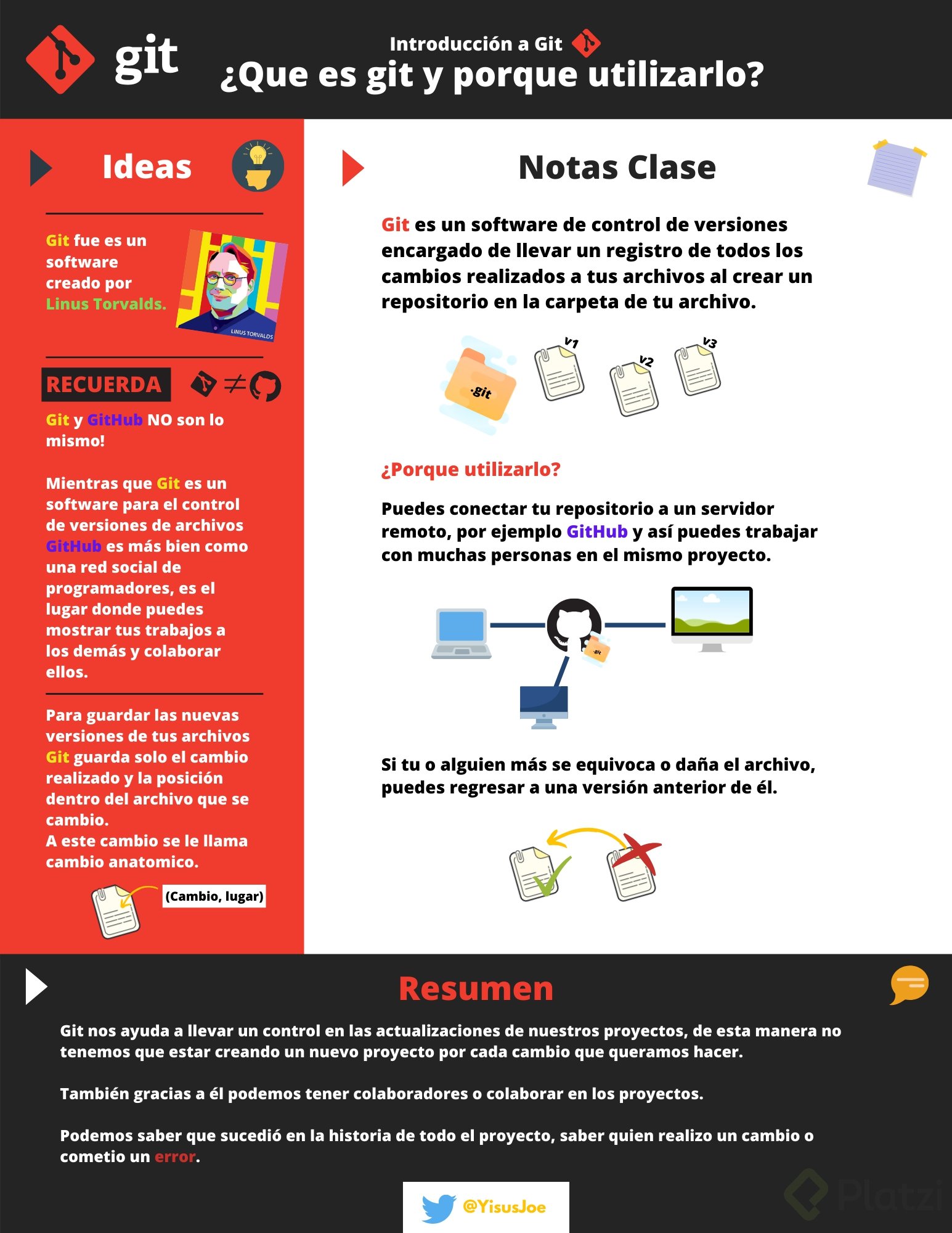
GIT: SCV (Sistema de control de Versiones o VCS Version control system)





Un [sistema de control de versiones como Git](https://platzi.com/clases/1557-git-github/20215-que-es-git/) nos ayuda a guardar el historial de cambios y crecimiento de los archivos de nuestro proyecto.

En realidad, los cambios y diferencias entre las versiones de nuestros proyectos pueden tener similitudes, algunas veces los cambios pueden ser solo una palabra o una parte específica de un archivo específico. Git está optimizado para guardar todos estos cambios de forma atómica e incremental, o sea, aplicando cambios sobre los últimos cambios, estos sobre los cambios anteriores y así hasta el inicio de nuestro proyecto.

“Guardar archivos binarios en el repositorio de Git no es una buena práctica, únicamente deberían guardarse archivos pequeños (como logos) que no sufran casi modificaciones durante la vida del proyecto. Los binarios deben guardarse en un CDN

* Git cuenta con 3 estados en los que es posible localizar archivos: Staged, Modified y Committed.
* El comando para iniciar nuestro repositorio, o sea, indicarle a Git que queremos usar su sistema de control de versiones en nuestro proyecto, es git init.
* El comando para que nuestro repositorio sepa de la existencia de un archivo o sus últimos cambios es git add. Este comando no almacena las actualizaciones de forma definitiva, únicamente las guarda en algo que conocemos como “Staging Area” (área de montaje o ensayo).
* El comando para almacenar definitivamente todos los cambios que por ahora viven en el staging area es git commit. También podemos guardar un mensaje para recordar muy bien qué cambios hicimos en este commit con el argumento -m "Mensaje del commit".
* Por último, si queremos mandar nuestros commits a un servidor remoto, un lugar donde todos podamos conectar nuestros proyectos, usamos el comando git push.

**Comandos básicos de git**

* git init: inicializa un repositorio de GIT en la carpeta donde se ejecute el comando.
* git add: añade los archivos especificados al área de preparación (staging).
* git commit -m “commit description”: confirma los archivos que se encuentran en el área de preparación y los agrega al repositorio.
* git commit -am “commit description”: añade al staging area y hace un commit mediante un solo comando. (No funciona con archivos nuevos)
* git status: ofrece una descripción del estado de los archivos (untracked, ready to commit, nothing to commit).
* git rm (. -r, filename) (–cached): remueve los archivos del index.
* git config --global user.email [tu@email.com](mailto:tu@email.com): configura un email.
* git config --global [user.name](http://user.name/) <Nombre como se verá en los commits>: configura un nombre.
* git config --list: lista las configuraciones.
* **git pull** -Agrega los cambios del commit más nuevo al repositorio local

**Analizar cambios en los archivos de un proyecto Git**

* git log: lista de manera descendente los commits realizados.
* git log --stat: además de listar los commits, muestra la cantidad de bytes añadidos y eliminados en cada uno de los archivos modificados.
* git log --all --graph --decorate --oneline: muestra de manera comprimida toda la historia del repositorio de manera gráfica y embellecida.
* git show filename: permite ver la historia de los cambios en un archivo.
* git diff <commit1> <commit2>: compara diferencias entre en cambios confirmados.

**Volver en el tiempo con branches y checkout**

* git reset <commit> --soft/hard: regresa al commit especificado, eliminando todos los cambios que se hicieron después de ese commit.
* git checkout <commit/branch> <filename>: permite regresar al estado en el cual se realizó un commit o branch especificado, pero no elimina lo que está en el staging area.
* git checkout – <filePath>: deshacer cambios en un archivo en estado modified (que ni fue agregado a staging)

**git rm y git reset**

**git rm:**

Este comando nos ayuda a eliminar archivos de Git sin eliminar su historial del sistema de versiones. Esto quiere decir que si necesitamos recuperar el archivo solo debemos “viajar en el tiempo” y recuperar el último commit antes de borrar el archivo en cuestión.

git rm no puede usarse por sí solo, así nomás. Se debe utilizar uno de los flags para indicar a Git cómo eliminar los archivos que ya no se necesitan en la última versión del proyecto:

* git rm --cached <archivo/s>: elimina los archivos del área de Staging y del próximo commit, pero los mantiene en nuestro disco duro.
* git rm --force <archivo/s>: elimina los archivos de Git y del disco duro. Git siempre guarda todo, por lo que podemos acceder al registro de la existencia de los archivos, de modo que podremos recuperarlos si es necesario (pero debemos aplicar comandos más avanzados).

**git reset**

Con git reset volvemos al pasado sin la posibilidad de volver al futuro. Borramos la historia y la debemos sobreescribir.

* git reset --soft: Vuelve el branch al estado del commit especificado, manteniendo los archivos en el directorio de trabajo y lo que haya en staging considerando todo como nuevos cambios. Así podemos aplicar las últimas actualizaciones a un nuevo commit.
* git reset --hard: Borra absolutamente todo. Toda la información de los commits y del área de staging se borra del historial.
* git reset HEAD: No borra los archivos ni sus modificaciones, solo los saca del área de staging, de forma que los últimos cambios de estos archivos no se envíen al último commit. Si se cambia de opinión se los puede incluir nuevamente con git add.

**Ramas o Branches en git**

Al crear una nueva rama se copia el último commit en esta nueva rama. Todos los cambios hechos en esta rama no se reflejarán en la rama master hasta que hagamos un merge.

* git branch <new branch>: crea una nueva rama.
* git checkout <branch name>: se mueve a la rama especificada.
* git merge <branch name>: fusiona la rama actual con la rama especificada y produce un nuevo commit de esta fusión.
* git branch: lista las ramas generadas.

Terminal y línea de comandos:

La línea de comandos nos permite interactuar con nuestro computador sin necesidad de utilizar una interfaz gráfica. Sin embargo, los computadores emplean distintos sistemas de archivos y manejan diferentes comandos, dependiendo del sistema operativo que utilicen.

**Comandos básicos en la terminal:**

* **pwd**: Nos muestra la ruta de carpetas en la que te encuentras ahora mismo.
* **mkdir**: Nos permite crear carpetas (por ejemplo, mkdir Carpeta-Importante).
* **touch**: Nos permite crear archivos (por ejemplo, touch archivo.txt).
* **rm**: Nos permite borrar un archivo o carpeta (por ejemplo, rm archivo.txt). Mucho cuidado con este comando, puedes borrar todo tu disco duro.
* **cat**: Ver el contenido de un archivo (por ejemplo, cat nombre-archivo.txt).
* **ls**: Nos permite cambiar ver los archivos de la carpeta donde estamos ahora mismo. Podemos usar uno o más argumentos para ver más información sobre estos archivos (los argumentos pueden ser -- + el nombre del argumento o - + una sola letra o shortcut por cada argumento).  
  - ls -a: Mostrar todos los archivos, incluso los ocultos.  
  - ls -l: Ver todos los archivos como una lista.
* **cd**: Nos permite navegar entre carpetas.  
  - cd /: Ir a la ruta principal:  
  - cd o cd ~: Ir a la ruta de tu usuario  
  - cd carpeta/subcarpeta: Navegar a una ruta dentro de la carpeta donde estamos ahora mismo.  
  - cd .. (cd + dos puntos): Regresar una carpeta hacia atrás.  
  - Si quieres referirte al directorio en el que te encuentras ahora mismo puedes usar cd . (cd + un punto).
* **history**: Ver los últimos comandos que ejecutamos y un número especial con el que podemos repetir su ejecución.
* **! + número**: Ejecutar algún comando con el número que nos muestra el comando history (por ejemplo, !72).
* **clear**: Para limpiar la terminal. También podemos usar los atajos de teclado Ctrl + L o Command + L.

Todos estos comandos tiene una función de autocompletado, o sea, puedes escribir la primera parte y presionar la tecla Tab para que la terminal nos muestre todas las posibles carpetas o comandos que podemos ejecutar. Si presionas la tecla Arriba puedes ver el último comando que ejecutamos.

Recuerda que podemos descubrir todos los argumentos de un comando con el argumento --help (por ejemplo, cat --help).

Crea tu primer repositorio y haz tu commit:

Le indicaremos a Git que queremos crear un nuevo repositorio para utilizar su sistema de control de versiones. Solo debemos posicionarnos en la carpeta raíz de nuestro proyecto y ejecutar el comando:



Recuerda que al ejecutar este comando (y de aquí en adelante) vamos a tener una nueva carpeta oculta llamada .git con toda la base de datos con cambios atómicos en nuestro proyecto.

Recuerda que Git está optimizado para trabajar en equipo, por lo tanto, debemos darle un poco de información sobre nosotros. No debemos hacerlo todas las veces que ejecutamos un comando, basta con ejecutar solo una sola vez los siguientes comandos con tu información:



Existen muchas otras configuraciones de Git que puedes encontrar ejecutando el comando git config --list (o solo git config para ver una explicación más detallada).

Si quieres ver los archivos ocultos de una carpeta puedes habilitar la opción de Vista > Mostrar u ocultar > Elementos ocultos (en Windows) o ejecutar el comando ls -a.

**Comandos para iniciar tu repositorio con Git**

* git init: para inicializar el repositorio git y el staged
* git add nombre\_del\_archivo.txt: enviar el archivo al staged
* git status: ver el estado, si se requiere agregar al starget o si se requiere commit
* git conf: para ver las posibles configuraciones
* git conf --list: para ver la lista de configuraciones hechas
* git conf --list --show-origin: para mostrar las configuraciones y sus rutas
* git rm --cached nombre\_del\_archivo.txt: para eliminar el archivo del staged(ram)
* git rm nombre\_del\_archivo.txt: para eliminar del repositorio

Si por algún motivo te equivocaste en el nombre o email que configuraste al principio, lo puedes modificar de la siguiente manera:  
git config --global --replace-all user.name “Aquí va tu nombre modificado”  
O si lo deseas eliminar y añadir uno nuevo  
git config --global --unset-all user.name :Elimina el nombre del usuario  
git config --global --add user.name “Aquí va tu nombre”



Para encontrar los archivos, y para solucionar problemas avanzados.

El comando **git show** nos muestra los cambios que han existido sobre un archivo y es muy útil para detectar cuándo se produjeron ciertos cambios, qué se rompió y cómo lo podemos solucionar. Pero podemos ser más detallados.

Si queremos ver la diferencia entre una versión y otra, no necesariamente todos los cambios desde la creación del archivo, podemos usar el comando **git diff commitA commitB**.

## Comandos para analizar cambios en GIT

* **git log nombre\_de\_archivos.extensión**: histórico de cambios con detalles
* **git push**: envía a otro repositorio remoto lo que estamos haciendo
* **git pull**: traer repositorio remoto
* **ls**: listado de carpetas en donde me encuentro. Es decir, como emplear dir en windows.
* **pwd**: ubicación actual
* **mkdir**: make directory nueva carpeta
* **touch archivo.extensión**: crear archivo vacío
* **cat archivo.extensión**: muestra el contenido del archivo
* **history**: historial de comandos utilizados durante esa sesión
* **rm archivo.extensión**: Eliminación de archivo
* **comando --help**: ayuda sobre el comando
* **git checkout**: traer cambios realizados
* **git rm --cached archivo.extensión**: se utiliza para devolver el archivo que se tiene en ram. Cuando escribimos git add, lo devuelve a estado natural mientras está en *staging*.
* **git config --list**: muestra la lista de configuración de git
* **git config --list --show-origin**: rutas de acceso a la configuración de git
* **git log archivo.extensión**: muestra la historia del archivo



Cuando no añadimos un commit, nos abre un editor de código llamado VIM.

Para comenzar a escribir texto escribimos ESC and I.

Añadimos nuestro comentario sin #

Esc + shift + ZZ (Con esto guarda en VIM, por lo que fuerza el envio)

PARA CUANDO NO ENVIO EL COMMIT, Y ME OBLIGA A HACERLO!!!

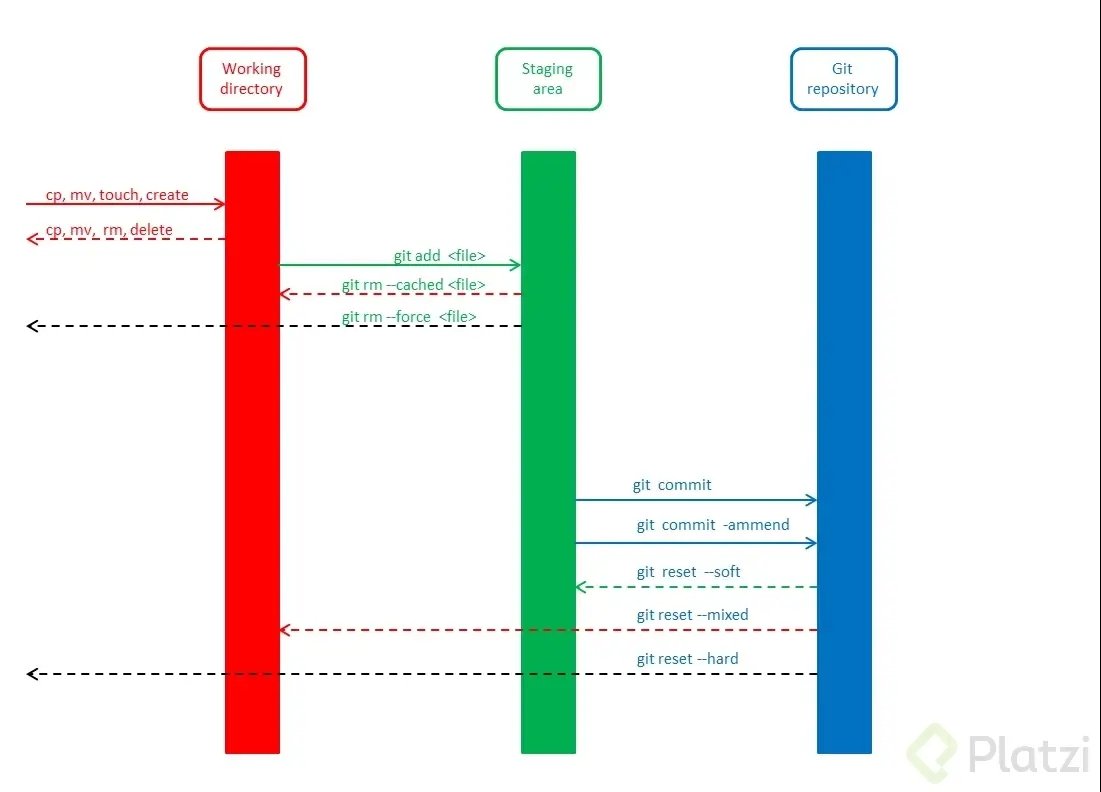
Que es Staging:

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 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.



## ****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.

### ****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.

### G****it reset HEAD****

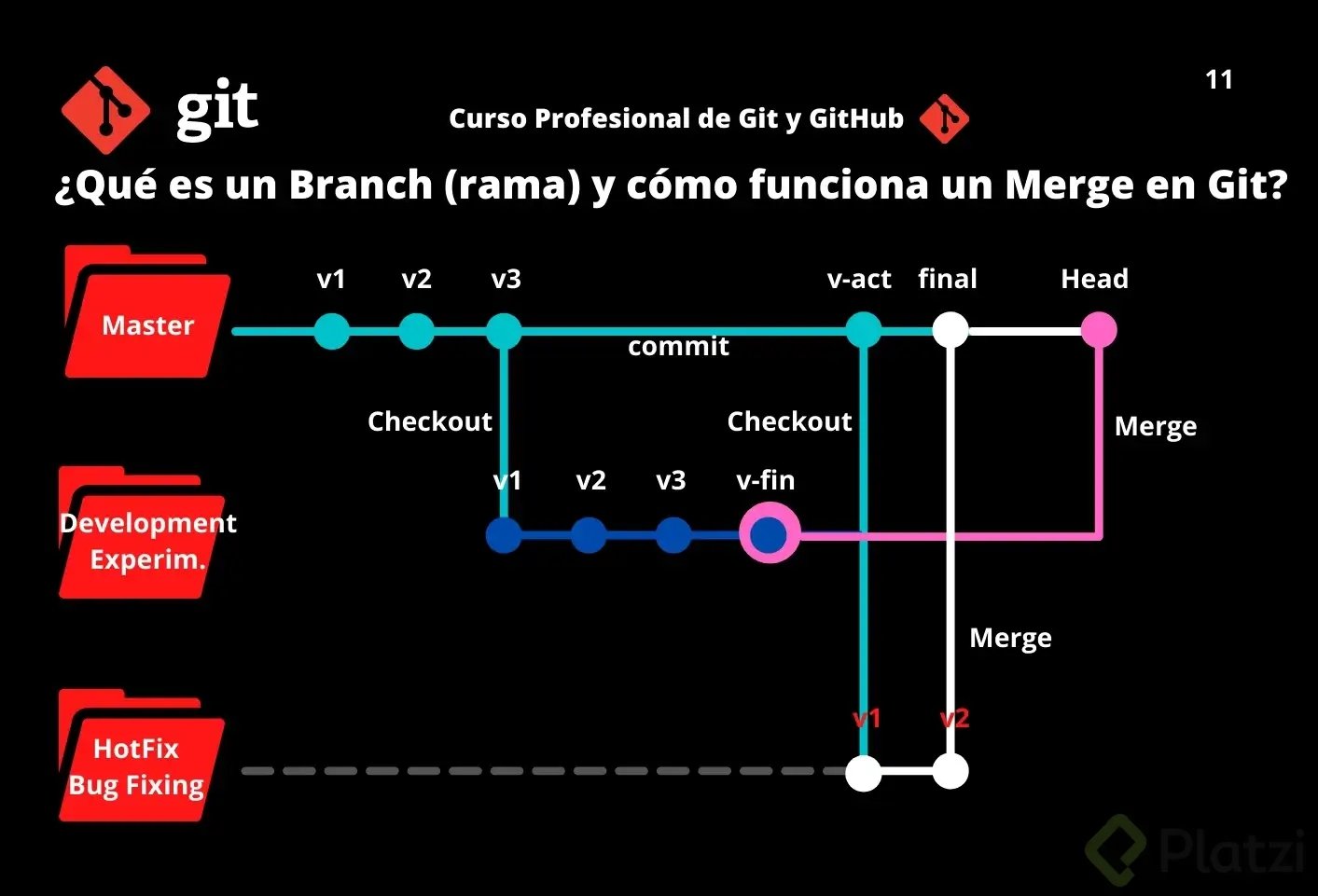
Nos ayuda a sacar archivos del estado staged para devolverlos a su estado anterior. Si los archivos venían de unstaged, vuelven allí. Y lo mismo se venían de untracked.

**Git 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).

¿Qué es branch (rama) y cómo funciona un Merge en Git?



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 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.

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.

DATO CURIOSO: De Master a Main.

El movimiento de #BlackLivesMatter ha ayudado a que GitHub sustituya algunas palabras usadas en su plataforma con relación al racismo.

Palabras como master, whitelist, blacklist y slave se encuentran en este proceso de cambio. Pero el más importante en ese momento y que ya ha empezado a tener efecto es que la rama **master** ahora se llamará **main**.

## Master a Main

A partir del primero de octubre todos los repositorios **nuevos** que se creen empezarán a mostrar main como rama principal y tendrán que hacer sus comandos respectivos allí.

Los repositorios existentes seguirán usando **master** como rama principal, porque hacer el cambio de nombre en este momento puede generar muchos conflictos para esos proyectos. [Microsoft promete que para final de año](https://github.com/github/renaming), este cambio sea más fácil de aplicar para proyectos existentes.

Es esperable que estos mismos cambios sean aplicados por otras compañías como Apple y LinkedIn. Creo que palabras como “blacklist” y “whitelist” son muy utilizados en la industria tecnológica, no es mal momento para empezar a cambiar dichos términos por otros.

El flujo de Gitflow es así:

En la rama **master** tendremos solo lo que se ha liberado.

1.- Se crea la rama **develop**, es la rama en la que estamos trabajando (lo que vamos a liberar).

2.- Liberar a producción con tu equipo de trabajo se crea una **release** desde develop.

No se pasa directo de develop a master, Git Flow crea la nueva rama de release.

3.- Por cada petición o tarea se genera una rama llamada **feature** a partir de develop.

4.- Por ejemplo una pantalla nueva, se crea y está completa el **feature** de pantalla se cierra y se afusiona con **develop**.

5.- Cuando tienes la rama release terminada, fusionas con develop y master.

6.- Si hay problema en master se crea **hotfix** que son los cambios sobre algo que está en producción.

7.- Se crea una nueva rama se trabaja y se reintegra. Una vez que hotfix se completa, se fusiona a ambos develop y master.