***Profundización en GIT/GITHUB***

Cuando uno inicializa GIT (con el comando git init), suceden dos cosas especialmente:

1. Se crea un area en memoria RAM que se llama *“Staging”,* justamente por estar situada en la memoria RAM a esta area también se le conoce como: “area temporal”. Básicamente, es un area completamente desconectada que funciona en memoria RAM y es donde, al principio, vas a ir agregando los cambios que vayas haciendo   
(de tu carpeta), cambios que todavía no han sido guardados en una copia de seguridad (es decir, que todavía no están en repositorio local); importante esto, tan sólo se les está haciendo un seguimiento en un area *de paso*. Todo cambio registrado que quiera hacerle un seguimiento en el area temporal (staging area) debe hacerlo con el comando: **“git** **add” (+ fichero).** Y si, por el contrario, quiere dejar de rastrearlo o de seguirlo (al fichero o carpeta modificada en cuestión), situandolo en su punto de partida de nuevo (ya sea porque se equivocó en alguna modificación y quiere volver a manipular algunas cosas); usted puede ejecutar el siguiente comando: **“git** **rm** **--cached” (+ fichero).** De nuevo, su directorio o fichero en cuestión, vuelve a ser ***untracked***(deja de ser rastreado).

2. Ya lo introducimos antes, se logra la creación de un repositorio local de la carpeta actual (local repo) que recibe por nombre: **“.git”,** una carpeta oculta que emula a la carpeta original y que sirve para guardar, potencialmente, copias de seguridad **(commits)** de todos los cambios o ediciones que estemos haciendo dentro de nuestra carpeta en cuestión. Para tener en cuenta, antes de recurrir a crear una copia de seguridad de algún cambio o edición que haya hecho... debe primero hacerle un seguimiento; es decir, llevarlo a “area temporal”; y, eventualmente, luego querrá llevar esos cambios a repositorio local (ahora sí, al crearles copias de seguridad).

Es preciso decir que, depende de nosotros qué cambios almacenar de forma definitiva en nuestro disco duro por medio de copias de seguridad llamadas “commits”... en dicha carpeta oculta llamada “.git”: nuestro repositorio local. En repositorio local no habrá nada mientras no se haya decidido qué cosas guardar en copias de seguridad.

Justamente, el comando que se encarga de crear esas copias de seguridad se ejecuta así: **“git** **commit** **-m** **‘’”.** La *-m* podría ser considerada como un *elemento* (una extensión del comando “git commit”) que nos permite darle una referencia identificativa y explicita, por medio de un comentario o mensaje, al cambio que estamos almacenando en una copia de seguridad: básicamente nos recuerda qué fue lo que puntualmente modificamos de nuestra anterior versión. **nota:** Si se le da por no poner un mensaje *(-m),* Git lo conducirá a un editor de texto “vi” para que lo haga en la primera línea de texto. Debajo, por medio de comentarios que el compilador ignora, Git le explica que es una buena práctica hacerlo... *así que simplemente hágalo, se lo agradecerá.*

Al ejecutar “git commit” sobre nuestros cambios, éstos pasan de estar seguidos/rastreados ***(tracked)*** por un “area temporal” para llegar a un “repositorio local”. Todo “commit” generado recibe un número de identificación, se llama: **identificador** (un nombre interno de la base de datos de Git para cada cambio), y básicamente con él es que podemos hacer todo tipo de manipulación sobre los “commits”. Al último “commit” o copia de seguridad, nuestra última versión, se le llama también: **“Head”.**

Entonces, tenemos que todas las copias de seguridad que hagamos se almacenarán en repositorio local (y sólo eso). El repositorio local recibe un nombre por defecto de: **“Master”.** *Master* realmente es una rama, la rama principal de nuestro repositorio. Eventualmente necesitaría crear varias cuando esté trabajando un proyecto grande o cuando esté trabajando un proyecto en equipo.

Por ejemplo, con **Git Branch** es posible crear diferentes ramas de desarrollo que pueden converger en el mismo repositorio. Podríamos hablar de: una rama *estable,* una rama *experimental* ***“development”*** *(por ejemplo, para probar nuevo código, fuera de lo convencional que te pide o exige el proyecto, que podría ayudarte a encontrar “por accidente” algo mejor que lo convencional)* y una rama *inestable*, en las industrias a un tipo de ellas se les conoce como:***“bugfixing”*** *o* ***“hotfix”*,** que sirven para arreglar un bug de la versión actual de nuestra rama principal, sin estropear en absoluto todo el trabajo que ya se lleva hecho en ella (para cuidar el trabajo que ya tenemos hecho en código precisamente); entonces, todos los errores a los que nos expongamos, posiblemente, en el intento de resolución de nuestro bug... que se queden y se den, con toda tranquilidad, en la rama experimental. Cuando ya por fin hayamos podido encontrar, en dicha rama experimental, la resolución de problema a nuestro bug, luego de desordenarlo todo sin mosquearnos por eso (no hay nada de qué preocuparse, para eso se creo esta rama); bueno, ahí sí podrá unir la rama experimental, su última versión, a la versión última de la rama *master.* Ya entre poco dejaremos claro cuál es el comando qué se encarga de unir las ramas.

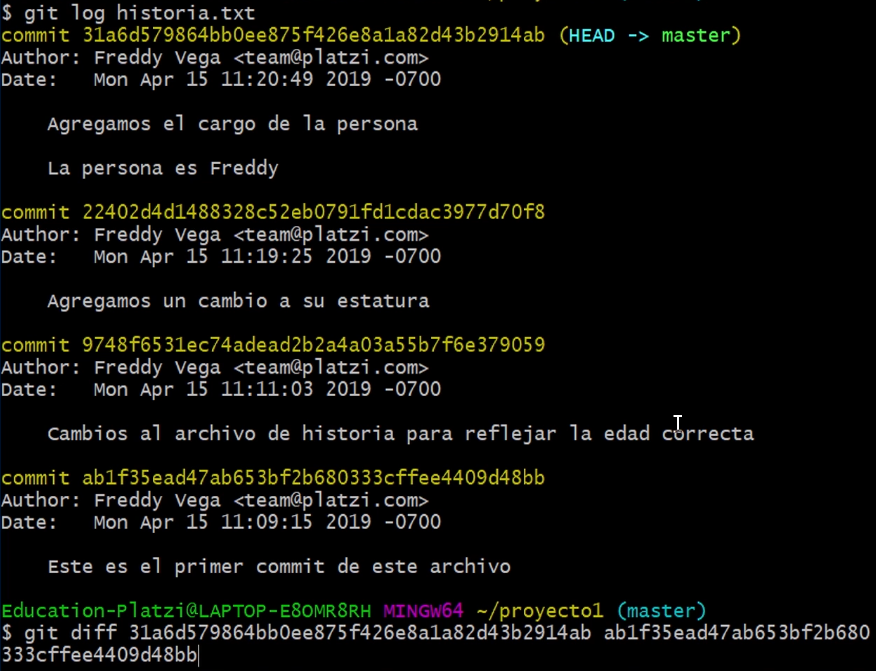
Caso aparte, si está trabajando en equipo, cada integrante del mismo puede trabajar su parte en ramas diferentes; y, al final, cuando cada integrante ya tenga la edición definitiva de su parte (de cada sub-rama derivada de la rama master); entonces, ahí puede unir todas las sub-ramas a la rama principal *master,* logrando así que todos los avances definitivos de cada compañero de trabajo estén unidos a un proyecto final. En general, esta unión entre los avances de las sub-ramas y la rama *master* se hace por medio de un comando que se llama: **“git** **merge”.**

tip: Si por ejemplo me gustaría ver la totalidad de los cambios (en términos de copias de seguridad o “commits”, solamente. Sin irse a más detalles) que he hecho sobre una carpeta, o fichero en especifico; podríamos ejecutar el siguiente comando: **“git** **log” (+ fichero).** Hay podríamos observar, por ejemplo, cuántos “commits” o copias de seguridadad le he creado a una carpeta o a un fichero en especifico.

Ahora, si deseo comparar explicitamente (al detalle) los cambios más recientes que he hecho en mí última versión de copia de seguridad (head), de un fichero o archivo, en comparación a la versión anterior a esa última, idealmente se ejecuta el comando: **“git show”.** Si estamos ante un fichero, por ejemplo, git nos mostrará al detalle, letra por letra, donde hubieron cambios en nuestro último commit en comparación al anterior a él. Si el fichero es un archivo de texto, por ejemplo, git señala a color rojo y con un signo menos (-) lo que hemos eliminado en nuestro último commit cargado en comparación al penultimo (del mismo archivo). Algo parecido ocurre cuando, en vez de haber eliminado algo, lo hemos adicionado: git señala en color verde y con un signo más (+) todas nuestas adicciones. Y, lo que no sufrió de alteraciones, lo deja simplemente en color gris.

Resulta muy importante la utilidad de este comando si queremos averiguar, por ejemplo, qué pudo salir mal a la hora de revisar las modificaciones de un archivo en comparación a su anterior versión aún no modificada. Si estamos ante un archivo donde tenemos, por ejemplo, nuestro código de programación; y resulta que, a partir de nuestra última actualización, el programa dejó de correr correctamente cuando fue compilado (o peor aún, no pudo ser compilado en la última actualización): sería de mucha utilidad, entonces, revisar línea por línea de código dondé se registraron los cambios o modificaciones y enfocar toda nuestra atención hacia ellos para ver dónde potencialmente está nuestro error. **El comando git show sería de mucha ayuda para eso.**

**Aclaración:** si desea trabajar con este comando, eventualmente será necesario que sus archivos sean creados y editados desde la propia consola. De lo contrario, se expone a que la consola (o el comando) no interprete de manera correcta el archivo en cuestión que desea analizar.

Ahora, si por ejemplo deseo comparar a fondo, con detalles, entre dos versiones o “commits” de un mismo archivo, sin importar qué versiones comparar en todo su historico de “commits”, se hace por medio del comando: **“ git diff” y los identificadores de los commits a comparar de mi interés.** Tal que así, por ejemplo:

Fijese en la última parte de la imagen, antes de ejecutar el comando **“git diff”**. Observe que, luego de alistar todo el historico de “commits” del archivo de texto “historia.txt” con el comando “git log”; nos interesa ahora saber, entonces, todos los cambios que hubieron o hay (por ejemplo) solamente comparando la primera versión del archivo (su primera copia de seguridad) con la última versión del mismo (su última copia de seguridad); es decir, **y ojo con esto**, no se verán reflejados (o se mantendrán estaticos, en color gris) los cambios que se hicieron en otras versiones (en las versiones que hay entre la primera y la última, por ej.). Tal que así:

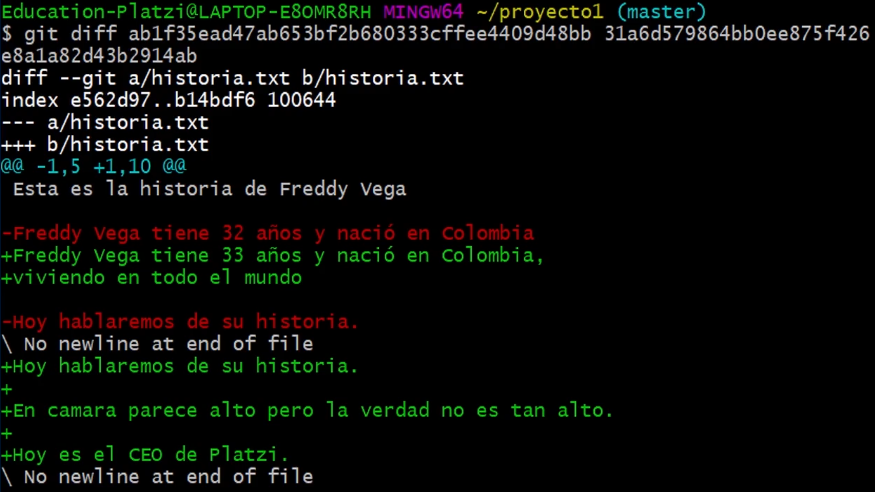


**Consejo:** Después de “git diff” ponga siempre de primero la versión del archivo más “vieja” y de último la versión del archivo más reciente. Esto para que entienda el comando bien qué fue lo que se agregó y que se suprimió realmente en una línea de tiempo que va de un tiempo pasado a un tiempo presente, de nuestra versión “vieja” a una más reciente. (y no invierta dicha información).

Por lo anterior, si se fija bien, nos toca intercambiar las posiciones de los identificadores en nuestro ejemplo y situarlos así (versión más vieja a comparar y luego la versión más reciente a comparar):



Todo el listado de cambios registrados que hubieron entre las dos versiones del archivo se deberían ver más o menos así:



***Es la misma lógica que explicamos con “git show” para entender qué se agregó y que se suprimió.***

Ahora, hay un comando adicional que nos permite viajar entre commits y es el siguiente: “**git reset + identificador**”. Por ejemplo, si resulta que actualmente estamos trabajando con una última copia de seguridad guardada: “48c2e46”, y queremos irnos de nuevo a una copia de seguridad anterior: “2f15ade”, debemos ejecutar el siguiente comando, tal que así: “git reset 2f15ade”. Y listo! Apenas presionen *enter* en pantalla se verá reflejado, de nuevo, sólamente lo que teníamos registrado en el identificador “2f15ade” y no en el identificador “48c2e46” **(anulando así todas las copias de seguridad situadas por encima del identificador “2f15ade”); es decir, se verá ahora el archivo “2f15ade”, lo que equivale a decir: antes de los cambios del archivo “48c2e46”.** Esto nos debe hacer reflexionar muy a fondo cada vez que queramos proceder de esta manera con git, **especialmente cuando estamos trabajando de forma compartida nuestro repositorio, pues estamos ante un comando destructivo.**

**Por qué? en el fondo el programa que está haciendo?** (y es en esto en lo que tiene que tener cuidado), **en el fondo el comando “reset” lo que está haciendo realmente es eliminar, de forma permanente mientras se lo permita, todos los “commits” que habían sido creados justo después del commit “2f15ade”.**

*Git reset*, por defecto, lo que hace es, en resumidas cuentas, eliminar las copias de seguridad (“commits”) y, además, situarte en una copia de seguridad “vieja”. (la que has seleccionado y ha removido todas las copias de seguridad posteriores a ella, ese “commit” elegido con “git reset”): esto sin dejar rastro alguno en el “staging area” (area temporal); entonces, si por ejemplo, desea crear una nueva copia de seguridad a partir de los cambios que venía trabajando (los cambios que ya había hecho inmediatamente después de la versión “vieja” seleccionada y que ya estaba en “commit”); debe primero hacerles seguimiento (con git add) a dishcos cambios (de nuevo); y, posteriormente, ejecutar ahora así el comando “git commit” para guardar una nueva copia de seguridad inalterable (básicamente para volver a guardar una nueva copia de seguridad, de los cambios que ya había hecho, de la versión que venía inmediatamente después al commit “2f15ade”).

Sin embargo, git reset, también hace uso de funciones más especificas, se compone por el uso de *elementos* propios como: “hard” y “soft”. Se ejecutan así: **“git reset --hard + identificador”** y **“git reset --soft + identificador”**. La diferencia entre una y otra, respectivamente, consiste en que? Ya lo veremos.

**“hard”:** elimina el “commit” y, además, elimina de forma totalitaria todos los cambios que se habían hecho en él; es decir, no es que deje los cambios por fuera del stage (o area temporal) como sí lo hace por defecto “git reset” (pero al menos con *git reset* tiene los cambios ahí aún, sólo que le toca hacerle seguimientos con “git add” si desea recuperarlos total o parcialmente); en cambio, con “git --hard reset” simplemente los cambios se quedan por fuera de todo (ya no existen), se quedan por fuera de la inicialización propia del git, de su repositorio.

Ahora, **“soft”** es mucho más suave en su tratamiento efectivamente: también elimina el “commit”; pero, esta vez, los cambios no sólo se quedan en la inicialización del repositorio (como lo hace “git reset” por defecto); sino que también mantienen su seguimiento; es decir, los cambios del “commit” recientemente eliminado se mantienen dentro del staging (area temporal), no los borra. Y, no le toca, por ejemplo, ejecutar el comando “git add” porque los cambios ya están ahí (esperando por ser subidos nuevamente en una nueva copia de seguridad inalterable, en un nuevo *commit* por ejemplo). En resumidas cuentas, “soft” sólo deshace las operaciones del “git commit”, pero sin eliminar las operaciones del “git add”.

Existe otro comando para desplazarse entre los “commits” relacionados a un archivo en cuestión (para ir a cualquiera de sus versiones ya guardadas en copias de seguridad), el comando: **“git checkout + identificador + fichero”.** Pero, es igual de peligroso a *reset.* Ahora, si deseamos regresar a nuestra última copia de seguridad de dicho archivo, con este comando, sería así: **“git checkout + master + fichero”.**