Curso de git:

COMANDOS DE GIT UTILIZADOS AL MOMENTO

**$git init

**Inicia un repositorio Git en el directorio actual.

**$git add

**Añade al stage area la nueva modificación.

$git commit -m "un mensaje"

Añade la nueva versión al repositorio.

$git status

Muestra el estado del repositorio para dicha carpeta, especificando qué archivos han r ecibido modificaciones y cuáles necesitan guardarse.

**$git log **

Muestra el historial de commits realizados a dicho archivo.

**$git show **

Muestra las diferencias entre la versión actual (HEAD) y la versión anterior.

$git diff Index1 Index2

Muestra las diferencias entre la version del Index1 y la versión del Index2

$git reset ID --hard

TODO vuelve a

la versión de dicho ID, borrando el resto. Ahora el HEAD será esa nueva versión.

$git reset ID --soft

Vuelve a la versión de dicho ID pero lo de

staging ahí permanece para su próximo commit. En el directorio es donde se vuelve a la versión anterior.

$git reset HEAD

Este es el comando para sacar archivos del área de Staging.

$git diff

Muestra las diferencias entre lo que está en staging

y como lo tengo en mi directorio actual.

$git log --stat

Me muestra más información sobre los cambios.

$git checkout ID archivo.txt

Ahora en el directorio actual estaré viendo dicha versión.

$git checkout master archivo.txt

Para volver al archivo más nuevo.

$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 reset y git rm
son comandos con utilidades muy diferentes, pero aún así se confunden muy fácilmen te.
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 t iempo” 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 indicarle a
Git cómo eliminar los archivos que ya no necesitamos en la última versión del proyec to:

- git rm --cached: Elimina los archivos de nuestro repositorio local y del área de staging, pero los mantiene en nuestro disco duro. Básicamente le dice a Git que deje de trackear el historial de cambios de estos archivos, por lo que pasaran a un estado untracked.
- 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 avan zados).

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. Recue rda 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.

¡Pero todavía falta algo!

- 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 n o 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 mant endrá 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.

Mis anotaciones

#To config the user
$ git config --global user.name “johan7perez”
$ git config --global user.email "johan7perez@gmail.com"
#To see the current config
$ git config --list

#To see where the git configuration is stored
$ git config --list --show-origin
#To start a new empty project with git
$ git init
#To see the status of the current project

$ git status
#To add a new file/modificated file to the staging area
$ git add
#To remove a new file/modificated file from the staging area (cached, remove the file from the ram memory also)
$ git rm --cached
#To see the history of a file
$ git show
#To show the log of the commit or a file, stat is to see the changes at a bytes level
$ git log [–stat]
#To confirm the changes that we have in the staging area
$ git commit -m “Message”
#To see the differences of two files / if we do not pass any flag it will compare the current directory and the staging area
$ git diff [commit file code] [another commit file code]
#To back in the time, if it’s hard will delete everithing made, but if it’s not, will treat the file as a modificated/untracked file
$ git reset [commit code] [–hard]
#To see back in the pass and see the status of a file / for leave the file in the original state just use ‘git checkout master’
$ git checkout [commit code] [file name]

Ramas
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.
git branch -nombre de la rama-: Con este comando se crea una nueva rama.
git checkout -nombre de la rama-: Con este comando puedes saltar de
una rama a otra.
git checkout -b rama: Crea una rama y nos mueve a ella automáticamente, Es decir, es la combinación de git brach y git checkout al mismo tiempo.

git reset id-commit: Nos lleva a cualquier commit
no importa la rama, ya que identificamos el id del tag., eliminando el historial de los commit posteriores al tag seleccionado.
git checkout rama-o-id-commit: Nos lleva a cualquier commit sin borrar los commit posteriores al tag seleccionado.

Comandos a ser utilizados : para conectar tu git con github

git remote add origin URL -Comando para guardar la url del repositorio de github
git remote -v -Comando para verificar que la URL se haya guardado corretamente, nos deberia traer la url donde hariamos el fecht y el push
git pull origin master --alow-unrelated-histories -Comando que trae la versión del repositorio remoto y hacer merge para crear un commit con los archivos de ambas partes.
git push origin master - Comando para enviar los cambios del repositorio local de git al repositorio remoto de github.com

llaves publicas y privadas:

SSH o Secure Shell: Es un protocolo de red que permite acceso remoto seguro a través de una conexión encriptada. Este método de autenticación requiere un passphrase (contraseña) o tambien puede funcionar sin passphrase sobre la clave.
En el directorio Home ~

$ git config -l: Muestra la configuración dentro de Git(user y email), Este comando funciona por que Git esta instalado en todo el equipo local.

$ git config --global user.email "nombre_email_cambiado" : Se puede utilizar este mismo comando para cambiar el email.

Creamos la llave SSH

$ ssh-keygen -t rsa -**b** 4096 -C "tu_email@gmail.com"
1.-t = Especifica cual es el algoritmo que vamos a usar para crear esa llave.
2.rsa = Algoritmo a usar, hasta el momento el mas popular.
3.-b = Especifica que tan compleja es la llave.
4.4096 = Complejidad de la llave desde una perspectiva matemática.
5.-C = Indica a que correo electrónico va estar conectado esta llave
6."tu_email@gmail.com" = Correo electrónico.
.
Dato: Guardar la llave en la dirección predeterminada.

Dato2: Passphrase: Password con espacios o Contraseña adicional de texto que le vas a poner a tu llave pública y privada.

.

1er Paso:

Una ves que tengamos la llave, tenemos que agregarlo al entorno, y el entorno es básicamente que el sistema operativo donde tu trabajas sepa que la llave existe. Para ello ejecutamos lo sgte:

.

revisar el servidor de llaves / Evalúa que un comando se dispare.

$ eval $(ssh-agent -s)
Dato: Agent pid 4724

1.Agent = Significa que el servidor de SHH esta corriendo.
2.pid = Process id o identificador del proceso.
3.4724 = Número que al sistema operativo le dice que el proceso esta corriendo.

.

2do Paso:

Agregamos la llave privada a nuestro sistema o al servidor por que no basta con que la llave solo exista, sino debemos decirle que existe. Para ello ejecutamos el siguiente comando:

$ ssh-add ~/.ssh/id_rsa
1.~ = Home
2..ssh = carpeta ssh
3.id_rsa = llave privada la que nunca debemos de mostrar.

Conexión a GitHub con SSH

Importante aclarar: cada usuario, cada computadora tiene que tener una llave única conectadas con el repositorio o no funcionara. No es buena idea compartir las llaves de una computadora a otra porque pueden ser interceptadas.

Para añadir la llave vamos a Github en la opción **Settings** y luego a la opción **SSH and GPG Keys** en el titulo ponemos la descripción de cual computador será y en el recuadro de abajo pegamos la llave, le damos agregar y ponemos la contraseña de nuestra cuenta Github. De esta forma ya tendríamos conectado nuestro repositorio de github con la llave publica del computador de donde hayamos generado las llaves, luego vamos a nuestros repositorios, la opción **clone or download (Code)** y luego damos click al boton **Use SSH** y copiamos esa url para setearla como el repositorio al cual nos conectaremos.

git remote set-url origin ExampleSSHurl Esto nos permite cambiar la url de origin (del repositorio origin). Podemos confirmar que ya cambio usando nuevamente **git remote-v**

git pull origin master Con este le decimos que traiga de origin (mi repositorio remoto) y lo vamos a fusionar con la rama master.

git push origin master Este nos permite enviar commits que hicimos en master al repo origin

tags y manejo de ramas

git log --all: Muestra toda la historia de commits de nuestro proyecto
git log --all --graph: Además de mostrar la historia de commits muestra gráficamente las ramas
git log --all --graph --decorate --oneline: Muestra la historia de commits y ramas de

manera compacta
git tag -a nombre-del-tag -m “Mensaje del tag” id-del-commit: Creamos un tag y asignarlo a un commit
git tag: Muestra la lista de todos los tag
git show-ref --tags: Muestra la lista de tags y a que commits están asignados
git push origin --tags: Envía un tag a un repositorio remoto
git tag -d nombre-del-tag: Borra un tag en el repositorio local
git push origin :refs/tags/nombre-del-tag: Borra un tag en el repositorio remoto

Aprendido: Como enviar y borrar ramas a github.
#1 creamos una rama (git branch name_branch)

#2 Enviamos la rama a github (git push origin name_branch)

#3 Para eliminar la rama localmente (git branch -d name_branch)

#4 Para eliminar la rama en github (git push --delete origin name_branch)

Nuevos commit para manejar ramas:

•Para ver las ramas existentes (git show-branch --all)
•git show-branch: muestra cuales son las ramas que existen y cual ha sido su historia.

•git show-branch --all: muestra algo muy similar a git show-branch pero con mas datos.

•gitk: abre en un software con la historia de una manera visual ahí se pueden ver los cambios que se han hecho, da una idea de como esta estructurado el proyecto en cabecera, master, etc.

Pull request:

Un pull request es un estado intermedio antes de enviar el merge.

•El pull request permite que otros miembros del equipo revisen el código y así aprobar el merge a la rama.
•Permite a las personas que no forman el equipo , trabajar y colaborar con una rama.
•La persona que tiene la responsabilidad de aceptar los pull request y hacer los merge tienen un perfil especial y son llamados DevOps

NADIE que sea colaborador debe hacer push a master, para hacer reflejar los cambios, se debe hacer Pull request y un colaborador o el mismo propietario del proyecto, hará code review y posteriormente autorizará hacer el merge, lo usual y lo más profesional es que haya un perfil denominado DevOps, que será encargado de hacer estas revisiones y hacer la vida del programador más fácil. El pull request es un estado intermedio antes de hacer el merge.
Como buena práctica, se debería tener una rama de staging o development para probar todas las funcionalidades que se estén evaluando, una vez aprobados estos cambios y se está conforme con el producto final, se realiza un merge con la rama master, nuevamente, lo ideal NUNCA es trabajar sobre master, sino sobre las ramas.

Pull request:

Es una funcionalidad de github (en gitlab llamada merge request y en bitbucket push request), en la que un colaborador pide que revisen sus cambios antes de hacer merge a una rama, normalmente master.

Al hacer un pull request se genera una conversación que pueden seguir los demás usuarios del repositorio, así como autorizar y rechazar los cambios.

El flujo del pull request es el siguiente

1.Se trabaja en una **rama paralela** los cambios que se desean (git checkout -b <rama>)
2.Se hace un **commit** a la rama (git commit -am '<Comentario>')
3.Se **suben** al **remoto** los **cambios** (git push origin <rama>)
4.En GitHub se hace el pull request comparando la **rama master** con la rama del **fix**.
5.Uno, o varios colaboradores revisan que el **código sea correcto** y dan **feedback** (en el chat del pull request)
6.El colaborador hace los cambios que desea en la **rama** y lo **vuelve a subir** al remoto (automáticamente jala la historia de los cambios que se hagan en la rama, en remoto)
7.Se **aceptan los cambios** en GitHub
8.Se hace **merge** a master desde GitHub
Importante: Cuando se modifica una rama, también se modifica el pull 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 poryecto en GitHub, te conviertes en dueñ@ del repositorio fork, puedes trabajar en éste 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 contribuír 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 ésto 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

Ignorar archivos para no subirlos al repositorio (.gitignore)

Por diversas razones, **no todos los archivos** que agregas a un proyecto **deberían guardarse** en un repositorio, ésto porque hay archivos que no todo el mundo debería de ver, y hay archivos que al estar en el repositorio alentan el proceso de desarrollo (por ejemplo los binary large objects, blob, que se tardan en descargarse).

.

Para que no se suban estos archivos no deseados se puede crear un archivo con el nombre .gitignore en la raíz del repositorio con las reglas para los archivos que no se deberían subir (ver [sintaxis de los .gitignore(https://git-scm.com/docs/gitignore)).

.

Las razones principales para tomar la decisión de no agregar un archivo a un repositorio son:

.

•Es un archivo con contraseñas (normalmente con la extensión .env)
•Es un blob (binary large object, objeto binario grande), mismos que son difíciles de gestionar en git.
•Son archivos que se generan corriendo comandos, por ejemplo la carpeta node_modules que genera npm al correr el comando npm install

Readme.md y markdown

README.md es el lugar dónde se explica de qué trata el proyecto, cómo utilizarlo y demás información que se considere que se deba conocer antes de utilizar un proyecto.

.

Los archivos README son escritos en un lenguaje llamado **markdown**, por eso la extensión .md, mismo que es un estándar de escritura en diversos sitios (como platzi, como wikipedia y obvio GitHub), ver reglas de markdown.

.

Los README.md pueden estar en todas las carpetas, pero el más importante es el que se encuentra en la raíz y ayudan a que los colaboradores sepan información importante del proyecto, módulo o sección, puedes crear cualquier cualquier archivo con la extensión .md pero sólo losn README.md los mostrará por defecto GitHub.

Tu sitio web público con GitHub Pages

Este es un sitio para nuestros proyectos donde lo único que tenemos que hacer es tener un repositorio hosteado.

En la pagina podemos seguir las instrucciones para crear este repo, tomar la llave SSH y hacer un **git clone #SSHexample** en mi maquina local (Home).

Luego accederemos a la carpeta nueva que aparece en nuestra maquina local, seguido de esto crearemos un nuevo archivo que se llame **index.html**

Guardamos los cambios, hacemos un git pull y seguido de esto un git push a master y luego vamos a las opcioens de settings de este repositorio y en la parte de abajo en la columna github pages configuramos source para que traiga la rama master y guardamos lo cambios.

Despues de esto podremos ver nuestro trabajo en la web como si tuviéramos nuestro propio servidor.

Git rebase:

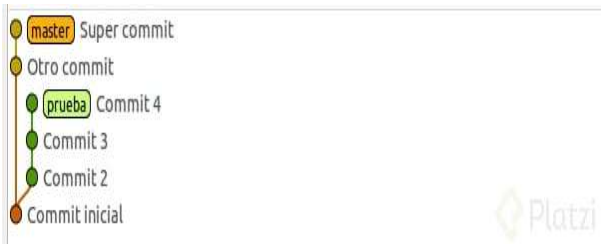
En git existen dos formas que nos permiten unir ramas, git merge y git rebase. La forma mas conocida es git merge, la cual realiza una fusión a tres bandas entre las dos últimas instantáneas de cada rama y el ancestro común a ambas, creando un nuevo commit con los cambios mezclados.

Git rebase básicamente lo que hace es recopilar uno a uno los cambios confirmados en una rama, y reaplicarlos sobre otra. Utilizar rebase nos puede ayudar a evitar conflictos **siempre que se aplique sobre commits que están en local y no han sido subidos a ningún repositorio remoto.** Si no tienen cuidado con esto último y algún compañero utiliza cambios afectados, seguro que tendrá problemas ya que este tipo de conflictos normalmente son difíciles de reparar.

.

Punto de partida

.



.

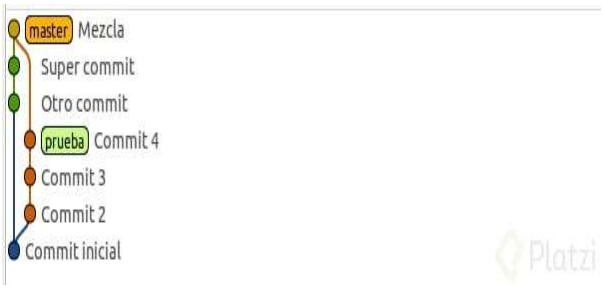
Ejempo de mezcla de ramas con git merge

.

git checkout master

git merge prueba

.



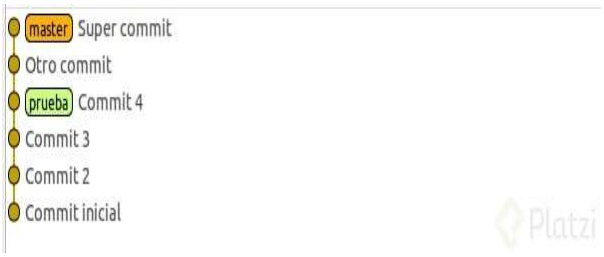
.

Ejempo de mezcla de ramas con git rebase

.

git checkout master
git rebase prueba

.



.

Utilizando git rebase conseguimos un log de commit mas sencillo y como veis es muy fácil utilizarlo, pero como he comentado antes, cuidado con usarlo con commits que están en servidores remotos.

Rebase

•Se sugiere usar rebase de manera LOCAL
•Reescribe la historia del repositorio
•Cambia la historia de dónde comenzó el branch
•Primero se le hace rebase a la rama que voy a desaparecer de la historia y luego se le hace rebase a la rama principal

-> (en experimento) git rebase master
-> (en master) git rebase experimento

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 elelmentos 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 desden 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 mas 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.

Limpiar el Repositorio

Mientras estamos trabajando en un repositorio podemos añadir archivos a él, que realmente no forma parte de nuestro directorio de trabajo, archivos que no se deberían de agregar al repositorio remoto.

El comando clean actúa en archivos sin seguimiento, este tipo de archivos son aquellos que se encuentran en el directorio de trabajo, pero que aún no se han añadido al índice de seguimiento de repositorio con el comando add.

$ git clean
La ejecución del comando predeterminado puede producir un error. La configuración global de Git obliga a usar la opción force con el comando para que sea efectivo. Se trata de un importante mecanismo de seguridad ya que este comando no se puede deshacer.

→ Revisar que archivos no tienen seguimiento.

$ git clean --dry-run
→ Eliminar los archivos listados de no seguimiento.

$ git clean -f

Cherry Pick

Este comando permite coger uno o varios commits de otra rama sin tener que hacer un merge completo. Así, gracias a cherry-pick, podríamos aplicar los commits relacionados con nuestra funcionalidad de Facebook en nuestra rama master sin necesidad de hacer un merge.

Para demostrar cómo utilizar git cherry-pick, supongamos que tenemos un repositorio con el siguiente estado de rama:

a - **b** - c - d Master
\
e - f - g Feature
El uso de git cherry-pick es sencillo y se puede ejecutar de la siguiente manera:

git checkout **master**
En este ejemplo, commitSha es una referencia de confirmación. Puedes encontrar una referencia de confirmación utilizando el comando git log. En este caso, imaginemos que queremos utilizar la confirmación ‘f’ en la rama master.

Para ello, primero debemos asegurarnos de que estamos trabajando con esa rama master.

git cherry-pick f
Una vez ejecutado, el historial de Git se verá así:

a - **b** - c - d - f Master
\

e - f - g Feature
La confirmación f se ha sido introducido con éxito en la rama de funcionalidad

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

Existe un mundo donde vamos avanzando en una rama pero necesitamos en master uno de esos avances de la rama para esto se usa un comando llamado cherry pick.

Para esto primero hacemos un **git log --oneline** y tomo el tag del commit que quiero traer (esto se hace en la rama donde esta ese commit), volvemos a master y desde master ejecutamos git cherry-pick #examplecommittag, damos 'Enter' y esto nos trae los cambios precisos de ese commit que estaba en otra rama a la rama master. (Esto es traerme un commit viejo de otra rama a mi rama master)

Si queremos fusionar los cambios hacemos un merge (parados en master la rama donde vamos a hacer la fusión hacemos **git merge #examplebranch"**)

Cherry-pick es una mala practica porque significa que estamos reconstruyendo la historia, es mucho mejor hacer el trabajo duro haciéndolo con un merge.

Git amend

el commit --amend es muy util, pero hay que tener cuidado en algunos casos, como en el caso de que el commit que quieras enmendar lo hayas pusheado al repositorio remoto, entonces quieras enmendar un commit que esta en remoto.

Así como en el caso de cherry-pick y rebase, hay que usarlo con cuidado porque modificará la historia de tu repositorio.

Digamos que haces un cambio al archivo a.txt y haces un commit.

Luego subes ese commit al repositorio haciendo push.

Pero se te olvido agregar cambios a ese commit y quieres enmendarlo.

Haces un git --amend y en la historia de tu repositorio local, pareciera que no ha pasado nada: enmendaste tu commit.

Pero resulta que en el repositorio remoto eso no ha ocurrido, ese git --amend no tuvo lugar en el repositorio remoto, y al hacer git status te mostrará un error así:

Your branch **and** 'origin/master' have diverged,
and have 1 **and** 1 different commits **each**, respectively.
(use "git pull" to merge **the** remote branch **into** yours)
y al momento de hacer push para sobreescribirlo, te aparecerá este error:

! [rejected] master -> master (non-fast-forward)
error: failed to push some refs to 'git@github.com:luisxxor/hyperblog-1.git'
hint: Updates were rejected because **the** tip **of** your current branch is behind
hint: its remote counterpart. Integrate **the** remote changes (e.g.
hint: 'git pull ...') **before** pushing again.
hint: See **the** 'Note about fast-forwards' **in** 'git push --help' **for** details.

Y tendrás que hacer git pull para mergear los cambios de tu repositorio remoto y finalmente hacer push. Es decir, se puede hacer, pero sería contraproducente, porque la idea del amend era enmendar un sólo commit y no generar commits adicionales, pero el resultado de continuar haciendolo en éste caso es que tienes:

1.El primer commit
2.El commit enmendado
3.Y el commit del merge

tl:dr

Es una mala práctica enmendar un commit que ya ha sido pusheado al repositorio remoto.

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 éstas puntas existe un mecanismo llamado registros de referencia o reflogs.

.

La gestión de estos cambios es mediante los hash'es 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>".

NOTAS CLASE:

•**git shortlog**: Ver cuantos commits a hecho los miembros del equipo
•**git shortlog -sn**: Las personas que han hecho ciertos commits
•**git shortlog -sn --all**: Todos los commits (también los borrados)
•**git shortlog -sn --all --no-merges**: muestra las estadisticas de los comigs del repositorio donde estoy
•**git config --global alias.stats "shortlog -sn --all --no-merges"**: configura el comando "shortlog -sn --all --no-merges" en un Alias en las configuraciones globales de git del pc
•**git blame -c blogpost.html**: Muestra quien ha hecho cambios en dicho archivo identado
•**git blame --help**: Muestra en el navegador el uso del comando
•**git blame archivo -L 35, 60 -c**: Muestra quien escribio el codigo con informacion de la linea 35 a la 60, EJ: git blame css/estilos.css -L 35, 60 -c
•**git branch -r**: Muestra las Ramas remotas de GitHub
•**git branch -a**: Muestra todas las Ramas del repo y remotas de GitHub