Comandos para manejo de GIT

Git reset y git rm son comandos con utilidades muy diferentes, pero aún así se confunden muy fácilmente.

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.
Recuerda que git rm no puede usarse así nomás. Debemos usar uno de los flags para indicar a Git cómo eliminar los archivos que ya no necesitamos en la última versión del proyecto:

- git rm --cached: Elimina los archivos del área de Staging y del próximo commit pero los mantiene en nuestro disco duro.
- git rm --force: 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 usar comandos más avanzados).

git reset

Este comando nos ayuda a volver en el tiempo. 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.
Este comando es muy peligroso y debemos usarlo solo en caso de emergencia. Recuerda que debemos usar alguna de estas dos opciones:
Hay dos formas de usar git reset: con el argumento --hard, borrando toda la información que tengamos en el área de staging (y perdiendo todo para siempre). O, un poco más seguro, con el argumento --soft, que mantiene allí los archivos del área de staging para que podamos aplicar nuestros últimos cambios pero desde un commit anterior.

- git reset --soft: Borramos todo el historial y los registros de Git pero guardamos los cambios que tengamos en Staging, así podemos aplicar las últimas actualizaciones a un nuevo commit.

- git reset --hard: Borra todo. Todo todito, absolutamente todo. Toda la información de los commits y del área de staging se borra del historial.

- git reset HEAD: Este es el comando para sacar archivos del área de Staging. No para borrarlos ni nada de eso, solo para que los últimos cambios de estos archivos no se envíen al último commit, a menos que cambiemos de opinión y los incluyamos de nuevo en staging con git add, por supuesto.

¿Por qué esto es importante?

Imagina el siguiente caso:

Hacemos cambios en los archivos de un proyecto para una nueva actualización. Todos los archivos con cambios se mueven al área de staging con el comando git add. Pero te das cuenta de que uno de esos archivos no está listo todavía. Actualizaste el archivo pero ese cambio no debe ir en el próximo commit por ahora.

¿Qué podemos hacer?

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.
¡Al usar git rm lo que haremos será eliminar este archivo completamente de git! Todavía tendremos el historial de cambios de este archivo, con la eliminación del archivo como su última actualización. Recuerda que en este caso no buscábamos eliminar un archivo, solo dejarlo como estaba y actualizarlo después, no en este commit.
En cambio, si usamos git reset HEAD, lo único que haremos será mover estos cambios de Staging a Unstaged. Seguiremos teniendo los últimos cambios del archivo, el repositorio mantendrá el archivo (no con sus últimos cambios pero sí con los últimos en los que hicimos commit) y no habremos perdido nada.
Conclusión: Lo mejor que puedes hacer para salvar tu puesto y evitar un incendio en tu trabajo es conocer muy bien la diferencia y los riesgos de todos los comandos de Git.

Por ahora, nuestro proyecto vive únicamente en nuestra computadora. Esto significa que no hay forma de que otros miembros del equipo trabajen en él.
Para solucionar esto están los servidores remotos: un nuevo estado que deben seguir nuestros archivos para conectarse y trabajar con equipos de cualquier parte del mundo.

Estos servidores remotos pueden estar alojados en GitHub, GitLab, BitBucket, entre otros. Lo que van a hacer es guardar el mismo repositorio que tienes en tu computadora y darnos una URL con la que todos podremos acceder a los archivos del proyecto para descargarlos, hacer cambios y volverlos a enviar al servidor remoto para que otras personas vean los cambios, comparen sus versiones y creen nuevas propuestas para el proyecto.

Esto significa que debes aprender algunos nuevos comandos:

- git clone url_del_servidor_remoto: Nos permite descargar los archivos de la última versión de la rama principal y todo el historial de cambios en la carpeta .git.
- git push: Luego de hacer git add y git commit debemos ejecutar este comando para mandar los cambios al servidor remoto.
- git fetch: Lo usamos para traer actualizaciones del servidor remoto y guardarlas en nuestro repositorio local (en caso de que haya, por supuesto).
- git merge: También usamos el comando git merge con servidores remotos. Lo necesitamos para combinar los últimos cambios del servidor remoto y nuestro directorio de trabajo.
- git pull: Básicamente, git fetch y git merge al mismo tiempo.

Las ramas son la forma de hacer cambios en nuestro proyecto sin afectar el flujo de trabajo de la rama principal. Esto porque queremos trabajar una parte muy específica de la aplicación o simplemente experimentar.

La cabecera o HEAD representan la rama y el commit de esa rama donde estamos trabajando. Por defecto, esta cabecera aparecerá en el último commit de nuestra rama principal. Pero podemos cambiarlo al crear una rama (git branch rama, git checkout -b rama) o movernos en el tiempo a cualquier otro commit de cualquier otra rama con los comandos (git reset id-commit, git checkout rama-o-id-commit).
Git nunca borra nada a menos que nosotros se lo indiquemos. Cuando usamos los comandos git merge o git checkout estamos cambiando de rama o creando un nuevo commit, no borrando ramas ni commits (recuerda que puedes borrar commits con git reset y ramas con git branch -d).

Git es muy inteligente y puede resolver algunos conflictos automáticamente: cambios, nuevas líneas, entre otros. Pero algunas veces no sabe cómo resolver estas diferencias, por ejemplo, cuando dos ramas diferentes hacen cambios distintos a una misma línea.
Esto lo conocemos como conflicto y lo podemos resolver manualmente, solo debemos hacer el merge, ir a nuestro editor de código y elegir si queremos quedarnos con alguna de estas dos versiones o algo diferente. Algunos editores de código como VSCode nos ayudan a resolver estos conflictos sin necesidad de borrar o escribir líneas de texto, basta con hundir un botón y guardar el archivo.
Recuerda que siempre debemos crear un nuevo commit para aplicar los cambios del merge. Si Git puede resolver el conflicto hará commit automáticamente. Pero, en caso de no poder resolverlo, debemos solucionarlo y hacer el commit.
Los archivos con conflictos por el comando git merge entran en un nuevo estado que conocemos como Unmerged. Funcionan muy parecido a los archivos en estado Unstaged, algo así como un estado intermedio entre Untracked y Unstaged, solo debemos ejecutar git add para pasarlos al área de staging y git commit para aplicar los cambios en el repositorio.
GitHub es una plataforma que nos permite guardar repositorios de Git que podemos usar como servidores remotos y ejecutar algunos comandos de forma visual e interactiva (sin necesidad de la consola de comandos).
Luego de crear nuestra cuenta, podemos crear o importar repositorios, crear organizaciones y proyectos de trabajo, descubrir repositorios de otras personas, contribuir a esos proyectos, dar estrellas y muchas otras cosas.
El README.md es el archivo que veremos por defecto al entrar a un repositorio. Es una muy buena práctica configurarlo para describir el proyecto, los requerimientos y las instrucciones que debemos seguir para contribuir correctamente.

Para clonar un repositorio desde GitHub (o cualquier otro servidor remoto) debemos copiar la URL (por ahora, usando HTTPS) y ejecutar el comando git clone + la URL que acabamos de copiar. Esto descargara la versión de nuestro proyecto que se encuentra en GitHub.

Sin embargo, esto solo funciona para las personas que quieren empezar a contribuir en el proyecto. Si queremos conectar el repositorio de GitHub con nuestro repositorio local, el que creamos con git init, debemos ejecutar las siguientes instrucciones:

Primero: Guardar la URL del repositorio de GitHub con el nombre de origin

- git remote add origin URL

Segundo: Verificar que la URL se haya guardado correctamente:

- git remote
- git remote -v

Tercero: Traer la versión del repositorio remoto y hacer merge para crear un commit con los archivos de ambas partes.

Podemos usar git fetch y git merge o solo el git pull con el flag --allow-unrelated-histories:

- git pull origin master --allow-unrelated-histories

Por último, ahora sí podemos hacer git push para guardar los cambios de nuestro repositorio local en GitHub:

- git push origin master

Para eliminar tienen que escribir lo siguiente:

- git remote remove origin

Las llaves públicas y privadas nos ayudan a cifrar y descifrar nuestros archivos de forma que los podamos compartir sin correr el riesgo de que sean interceptados por personas con malas intenciones.
La forma de hacerlo es la siguiente:

1. Ambas personas deben crear su llave pública y privada.
2. Ambas personas pueden compartir su llave pública a las otras partes (recuerda que esta llave es pública, no hay problema si la “interceptan”).
3. La persona que quiere compartir un mensaje puede usar la llave pública de la otra persona para cifrar los archivos y asegurarse que solo puedan ser descifrados con la llave privada de la persona con la que queremos compartir el mensaje.
4. El mensaje está cifrado y puede ser enviado a la otra persona sin problemas en caso de que los archivos sean interceptados.
5. La persona a la que enviamos el mensaje cifrado puede usar su llave privada para descifrar el mensaje y ver los archivos.

Puedes compartir tu llave pública pero nunca tu llave privada.

Primer paso: Generar tus llaves SSH. Recuerda que es muy buena idea proteger tu llave privada con una contraseña.

- ssh-keygen -t rsa -b 4096 -C "tu@email.com"

Segundo paso: Terminar de configurar nuestro sistema.
En Windows y Linux:
Encender el "servidor" de llaves SSH de tu computadora:

- eval $(ssh-agent -s)

Añadir tu llave SSH a este "servidor":

- ssh-add ruta-donde-guardaste-tu-llave-privada

En Mac:
Encender el "servidor" de llaves SSH de tu computadora:

- eval "$(ssh-agent -s)"

Si usas una versión de OSX superior a Mac Sierra (v10.12) debes crear o modificar un archivo "config" en la carpeta de tu usuario con el siguiente contenido (ten cuidado con las mayúsculas):

```
Host *
    AddKeysToAgent yes
    UseKeychain yes
    IdentityFile ruta-donde-guardaste-tu-llave-privada
```

Añadir tu llave SSH al "servidor" de llaves SSH de tu computadora (en caso de error puedes ejecutar este mismo comando pero sin el argumento -K):

```
ssh-add -K ruta-donde-guardaste-tu-llave-privada
```

Agrega tu llave

```
ssh-add -K ~/.ssh/id_rsa
```

Luego de crear nuestras llaves SSH podemos entregarle la llave pública a GitHub para comunicarnos de forma segura y sin necesidad de escribir nuestro usuario y contraseña todo el tiempo.

Para esto debes entrar a la Configuración de Llaves SSH en GitHub, crear una nueva llave con el nombre que le quieras dar y el contenido de la llave pública de tu computadora.
Ahora podemos actualizar la URL que guardamos en nuestro repositorio remoto, solo que, en vez de guardar la URL con HTTPS, vamos a usar la URL con SSH:

- git remote set-url origin url-ssh-del-repositorio-en-github

Los tags o etiquetas nos permiten asignar versiones a los commits con cambios más importantes o significativos de nuestro proyecto.

Comandos para trabajar con etiquetas:

- Crear un nuevo tag y asignarlo a un commit: git tag -a nombre-del-tag id-del-commit.
- Borrar un tag en el repositorio local: git tag -d nombre-del-tag.
- Listar los tags de nuestro repositorio local:
    - git tag o
    - git show-ref --tags.
- Publicar un tag en el repositorio remoto: git push origin --tags.
- Borrar un tag del repositorio remoto:
    - git tag -d nombre-del-tag
    - git push origin :refs/tags/nombre-del-tag.

Para ver todo el historial de lo que hemos desarrollado:

- git log --all --graph
- git log --all --graph --decorate --oneline

Para renombrar una línea de comando demasiada grande.

- alias historial="git log --all --graph --decorate --oneline"

Manejo de ramas en GitHub

Puedes trabajar con ramas que nunca envías a GitHub, así como pueden haber ramas importantes en GitHub que nunca usas en el repositorio local. Lo importante es que aprendas a manejarlas para trabajar profesionalmente.

- Crear una rama en el repositorio local:
    - git branch nombre-de-la-rama
    - git checkout -b nombre-de-la-rama

- Publicar una rama local al repositorio remoto:
    - git push origin nombre-de-la-rama

Recuerda que podemos ver gráficamente nuestro entorno y flujo de trabajo local con Git usando el comando gitk.

Configurar múltiples colaboradores en un repositorio de GitHub

Por defecto, cualquier persona puede clonar o descargar tu proyecto desde GitHub, pero no pueden crear commits, ni ramas, ni nada.

Existen varias formas de solucionar esto para poder aceptar contribuciones. Una de ellas es añadir a cada persona de nuestro equipo como colaborador de nuestro repositorio.

Solo debemos entrar a la configuración de colaboradores de nuestro proyecto (Repositorio > Settings > Collaborators) y añadir el email o username de los nuevos colaboradores.

Flujo de trabajo profesional con Pull requests

En un entorno profesional normalmente se bloquea la rama master, y para enviar código a dicha rama pasa por un code review y luego de su aprobación se unen códigos con los llamados merge request.

Para realizar pruebas enviamos el código a servidores que normalmente los llamamos staging develop (servidores de pruebas) luego de que se realizan las pruebas pertinentes tanto de código como de la aplicación estos pasan a el servidor de producción con el ya antes mencionado merge request.

Forks o Bifurcaciones

Es una característica única de GitHub en la que se crea una copia exacta del estado actual de un repositorio directamente en GitHub, éste repositorio podrá servir como otro origen y se podrá clonar (como cualquier otro repositorio), en pocas palabras, lo podremos utilizar como un git cualquiera

.

Un fork es como una bifurcación del repositorio completo, tiene una historia en común, pero de repente se bifurca y pueden variar los cambios, ya que ambos proyectos podrán ser modificados en paralelo y para estar al día un colaborador tendrá que estar actualizando su fork con la información del original.

.

Al hacer un fork de un proyecto en GitHub, te conviertes en dueñ@ del repositorio fork, puedes trabajar en este con todos los permisos, pero es un repositorio completamente diferente que el original, teniendo alguna historia en común.

.

Los forks son importantes porque es la manera en la que funciona el open source, ya que, una persona puede no ser colaborador de un proyecto, pero puede contribuir al mismo, haciendo mejor software que pueda ser utilizado por cualquiera.

.

Al hacer un fork, GitHub sabe que se hizo el fork del proyecto, por lo que se le permite al colaborador hacer pull request desde su repositorio propio.

Trabajando con más de 1 repositorio remoto

Cuando trabajas en un proyecto que existe en diferentes repositorios remotos (normalmente a causa de un fork) es muy probable que desees poder trabajar con ambos repositorios, para esto puedes crear un remoto adicional desde consola.

- git remote add <nombre_del_remoto> <url_del_remoto>

- git remote upstream https://github.com/freddier/hyperblog

Al crear un remoto adicional podremos, hacer pull desde el nuevo origen (en caso de tener permisos podremos hacer fetch y push)

- git pull <remoto> <rama>

- git pull upstream master

Éste pull nos traerá los cambios del remoto, por lo que se estará al día en el proyecto, el flujo de trabajo cambia, en adelante se estará trabajando haciendo pull desde el upstream y push al origin para pasar a hacer pull request.

- git pull upstream master
- git push origin master

Git Rebase: reorganizando el trabajo realizado

El comando rebase es una mala práctica, nunca se debe usar, pero se los voy a enseñar para que hagan sus propios experimentos. Con rebase puedes recoger todos los cambios confirmados en una rama y ponerlos sobre otra.

Cambiamos a la rama que queremos traer los cambios

- git checkout experiment

Aplicamos rebase para traer los cambios de la rama que queremos

- git rebase master

Git Stash: Guardar cambios en memoria y recuperarlos después

Stashed:

El stashed nos sirve para guardar cambios para después, Es una lista de estados que nos guarda algunos cambios que hicimos en Staging para poder cambiar de rama sin perder el trabajo que todavía no guardamos en un commit

Ésto es especialmente útil porque hay veces que no se permite cambiar de rama, ésto porque porque tenemos cambios sin guardar, no siempre es un cambio lo suficientemente bueno como para hacer un commit, pero no queremos perder ese código en el que estuvimos trabajando.

El stashed nos permite cambiar de ramas, hacer cambios, trabajar en otras cosas y, más adelante, retomar el trabajo con los archivos que teníamos en Staging pero que podemos recuperar ya que los guardamos en el Stash.

git stash

El comando git stash guarda el trabajo actual del Staging en una lista diseñada para ser temporal llamada Stash, para que pueda ser recuperado en el futuro.
Para agregar los cambios al stash se utiliza el comando:
git stash

Podemos poner un mensaje en el stash, para asi diferenciarlos en git stash list por si tenemos varios elementos en el stash. Ésto con:
git stash save "mensaje identificador del elemento del stashed"

Obtener elementos del stash

El stashed se comporta como una Stack de datos comportándose de manera tipo LIFO (del inglés Last In, First Out, «último en entrar, primero en salir»), así podemos acceder al método pop.

El método pop recuperará y sacará de la lista el último estado del stashed y lo insertará en el staging area, por lo que es importante saber en qué branch te encuentras para poder recuperarlo, ya que el stash será agnóstico a la rama o estado en el que te encuentres, siempre recuperará los cambios que hiciste en el lugar que lo llamas.

Para recuperar los últimos cambios desde el stash a tu staging area utiliza el comando:

- git stash pop

Para aplicar los cambios de un stash específico y eliminarlo del stash:

- git stash pop stash@{<num_stash>}

Para retomar los cambios de una posición específica del Stash puedes utilizar el comando:

- git stash apply stash@{<num_stash>}

Donde el <num_stash> lo obtienes desde el git stash list

Listado de elementos en el stash

Para ver la lista de cambios guardados en Stash y así poder recuperarlos o hacer algo con ellos podemos utilizar el comando:
git stash list

Retomar los cambios de una posición específica del Stash || Aplica los cambios de un stash específico

Crear una rama con el stash

Para crear una rama y aplicar el stash más reciente podemos utilizar el comando

- git stash branch <nombre_de_la_rama>

Si deseas crear una rama y aplicar un stash específico (obtenido desde git stash list) puedes utilizar el comando:

- git stash branch nombre_de_rama stash@{<num_stash>}

Al utilizar estos comandos crearás una rama con el nombre <nombre_de_la_rama>, te pasarás a ella y tendrás el stash especificado en tu staging area.

Eliminar elementos del stash

Para eliminar los cambios más recientes dentro del stash (el elemento 0), podemos utilizar el comando:
git stash drop

Pero si en cambio conoces el índice del stash que quieres borrar (mediante git stash list) puedes utilizar el comando:
git stash drop stash@{<num_stash>}

Donde el <num_stash> es el índice del cambio guardado.
Si en cambio deseas eliminar todos los elementos del stash, puedes utilizar:
git stash clear

Consideraciones:

- El cambio más reciente (al crear un stash) SIEMPRE recibe el valor 0 y los que estaban antes aumentan su valor.
- Al crear un stash tomará los archivos que han sido modificados y eliminados. Para que tome un archivo creado es necesario agregarlo al Staging Area con git add [nombre_archivo] con la intención de que git tenga un seguimiento de ese archivo, o también utilizando el comando git stash -u (que guardará en el stash los archivos que no estén en el staging).
- Al aplicar un stash este no se elimina, es buena práctica eliminarlo.

Git Clean: limpiar tu proyecto de archivos no deseados

A veces creamos archivos cuando estamos realizando nuestro proyecto que realmente no forman parte de nuestro directorio de trabajo, que no se deberían agregar y lo sabemos.

- Para saber qué archivos vamos a borrar tecleamos git clean --dry-run

- Para borrar todos los archivos listados (que no son carpetas) tecleamos git clean -f

Git cherry-pick: traer commits viejos al head de un branch

Existe un mundo alternativo en el cual vamos avanzando en una rama pero necesitamos en master uno de esos avances de la rama, para eso utilizamos el comando

- git cherry-pick IDCommit.

cherry-pick es una mala práctica porque significa que estamos reconstruyendo la historia, usa cherry-pick con sabiduría. Si no sabes lo que estás haciendo ten mucho cuidado.

Reconstruir commits en Git con amend

A veces hacemos un commit, pero resulta que no queríamos mandarlo porque faltaba algo más. Utilizamos git commit --amend, amend en inglés es remendar y lo que hará es que los cambios que hicimos nos los agrega al commit anterior.

Git Reset y Reflog: úsese en caso de emergencia

¿Qué pasa cuando todo se rompe y no sabemos qué está pasando? Con git reset HashDelHEAD nos devolveremos al estado en que el proyecto funcionaba.

- git reset --soft HashDelHEAD te mantiene lo que tengas en staging.
- git reset --hard HashDelHEAD resetea absolutamente todo incluyendo lo que tengas en staging.

git reset es una mala práctica, no deberías usarlo en ningún momento; debe ser nuestro último recurso.

Git nunca olvida, git reflog

Git guarda todos los cambios aunque decidas borrarlos, al borrar un cambio lo que estás haciendo sólo es actualizar la punta del branch, para gestionar estas puntas existe un mecanismo llamado registros de referencia o reflogs.
.
La gestión de estos cambios es mediante los hashes de referencia (o ref) que son apuntadores a los commits. Los recoges registran cuándo se actualizaron las referencias de Git en el repositorio local (sólo en el local), por lo que si deseas ver cómo has modificado la historia puedes utilizar el comando:

- git reflog

Muchos comandos de Git aceptan un parámetro para especificar una referencia o “ref”, que es un puntero a una confirmación sobre todo los comandos:

- git checkout Puedes moverte sin realizar ningún cambio al commit exacto de la ref

git checkout eff544f

- git reset: Hará que el último commit sea el pasado por la ref, usar este comando sólo si sabes exactamente qué estás haciendo

- git reset --hard eff544f # Perderá todo lo que se encuentra en staging y en el Working directory y se moverá el head al commit eff544f
- git reset --soft eff544f # Te recuperará todos los cambios que tengas diferentes al commit eff544f, los agregará al staging area y moverá el head al commit eff544f
- git merge: Puedes hacer merge de un commit en específico, funciona igual que con una branch, pero te hace el merge del estado específico del commit mandado

git checkout master

- git merge eff544f # Fusionará en un nuevo commit la historia de master con el momento específico en el que vive eff544f

Buscar en archivos y commits de Git con Grep y log

A medida que nuestro proyecto se hace grande vamos a querer buscar ciertas cosas.
Por ejemplo: ¿cuántas veces en nuestro proyecto utilizamos la palabra color?
Para buscar utilizamos el comando git grep color y nos buscará en todo el proyecto los archivos en donde está la palabra color.

- Con git grep -n color nos saldrá un output el cual nos dirá en qué línea está lo que estamos buscando.
- Con git grep -c color nos saldrá un output el cual nos dirá cuántas veces se repite esa palabra y en qué archivo.
- Si queremos buscar cuántas veces utilizamos un atributo de HTML lo hacemos con git grep -c "<p>".

git grep color -->use la palabra color
git grep la --> donde use la palabra la

git grep -n color–> en que lineas use la palabra color
git grep -n platzi --> en que lineas use la palabra platzi
git grep -c la --> cuantas veces use la palabra la
git grep -c paltzi --> cuantas veces use la palabra platzi
git grep -c “<p>”–> cuantas veces use la etiqueta <p>
grep–> para los archivos
log --> para los commits.
Use el git log -S “palabra” no trae el conteo, trae las palabras parecidas a las que estamos buscando.
git log --all --oneline | grep “cabecera”, te trae los commits en donde se encuentra la palabra.

Comandos y recursos colaborativos en Git y GitHub

- git shortlog -sn = muestra cuántos commit han hecho cada miembro del equipo.
- git shortlog -sn --all = muestra cuántos commit han hecho cada miembros del equipo hasta los que han sido eliminado
- git shortlog -sn --all --no-merge = muestra cuántos commit han hecho cada miembros quitando los eliminados sin los merges
- git blame ARCHIVO = muestra quién hizo cada cosa linea por linea
- git COMANDO --help = muestra cómo funciona el comando.
- git blame ARCHIVO -Llinea_inicial,línea final= muestra quién hizo cada cosa linea por linea indicando desde que linea ver ejemplo -L35,50
- **git branch -r **= se muestran todas las ramas remotas
- git branch -a = se muestran todas las ramas tanto locales como remotas