***Tutorial de GIT/GITHUB***

***Tutorial de GIT***

Git es un software de control de versiones que sirve de herramienta para guardar o registrar, en una especie de base o nube, todas las ediciones de nuestros proyectos de codificación realizados; digamos, para que ningún cambio, en un proyecto, se nos pierda en el transcurso de su realización; sirve en caso que queramos retroalimentar ciertos detalles que hicimos en el pasado o modificar ciertas líneas de códigos ya ejecutadas en el tiempo y en diferentes versiones. Adicionalmente coordina el trabajo que varias personas realizan sobre archivos compartidos.

Funciona más o menos así: hay un elemento de interés y son: las carpetas, las carpetas que usamos para guardar nuestros proyectos de código; esa carpeta en Git se le conoce como: “working directory”. Ya iniciado Git; es decir, ya estando la carpeta interpretada como un “working directory”. Working Directory lo que hará ahora es mandar ciertos archivos contenidos en él (o todos, si desea) a un area. La función del “area temporal” (staging area) es la de guardar los cambios que nosotros realicemos en los directorios (en los archivos seleccionados del directorio); es decir, guardar los cambios ejecutados de las carpetas que, ahora son llamadas en función de Git, como: “working directory”. Y si, por ejemplo, ya estando en el “area temporal” (staging area) queremos guardar de forma definitiva esos cambios que hemos ejecutado, se hace por medio de lo que se conoce como: “repositorio local” (local repo).; es decir, de forma oculta, Git va a tener un repositorio con todas nuestras versiones de cambio (las registradas en él). En práctica se podrá entender mejor. Veamos:

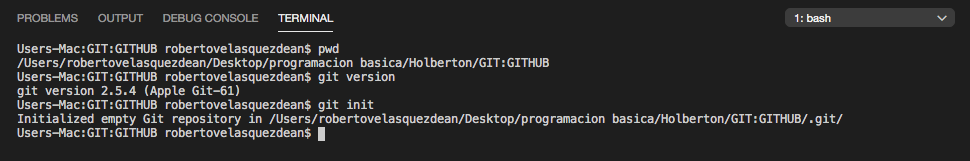
Muy avanzadamente en el Visual Studio Code puede abrir directamente, desde allí, por medio de “View” la terminal de Mac. Antes de eso, arraste un directorio (carpeta), la cual pasaría a hacer la carpeta con la que usted va a trabajar para hacerle seguimiento a los archivos de su selección en “staging area” y/o “local repo”. Es decir, es desde esta carpeta arrastrada “working directory” donde se seleccionará todos los archivos, o algunos archivos en cuestión, que desee almacenar en el repositorio local. Importante: en la terminal, por medio del comando “pwd”, debe fijarse que se encuentre en la ruta o dirección donde esté ubicado su carpeta de interés “working directory”, la que desea almacenar eventualmente en un repositorio.

Aclaración: Un repositorio, en teoria, no es más que un almacen o lugar donde se guardan ciertas cosas.

1. Ya instalado el Git y verificado que se encuentre en la ruta correcta. Ejecute: “git version”.

2. Ahora, lo que habíamos planteado, los archivos en cuestión de nuestro “working directory” (todos, o para ser más precisos los que seleccionamos) ... para relacionarlos con un repositorio de caracter oculto (creación de carpeta oculta. git), hay que ejecutar el siguiente comando (una sola vez por carpeta): “git init”. Aclaremos algo, si desea tener la posibilidad de guardar toda la carpeta en sí o parcialmente en un repositorio; es decir, incluyendo o no todos sus archivos (los cuales se irán respaldando de forma oculta) debe arrastrar directamente la carpeta en el Visual Studio Code y ejecutar este comando ya estando en la terminal.

Un ejemplo de cómo se ve nuestra carpeta de interés “GIT:GITHUB” al ya estar a disposición de un repositorio GIT (no lo está todavía, ojo, pero ya tiene la posibilidad de estarlo!). Acá los pasos:

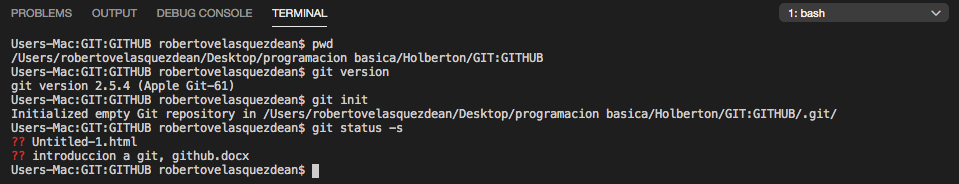


Por lo anterior, todos los archivos contenidos dentro de la carpeta GIT:GITHUB *potencialmente* serán almacenados en un repositorio local (todavía no lo están y pueda que no le interese que todos estén).

Ahora bien, existe también otro tipo de repositorio GIT, pero es de caracter remoto. Existen varios tipos de ellos, el más popular a nivel mundial es: GITHUB (empresa). Es decir, con GIT podríamos tener de forma almacenada todos nuestros archivos en una carpeta oculta “GIT” en nuestro computador (local repo); pero, con un “GIT REMOTO”, no los tenemos en nuestro computador; sino, en una nube o en un servidor externo (remote repo).

Continuemos: en caso que deseemos visualizar que archivos no tienen seguimiento (es decir, verificar cuáles archivos realmente tienen la posibilidad de estar almacenados en un repositorio local... o antes de un repositorio local, en una area temporal) podemos saberlo ejecutando el siguiente comando: “git status -s” y se nos alistan dichos archivos.

Verifiquemos lo anterior con nuestra carpeta de modelación GIT:GITHUB.

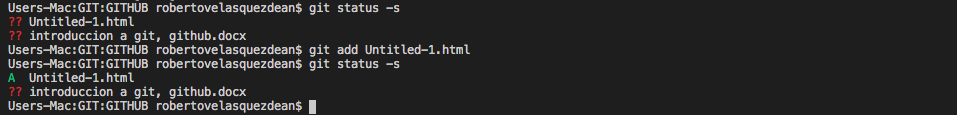


Se dará cuenta que, justamente los archivos que tienen la posibilidad de ser enviados a un repositorio GIT, son los mismos contenidos en nuestra carpeta GIT:GITHUB. (son los que se identifican con dos signos de interrogación color rojo, es así porque realmente no se ha denifido todavía un destino a esos archivos). Es decir, son los archivos que potencialmente deseamos guardar “sacarles una fotografia” para que, a futuro, si hacemos algún tipo de cambio o modificación parcial sobre ellos, podamos volver a “esa última fotografía tomada” si así lo deseamos (a la última o a varias “fotografias” que tomamos en el tiempo). Esa es la gran gracia de GIT para no vernos en la tediosa tarea de crear distintas ediciones por aparte de un mismo archivo (por ejemplo: index.html, index1.html, index2.html, indexfinal.html, indexfinalfinal.html, etc) NO, ya no será necesario hacer eso.

Ahora bien, para hacerle un seguimiento ahora sí a los archivos... nosotros tenemos que enviarlos, ahora sí, primeramente, a una area temporal “staging area”; eso se hace con el comando: “git add”. Acá viene una particularidad: ya con el comando “git add” podemos especificar si queremos que todo el “workind directory” pase a una area temporal (staging area) o sólo unos cuantos archivos de dicho direcotorio.

Las especificaciones se hacen de la siguiente forma, por ejemplo: si deseo solamente hacerle seguimiento al archivo “Untitled-1.html” guardando una copia del mismo en el area temporal (staging area), y no a todo el directorio llamado: “GIT:GITHUB”, tendríamos que ejecutar el siguiente comando luego de “git status -s”: “git add Untitled-1.html”. (si no les aparece ningún mensaje en la consola eso quiere decir que GIT está trabajando de forma correcta, tal como verá).

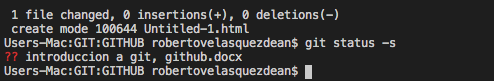
Ahora bien, podemos querer comprobar de igual manera que realmente sí pudimos guardar en el area temporal (staging area) dicho archivo (Untitled-1.html), pues bien, ejecutemos de nuevo el comando: “git status -s”. Verá lo siguiente:



Observe que, en un inicio, ambos archivos de la carpeta “GIT:GITHUB” no tenían ningún tipo de seguimiento (ambos: ??); pero, luego de ejecutar el comando “git add Untitled-1.html” y posteriormente revisando el estado del directorio (al ejecutar el comando “git status -s”), nos percatamos que, ahora, el archivo “Untitled-1.html” tiene una “A” mayuscula color verde en su inicio. Qué significa esto? Pues que el archivo “Untitled-1.html” ya tiene un seguimiento, una copia, en el area temporal (staging area), pero que es editable igual y está abierto a cambios y monitoreos. (su copia de seguridad es editable).

Pues bien, si en caso tal deseamos, de forma definitiva, por fin guardar una copia de seguridad inalterable en el tiempo (la foto como tal); es decir, mandarlos ya al repositorio local (local repo) lo podemos hacer por medio de un comando que se ejecuta tal que así: “ git commit -m “aquí insertar comentario entre comillas” ”. Así

“git commit -m (+ comentario)” tomará todos sus archivos previamente instalados en el area temporal (staging area) y los almacenerá en el repositorio local (local repo), se ve algo más o menos así:



*Nota: “ -m” es la simbologia que permite poner comentarios dentro de GIT:GITHUB. Siempre trate de poner un comentario en el comando “git commit”; pues, le dará un dolor de cabeza si no lo hace.*

*dato curioso: los archivos ya situados en el repositorio local también son mal llamados: “commit”.*

*Como ya se habría dado cuenta en la anterior imagen, si nuevamente queremos revisar el estado del directorio con el comando “git status -s” nos daremos cuenta que ya no saldrá nuestro archivo* “Untitled-1.html” alistado; ni con una A, ni con dos signos de interrogación (ni absolutamente nada)... Esto pasa porque simplemente ya se ha almacenado en un repositorio local dicho archivo en cuestión.

**Proceso de seguimiento a toda una carpeta (“Working Directory”) y su paso por area temporal (Staging Area) y posterior repositorio local (Local Repo).**

Luego de ejecutar el comando “git init” en la ruta dentro de la terminal que contiene el directorio de su interés, el que desea manipular (en este caso será “GIT:GITHUB”)... Si en caso dado queremos que todo el directorio, no sólo un archivo en cuestión contenido dentro de él, se mande al area temporal (staging area); primero que todo, debemos ubicarnos en la ruta o dirección especifica del directorio de nuestro interés, recordemos: “GIT:GITHUB”, dentro de la terminal (puede verificar eso con el comando pwd); y luego, seguidamente ejecutar el siguiente comando: “git add .”. Así, tal cual. *Qué hará “git add .”? Pues, tomará todos los archivos de su directorio, en este caso “GIT:GITHUB”, y se mandarán al area temporal (staging area), les hará seguimiento y podrán ser editables.*

Ahora sí, ya estando su directorio en el area temporal (staging area), ya podrá enviar inmediatamente desde ahí, el directorio en cuestión, a un repositorio local (local repo). De la misma forma, el comando a ejecutar para lograr ese fin sería el siguiente: “ git commit -m “aquí insertar comentario entre comillas” ”. *Quiere decir entonces que, todos los archivos contenidos dentro del directorio: “GIT:GITHUB”, por ejemplo, serán almacenados en una carpeta oculta “commit” que generará una copia de seguridad* ***inalterable*** *de cada uno de ellos, en su actual edición.Tal que así:*



A la copia de seguridad hecha sobre la carpeta “GIT:GITHUB” la hemos guardado con un comentario que dice: “copia seguridad git”.

*de interés: dentro del comentario del comando commit no haga uso de tildes ni caracteres extraños.*

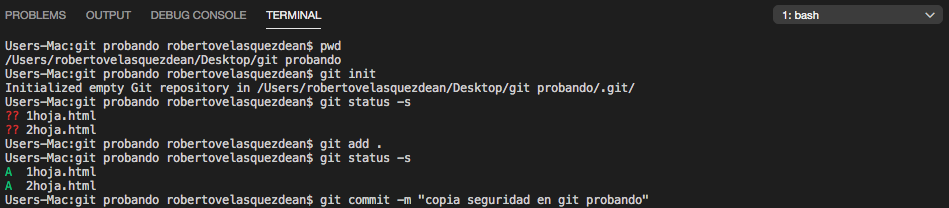
*Ahora bien, supongamos que queremos seguir editando los archivos que están contenidos dentro del directorio “GIT:GITHUB”; pues bien, usted puede hacerlo; pero, los cambios que haga, no se verán reflejados dentro del “GIT:GITHUB” recien mandado a repositorio local, No. Debe recordar que esas ediciones que decidio guardar en un repositorio local son inalterables. Entonces, dicho lo anterior, cada vez que haga un cambio sobre un archivo recien mandado a repositorio local (local repo), termine de hacer todos los cambios que desee y lo guarde en el editor de texto (con command + s)... para hacerle de nuevo un seguimiento a ese nuevo archivo modificado y que, posteriormente, se nos permita hacer nuevas copias de seguridad sobre esas nuevas modificaciones hechas; debe, primero, ejecutar de nuevo el comando “git add .”* “git add .” es el comando que nos permitirá registrar las modificaciones hechas sobre la totalidad de nuestros archivos del directorio que estamos manipulando y que las contiene (en el cual debemos vernos ubicados dentro de la ruta de la terminal, importante esto). Al igual, en caso que sólo queramos registrar modificaciones sobre un archivo en especifico; por ejemplo sobre el archivo “Untitled-1.html” de nuestro diretorio *“GIT:GITHUB”, podemos ejecutar el comando “git add* Untitled-1.html” directamente y no “git add .”.

Hemos olvidado decir que, luego de que un directorio o archivo en especifico sea guardado en una copia de seguridad inalterable con el comando “commit”; e, inmediatamante después, decidamos modificarle un par de cosas más y las guardemos en el editor de texto, el estado de los archivos modificados cambiará y también podrá percatarse de eso (esto lo puede visualizar antes de registrar los nuevos cambios hechos con el comando “git add”). Tal que así:

 M de modificado.

Ahora sí, lo que habíamos comentado antes: luego de hacer nuestras modificaciones, puede ejecutar de nuevo el comando: “git add .” o, bien, “git add 2hoja.html” en caso que sólo queramos hacerle seguimiento a ese único archivo modificado en cuestión; para luego sí poder crear, en el repositorio local (local repo), una nueva copia de seguridad inalterable sobre esa segunda y nueva edición del archivo “2hoja.html” por medio de una nueva ejecusión del comando: “ git commit -m “comentario cualquiera, sugiero comentar sobre el nuevo cambio en particular hecho” ”.

Repasemos absolutamente todo, ahora con un nuevo directorio de nombre: “git probando”. Veamoslo mejor:

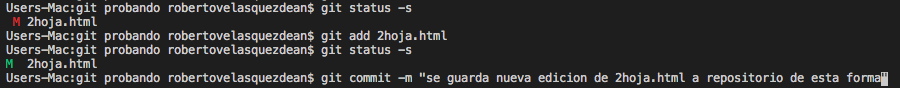


Ubicados ya en la ruta de la terminal que da con el directorio de nuestro interés a manipular: “git probando”. Hemos pasado nuestro directorio, luego de inicializado en GIT (git init) por nuestra area temporal (staging area) para hacerle un seguimiento con el comando “git add .” y, posteriormente, crearle copias de seguridad inalterables con el comando “git commit -m “” ”. Al igual, hemos verificado en cada proceso el estado del directorio con el comando “git status -s”.



Tiempo después, hemos decidido modificar un archivo en especifico del directorio luego de haberle creado ya una copia de seguridad en commit, ese archivo puntualmente es: “2hoja.html” (en la imagen se puede observar que, ejecutado el comando “status -s” luego de hacerle modificaciones al archivo, se aclara que el archivo ha sufrido de modificaciones efectivamente).

Sin embargo, recuerde que no se está modificando el archivo que ya está en el repositorio local, no, ese no se puede alterar ya: tan sólo se está haciendo una nueva edición adicional a esa, a partir de esa.

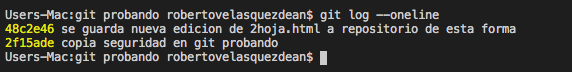


Esta nueva edición, si desea también guardarla en el repositorio local, debe primero también hacerle un seguimiento con el comando “git add” (esta vez como sólamente nos interesa hacerle seguimiento al único archivo que modificamos, entonces, dejamos eso establecido dentro del comando; tal que así: “git add 2hoja.html” y no “git add .” que corresponde a hacerle seguimiento a todo el directorio.)

Terminado el seguimiento, puede ahora sí, generar esa nueva copia de seguridad inalterable en el repositorio local (local repo) al ejecutar de nuevo el comando “git commit - m ‘edición 2 de 2hoja.html’ ”. Tal como se ve.

Ahora bien, es cierto que los archivos que están ya cargados en el repositorio local no pueden ser alterados; pero, definitivamente, sí puede crear versiones nuevas a partir de esas viejas, nuevos “commits”. (Caso aparte, los “commits” también pueden ser eliminados). Primero lo primero: hay que situarse en esa “vieja” versión. La idea es transportarse entre una copia de seguridad a otra dentro del repositorio local; esto para, posteriormente, tener la posibilidad de crear nuevas versiones a partir de viejas). Antes de proceder a ese paso, primero nos interesaría visualizar en un listado resumido todos los commits que hemos guardado; eso se hace con el comando “git log --oneline”: muestra en una lista los commits guardados.

Veamos cómo trabaja en el Visual Studio Code:



Efectivamente salen los únicos dos commits que guardamos en nuestro último ejemplo, *la parte amarilla* representa el nombre de cada uno de ellos y, al lado de sus nombres, los comentarios que nosotros como “users” pasamos por tecla para cada uno de ellos, ¿si los recuerda? Estos comentarios sirven, por consiguiente, como contenido referencial para ubicar puntualmente alguna edición guardada, de nuestro interés, en el repositorio local. Ese nombre del commit, *la parte amarilla*, tecnicamente es, en realidad, un identificador que nos sirve para viajar a través de las copias de seguridad ya guardadas (volver sobre cambios ya realizados anteriormente o a ediciones “viejas”, quizás nos arrepentimos de un avance que habíamos hecho, quién sabe!).

El comando que nos permite viajar entre commits es el siguiente: “git reset + identificador”. Por ejemplo, si resulta que actualmente estamos trabajando con la última copia de seguridad guardada: “48c2e46”, y queremos irnos de nuevo a la 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, pues estamos ante un comando destructivo, **especialmente cuando estamos trabajando de forma compartida nuestro repositorio. 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 temporal, 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 la versión que venía trabajando en esta “vieja” seleccionada; debe primero hacerles seguimiento (con git add) a los cambios que ha hecho; y, posteriormente, ejecutar ahora así el comando “git commit” para guardar una nueva copia de seguridad inalterable.

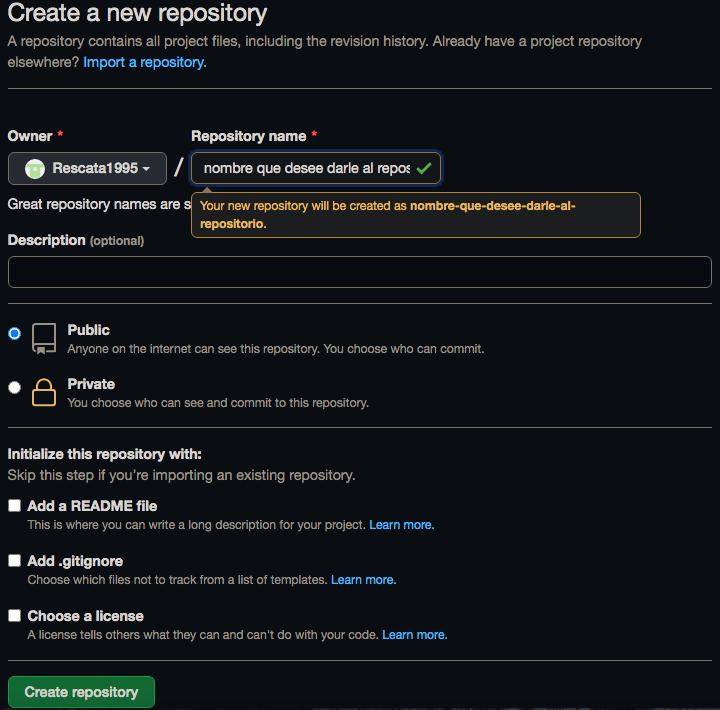
Sin embargo, git reset, también hace uso de funciones más especificas, se compone por el uso de *elementos* propios como: “hard” y “soft”. Se ejecutan así: “git reset --hard” y “git reset --soft”. La diferencia entre una y otra, respectivamente, consiste en que? Ya lo veremos.

**“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.

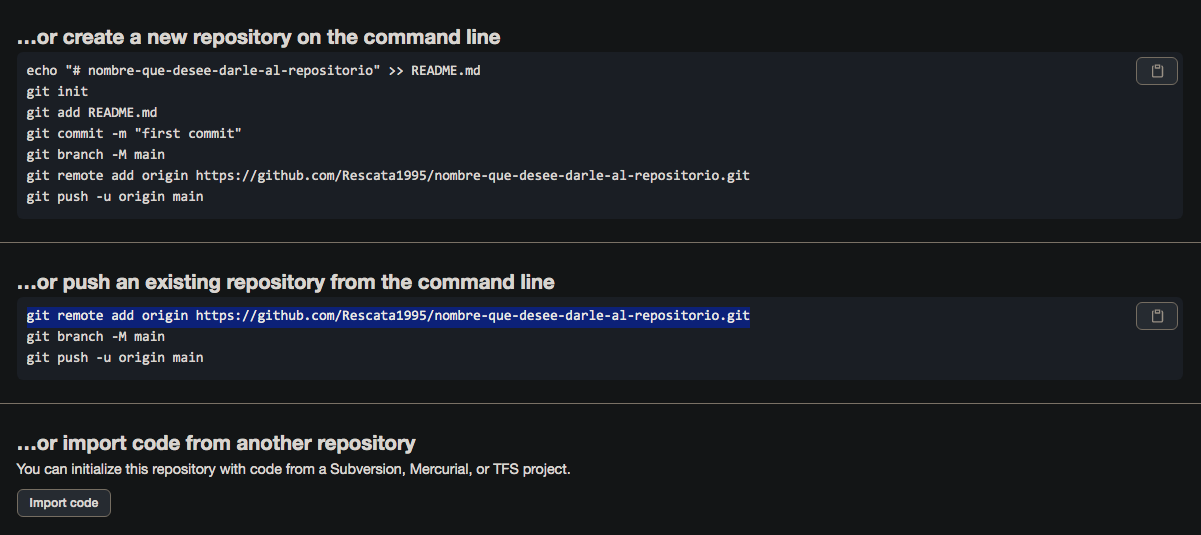
Ahora, **“soft”** es mucho más suave en su tratamiento efectivamente: también elimina el “commit”; pero, esta vez, los cambios no sólo se quedan en la inicialización del repositorio (como lo hace “git reset” por defecto); sino que también mantienen su seguimiento; es decir, los cambios del “commit” recientemente 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”.

***Tutorial de GITHUB***

Todos los repositorios que hemos creado en GIT ahora nos interesa guardarlos en una nube o servidor externo (GITHUB ocupa GIT en la nube, eso hace). Si en caso dado se nos quema nuestro disco duro ya evitaremos completamente el riesgo de perder absolutamente todo o me encuentro trabajando en un proyecto grupal en el que necesito cooperar con más programadores; pues, todos nuestros repositorios estarán en la nube, el cual podemos retornar nuevamente a nuestra pc en cualquier momento. Esa interacción entre GIT & GITHUB es la que nos interesa. Pasos desde la creación de cuenta en GITHUB hasta la creación de un repositorio remoto vinculado a nuestro editor de texto y terminal:



Le damos un nombre al repositorio remoto que deseemos crear y decidimos si será de caracter público o privado y listo: create repository!

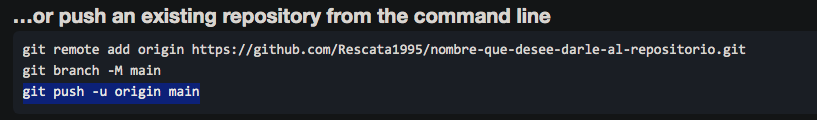


Luego, seleccionamos toda la ruta subrayada con fondo azul, la pegamos en nuestra terminal y presionamos “enter”, tal que así (el comando ***“git remote add origin”***... es el que tiene la función de vincular tu repositorio local con un repositorio remoto; de tal manera que, posteriormente, puedas exportar tus cambios guardados en el repositorio local a un repositorio remoto o, en su defecto, puedas importar los cambios guardados de un repositorio remoto a tu repositorio local”):



Esto le dirá a GIT que ya estaremos trabajando de forma remota. Es decir, va a crear un identificador de todos nuestros archivos o de todo el trabajo que hemos hecho con GIT en la URL que hemos importado de GITHUB (nuestro repositorio local ahora estará en una nube).

Entonces, al terminar de registrar o guardar todos los commits correspondientes en nuestro repositorio local; es decir, cuando terminemos la tarea, importaremos la siguiente ruta (subrayada en fondo color azul) a nuestra terminal (y presionamos “enter”):



Esto cargará de forma automatica todos nuestros repositorios locales a GITHUB; de tal manera que, no sólo tengamos repositorios locales, sino que también en la nube (libre de riesgos de pérdida): sólo actualice su página en GITHUB.

Ahora bien, con esto ya tenemos vinculado al GIT local con el GIT remoto; es decir, no es necesario llamar de nuevo las anteriores rutas subrayadas si quisieramos guardar un nuevo commit en nuestro repositorio local y cargarlo en la nube de GITHUB. ¿Entonces qué se hace? pues se ejecuta el siguiente comando: “git push”. Listo, eso es todo. Así hemos cargado en la nube GITHUB nuestro último commit guardado en nuestro repositorio local. En el caso contrario, es decir, que quisieramos importar un “commit” del servidor externo de GITHUB (ya sea porque lo hayamos creado allá y no en nuestro repositorio local); pues, el comando a ejecutar sería: “git pull”. Por último, si deseamos compaginar por igual los “commits” que tenemos en ambos repositorios, local y remoto, debemos ejecutar el siguiente comando: “git fetch”. (la explicación de este último comando puede mejorar).

***Sobre las Git* *Branch (citar)***

Una de las funciones más importantes de Git es el control de las ramas de desarrollo, conocidas como Branch, que ayudan a mejorar la creación de un proyecto. Las Git Branch ayudan a organizar múltiples versiones de una aplicación. Por eso es muy importante trabajar con ellas.

El uso de las ramas de desarrollo de Git es una excelente manera de trabajar con una aplicación mientras rastreas sus versiones.

Usar las ramas de desarrollo de Git tiene múltiples ventajas. Sin embargo, queremos enfatizar las siguientes dos:

- Hace posible desarrollar nuevas funciones para una aplicación sin obstaculizar el desarrollo de la rama principal.

- Con Git Branch es posible crear diferentes ramas de desarrollo que pueden converger en el mismo repositorio. Por ejemplo, una rama estable, una rama de prueba y una rama inestable.

***Cómo usar el comando Git Branch***

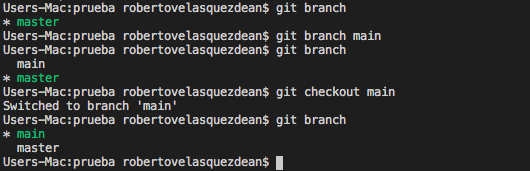
Los usos de las ramas de Git son inicialmente simples, como notarás en el uso de los comandos de Git Branch. Pero como con la mayoría de las cosas, cuantas más ramas tengas, más difícil será administrarlas.

En cualquier proyecto Git puedes ver todas las ramas ingresando el siguiente comando en la línea de comando: “git branch”.

Si no has creado una rama antes, no verás ningún resultado en el terminal. Crear una rama es realmente simple: “git branch [nueva\_rama]”. Luego, tienes que pasar a la rama de desarrollo recién creada. Para hacer esto, ejecuta el siguiente comando: “git checkout [nueva\_rama]”.

La salida te informará que cambiaste a una nueva rama. Para el ejemplo, digamos que llamamos a esta rama “prueba”, entonces será: “ Switched to branch ‘prueba’ ”.

Ahora, en esa nueva rama de desarrollo, puedes crear tantas modificaciones de código como quieras, sin tener que cambiar nada en la principal. Como puedes ver, el programa se mantendrá organizado para nuevas inclusiones de código. Si ejecutas el comando para volver a listar las ramas, verás que se agrega una nueva rama y que estás en ella: *git branch.* Tal que así:



Hay algo que debes tener en cuenta si quieres crear una nueva rama de desarrollo. Primero, debes comprometerla con la rama principal para que Git entienda cuál es la rama maestra. Si no haces esto, obtendrás un error. Primero, crea el enlace y luego crea las ramas de desarrollo.

***Si quieres eliminar una rama de Git, puedes hacerlo con el siguiente comando: “git branch -d [nombre\_de\_ rama]” (sigamos suponiendo que la rama se llama: “prueba”).***

Sin embargo, para hacer esto, no debes estar en la rama que quieres eliminar. Entonces, en este caso, te mueves a la rama maestra y desde allí eliminas la rama que acabas de crear, tal que así:

1. “git checkout master”.

***2. “git branch -d prueba”... si se pone complicado, “git branch -D prueba”***

Finalmente, llega un punto en el que has realizado muchas modificaciones en una rama de desarrollo adicional y de “tasting” (creada a partir de la principal) y se vuelve estable, por lo que quieres vincular eso nuevos cambios a tu rama principal. ***Para eso, existe el comando “merge” (fusión).*** Primero, ubica la rama de desarrollo (la principal) a la que se le adjuntarán los cambios de la segunda rama (la rama de “tasting” que creamos a partir de la principal para probar modificaciones y ver con qué quedarnos de ella, de forma definitiva, al llamarla luego en la rama principal).

Por ejemplo, si queremos adjuntar la rama de **“prueba”** a la rama **master** (porque ya nos gustó en su totalidad los cambios que hicimos en la rama de “prueba” y queremos vincular esas modificaciones definitivas en nuestra rama master: *la rama con la que presentaremos nuestro proyecto final, por decir algo*); lo que hay que hacer es, primero, ubicarnos de nuevo en la rama master (“***git checkout master***”) y fusionarla con la rama de “prueba” usando el siguiente comando: ***git merge prueba***

***Ahora, un detalle,*** *si nuestro repositorio local se encuentra vinculado a un servidor externo, tipo repositorio remoto GIT HUB; antes que nada,* ***e incluso antes de crear nuestra rama “prueba” en repositorio local*** *(por continuar con nuestro ejemplo), para manipular distintas ramas secundarias que desee, debe crearlas primero en GITHUB (antes que en su repositorio local, recuerde).*

*Luego, en repositorio local, sí procede a crearlas (con el mismo nombre con el que las registró en GITHUB); y, es desde ahí, desde su repositorio local, donde hará todas las modificaciones necesarias en las ramas secundarias que crea o creó a partir de la rama principal (en este caso, hemos creado la rama “prueba” a partir de la rama principal “master”, esa rama “prueba” ya debio estar asi tal cual creada en GITHUB desde un principio). Entonces, luego de que haga los respectivos cambios definitivos en repositorio local sobre la rama “prueba”; luego de eso, es que usted va a exportar dichos cambios a repositorio externo GITHUB con un comando:* ***“git push”. Y si dichos cambios tornan ser definitivos y utiles para incorporalos ya en la rama principal (master), es ahí que se hace una fusión entre rama principal “master” y rama “prueba”, tal cual como la explicamos.***

***En conclusión, una rama es como una copia de su proyecto. Se utiliza principalmente para:***

1. agregar una característica en desarrollo.

2. colaborando en el mismo proyecto con otros desarrolladores.

3. no romper todo tu repositorio.

4. no molestar a tus compañeros de trabajo.

5. El propósito de una rama es aislar su trabajo del código base principal de su proyecto y / o del trabajo de sus compañeros de trabajo.

***Otros comandos Git interesantes***

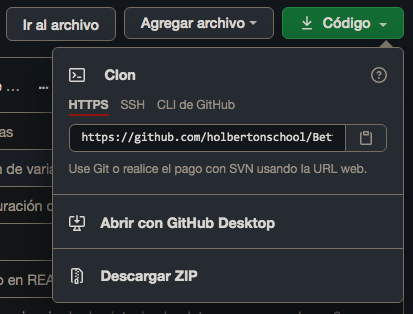
***git status:*** *nos indica en que rama (branch) estamos ubicados actualmente.*

***git checkout -b “nombre de rama”:*** *este comando lo que hace es crear una nueva rama secundaria; pero, adicionalmente y luego de creada, la terminal nos lleva a la ubicación de dicha rama. Es decir, no sólo la crea, sino que además nos situa en ella. (antes de ejecutar este comando, idealmente asegurese de estar ya situado en la rama principal, la rama de origen).*

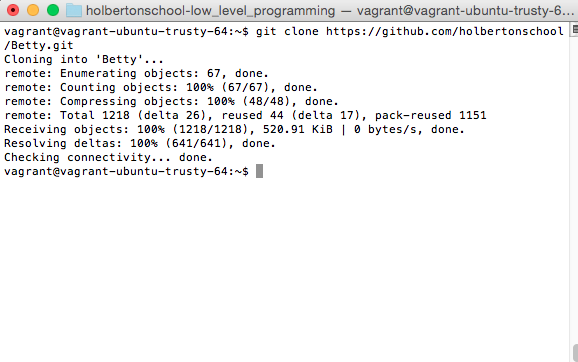
***git branch -m “nombre de la rama” “nuevo nombre de la rama”:*** *este comando lo que hace es cambiar el nombre de una rama existente, especificando también el nuevo nombre que va a recibir. (antes de ejecutar este comando, idealmente asegurese de estar ya situado en la rama principal, la rama de origen).*

***git branch -h:*** *comando que ejecuta el asistente de ayuda para conocer todo lo se puede hacer con las ramas (Branch).*

***git clone:*** *tiene la función de importar un repositorio remoto de github (u otro servidor externo), que aún no está vinculado a repositorio local (por lo general esto se hace cuando queremos vincular el repositorio remoto o en nube de un tercero). Si el repositorio no nos pertenece, para importarlo debe hacerlo de esta forma.*

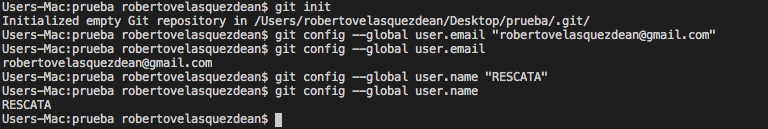


*Nos dirigimos a esta parte del repositorio remoto que queramos importar (este ejemplo se hace con GitHub) y copiamos link. Luego, para vincularlo a nuestro repositorio local, sería así:*



***Dato de importancia: trate de que se encuentren sincronizadas las ramas (branch) de su repositorio local (GIT) con el repositorio remoto (GITHUB); si no lo están, pues sincronicelas, no podrá cargar la información de su repositorio local a remoto si no están sincronizadas.***

***Opcional: No se olvide registrar su nombre de “user” y “email” por cada GIT inicializado, GIT REMOTO necesita saber quién está registrando copias de seguridad por cada repositorio local en su nube o servidor externo: la idea es conocer quién nos está colaborando con el material cargado en GIT HUB por ejemplo. Se ve más o menos así:***



***Obligación: tampoco se olvide de su user en GITHUB: GITHUBuser; ni de su contraseña: GITHUBpassword. La terminal se lo exige luego de que vincule y trabaje de la mano con dicho repositorio remoto.***

***Dato: Si instalo GIT por medio de la terminal, pues ejecute GIT directamente desde la terminal. Evitese futuros dolores de cabeza.***

***Dato: No se puede inicializar git luego de ya inicializado antes una carpeta, esto va a desconfigurar su repositorio local actual y todo lo que ya haya hecho en él; pues, justamente le estás pidiendo a GIT que se restaure, que se registre de nuevo un nuevo repositorio; entonces, si por ejemplo, desea cargar algún cambio a su repositorio remoto que anteriormente ya estaba vinculado al local... no pasará nada, porque prácticamente ahora está trabajando en un repositorio local nuevo que no está vinculado a ningún servidor externo (le tocaría crear uno nuevo, un nuevo repositorio remoto y vincularlo al nuevo repositorio local inicializado).***

***Ahora, si por ejemplo desea, eliminar un repositorio de Git creado con “git init” en un directorio; lo hace de la siguiente forma:***

***cd “directorio\_de\_interés\_donde\_inicializó\_git\_que\_desea\_eliminar”  
rm -rf .git***

***Es de la anterior manera porque “Git” guarda, de forma automatica, todos los cambios realizados en un directorio llamado “.git” (dentro del directorio donde se inicializó “git” inicialmente)***

