FLUJOS DE TRABAJO EN GIT

**Inicializar un repositorio de Git**

* **git init**: crea un respositorio de manera local en el directorio donde posicionemos el comando. Si ejecutamos el comando “git init” (sin especificar el directorio) nos creará el repositorio en el lugar donde estemos, en cambio, si ejecutamos el comando “git init [nombre\_directorio], nos creará el repositorio de git en el directorio especificado.

Para eliminar un repositorio de git sin eliminar todo el directorio, es necesario eliminar la carpeta oculta “.git” que tenemos en el directorio del repositorio de git. Para ver los archivos ocultos utiliza “**ls -a**”. Ejecuta esto para eliminar el repositorio de git: **rm -rf .git .**

**Agregando, quitando y viendo el estatus de archivos**

Hemos visto como inicializar un proyecto y, sabemos que Git trabaja con un proceso para introducir cambios que consta de tres estados: El Working Directory, el Staging Area y el Git Repository. Ahora vamos a ver como podemos proceder a mover archivos entre el Working Directory y el Staging Area.

* **git status** -> Nos permite ver el estado de nuestros archivos y directorios dentro del repositorio de Git.
  + Todos los archivos que no están “trackeados” significa que se encuentran en el *Working Directory*.
* **git add [archivos o directorios]** -> Para agregar todos los archivos que se encuentran en el *Working Directory.* Si escribes “(git add \*) o (git add -A)” añadirías todos los archivos y directorios de Git al *Staging Area.*
* **git rm --cached [archivo]** -> Elimina el archivo en concreto del *Staging Area*.
* **git rm -f [archivo]** -> Elimina el archivo completamente, tando del Staging Area como del Working Directory. Desaparece completamente.

**Confirmando cambios**

Llegado el punto de tener los archivos en el Staging Area, llega el momento de pasar esos archivos al repositorio, cambiando el estado de “to be commited” a “nothing to commit”.

Para ejecutar este paso y hacer el commit de los archivos en el “Staging Area” hay que escribir en la terminal:

* **git commit -m “**[mensaje]**”** -> Intentar ser descriptivos en el mensaje del commit para saber qué cambios se han realizado.

En caso de hacer un commit y querer anexar algún cambio que no habíamos añadido podemos utilizar el flag **--amend** después de **git commit**:

* **git commit --amend**

Si quieres sobreescribir el mensaje del commit puedes sencillamente añadir el flag **-m “[mensaje]”** después del **--amend**:

* **git commit --amend -m “**[mensaje]**”**

En el momento de ejecutar el commit con el flag --amend el terminal te abrirá el archivo donde se registra el commit y tan solo tendrás que darle a [esc] y escribir :wq para guardar los cambios y cerrar el archivo.

Si ejecutamos el comando **git log** podremos ver la lista de commits en el repositorio.

**Etiquetando confirmaciones**

Para versionar y etiquetar los cambios de un proyecto en git, podemos utilizar el comando **git tag.**

En git tenemos dos tipos de tags: Las etiquetas anotadas y las etiquetas ligeras.

- Las **etiquetas anotadas** son objetos completos en la base de datos de Git conteniendo varios parámetros entre los que se encuentran nombre del etiquetador, email, fecha, mensaje de la etiqueta, etc...

- En cambio, las **etiquetas ligeras** funcionan como un puntero al commit especificado.

* **git tag [etiqueta]** ->Nos permite añadir una etiqueta al último commit realizado. Ej: git tag 1.0
  + Si especificamos el SHA del commit podremos indicar el *tag ligero* exactamente en el commit que queremos.
* **git tag** o **git tag -l** -> Lista las etiquetas en orden alfabético.
  + Si queremos filtrar las etiquetas en una serie concreta podemos buscarla especificando el patrón concreto. Ej: **git tag -l “1.2.\*”** -> El asterisco indica que filtre todas las versiones dentro de la versión “1.2” (1.2.1, 1.2.2, 1.2.3, etc...).
* Podemos especificar a git tag el parámetro/flag:
  + **-a** -> Para indicar que es una **etiqueta anotada**. Si no lo especificamos añadiremos la etiqueta como **etiqueta ligera**.
  + **-m** -> Para indicar un mensaje a la etiqueta (esta opción solo está disponible para las etiquetas anotadas).
  + **-l** -> Para listar las etiquetas marcadas.
  + **-d** -> Para eliminar una etiqueta. Ej: **git tag -d 1.0**
  + **-f** -> Para renombrar una etiqueta modificando el número de versión y el mensaje de la etiqueta. Ej: **git tag -f -a [etiqueta] -m “[mensaje]” [clave SHA commit]** 
    - TEST: **git tag -f -a 1.1 -m “Etiqueta cambiada prototipo” 2d53c3acfeee18851095da0709b887ac6f1a727f**

**Git log**

En git podemos revisar el historial de nuestro proyecto. Podemos invocarlo ejecutando **git log** y podremos visualizar los commits que hemos hecho hasta ahora. Pero, lo interesante de git log es que es bastante personalizable. Podemos personalizar git log con:

* **--oneline** -> Permite visualizar el historial entero muy simplificado mostrando el HASH mucho más pequeño (7 carácteres), el tag y el mensaje del commit, todo en una sola linea.
* **--graph** -> Muestra los commits en las ramas correspondientes.
* **git log -[número]** -> Nos muestra el número de commits indicados en el número. Ej: **git log -3** (muestra los últimos 3 commits).
* **--stat** -> Podemos ver los archivos que fueron modificados en cada commit y la información general de cada archivo (peso, tipo de archivo, etc... )

Aquí dejo un comando personalizado para tener un log mucho más cool (**superlog**):

git config --global alias.superlog "log --graph --abbrev-commit --decorate --date=relative --format=format:'%C(bold blue)%h%C(reset) - %C(bold green)(%ar)%C(reset) %C(white)%s%C(reset) %C(dim white)- %an%C(reset)%C(bold yellow)%d%C(reset)' --all"

Para ejecutarlo, en vez de escribir “git log”, hay que escribir “git superlog”.

Si ejecutamos **git log > commits.txt** , nos genera un archivo de texto con todos nuestros commits.

**Revisar cambios entre versiones**

Para comprobar qué cambios hemos realizado entre un commit y otro, podemos utilizar la herramienta de revisión de cambios con el comando **git diff** y el SHA-1 de los commits.

* **git diff [SHA-1 commit]** -> Nos muestra los cambios sucedidos desde el commit que indicamos hasta el actual. Ej: **git diff cdfac54**

Si especificamos dos commits con sus respectivas llaves HASH, podremos comparar los cambios de dos commits específicos. Dependiendo del orden en el que pongamos un commit u otro nos indicará si hemos agregado o eliminado archivos y código, por lo tanto, es muy importante mantener el orden de la versión más anterior a la más actual. Ej:

* git diff [versión 1] vs [versión 2].

Cuando nos aparezca en rojo significa que se eliminó contenido y, si aparece en verde, significa que se ha añadido contenido de un commit a otro.

Además, no solo podemos hacer **git diff** con el SHA-1, también podemos utilizar los tags para especificar el/los commits que queremos comparar.

* **git diff [etiqueta] [etiqueta]** -> Ej: **git diff 0.1 0.2**

**Git reset**

A veces nos equivocamos y queremos eliminar el commit que hemos enviado. Para eso tenemos el comando **git reset**.

Es un comando muy delicado porque puedes eliminar varios commits y cargarte el repositorio.

También podemos eliminar del Staging Area archivos con el comando:

* **git reset HEAD [archivo]**

Antes de hacer cualquier reset es muy recomendable hacer un respaldo de la bitácora del log:

* **git log > bitacora\_log.txt**

Hay tres tipos principales de **git reset:**

**git reset --soft**

Git reset --soft te permite eliminar todos los commits desde el commit que especifiques pero dejándote los archivos preparados en el Staging Area para poder hacer el commit de nuevo de todos los commits borrados. No se especifica el commit que quieres eliminar concretamente sino desde cuál quieres que se borre, en un orden ascendente hasta el más reciente. El commit que especifiques no se eliminará, solo los siguientes a este hasta el actual.

* + git reset --soft [sha-1]

Podemos especificar desde qué commit queremos hacer el reset tanto con el **SHA-1** como con los **tags**.

Cuidado que al eliminar los commits, las etiquetas permanecen intactas. Podemos comprobarlo utilizando el comando **git tag -l.** Al hacer el nuevo commit no nos asociará la etiqueta, así que tendremos que eliminar la etiqueta y volver a asignarla al nuevo commit.

**git reset --mixed**

La diferencia de git reset --mixed a git reset --soft es que con soft te deja los cambios listos en el Staging Area y con mixed te deja todos los cambios en el Working Area.

**git reset --hard**

Con el git reset --hard hay que ir con especial cuidado porque a diferencia de los dos anteriores resets, este te puede eliminar los archivos no solo del commit sino del propio directorio.

Si ejecutas **git reset --hard** directamente en el repositorio de git sin parámetros ni especificando ningún commit, con archivos preparados en el Staging Area, esos archivos no solo te los eliminará del Staging Area sino que también te lo eliminará del Working Directory; básicamente, **te eliminará completamente los archivos de tu directorio**.

Si especificas el **sha-1** del commit hasta dónde quieres eliminar, te borrará tanto los commits como todos los cambios y archivos dentro de esos commits que hemos indicado que elimine.

Como hemos dicho es un comando delicado que puede borrar todo tu progreso, por suerte, si antes de hacer el **reset --hard**, has hecho un backup del log (**git log > bitacora.txt**) o tienes en otra ventana del terminal, el log con la llave sha-1 del commit al que quieres regresar, git tiene un truco para recuperar todo lo eliminado especificando ese mismo sha-1 con el mismo comando **git reset --hard**.

* Ej: **git reset --hard [sha-1 commit eliminado]**

También podemos utilizar el comando **git reflog** para poder todo el historial de lo que hemos hecho en git con los respectivos SHA-1s. Por lo tanto es otra forma de recuperar el contenido eliminado de un git reset --hard.