Git

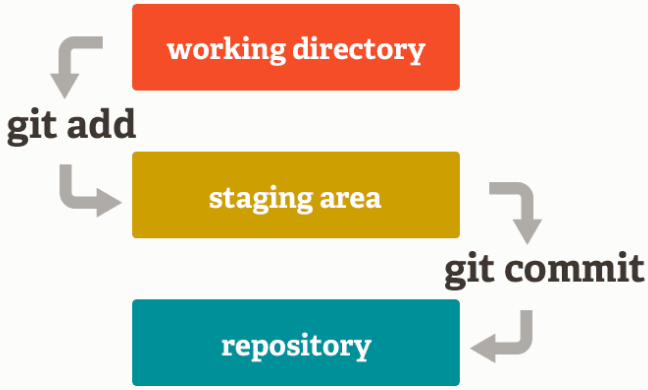
Git es un sistema de control de versiones distribuido libre y de código abierto diseñado para manejar todo, desde proyectos pequeños hasta muy grandes, con velocidad y eficiencia.

Conceptos:

Staging: Área en memoria RAM de preparación.

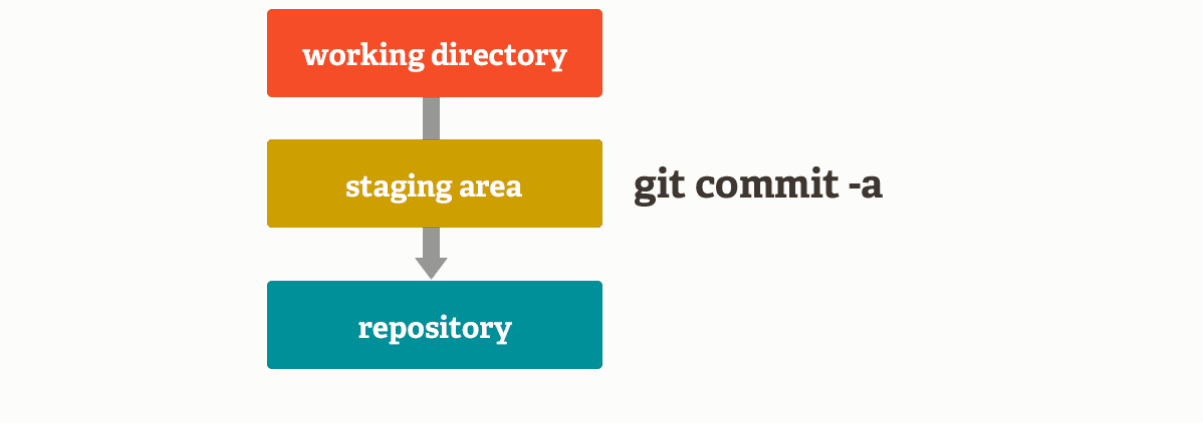
A diferencia de los otros sistemas, Git tiene algo llamado "área de ensayo" o "índice". Esta es un área intermedia donde los commits pueden formatearse y revisarse antes de completar el commit.

Una cosa que diferencia a Git de otras herramientas es que es posible organizar rápidamente algunos de sus archivos y confirmarlos sin comprometer todos los demás archivos modificados en su directorio de trabajo o tener que enumerarlos en la línea de comandos durante la confirmación.



Esto le permite organizar solo partes de un archivo modificado. Atrás quedaron los días de hacer dos modificaciones lógicamente no relacionadas a un archivo antes de que te dieras cuenta de que olvidaste cometer una de ellas. Ahora puede simplemente organizar el cambio que necesita para el compromiso actual y el otro cambio para el próximo compromiso. Esta característica escala hasta tantos cambios diferentes a su archivo como sea necesario.

Por supuesto, Git también facilita ignorar esta función si no desea ese tipo de control: simplemente agregue una '-a' a su comando de confirmación para agregar todos los cambios a todos los archivos en el área de preparación.



Repositorio: espacio en disco donde se guardan las versiones del aplicativo.

Comandos Git

Inicializando:

git init: Para iniciar el sistema de git y configurar parámetros necesarios.
git config --list: Veo las configuraciones de mi git
git config --global user.name *"nombre"*
git config --global user.email *"correo"*
git config --global core.editor nano

git status: Me muestra si los archivos están en Staging o todavía faltan subirlos..
git add: Agrego los archivos a Staging y quedan listos para el Commit.
git add . :Agrego todos los archivos.
git commit: Subo los archivos al repositorio de git y creo una version nueva.

- git commit -m "mensaje"

git log: muestra la lista de los commit.

- git log --stat: muestra los cambios específicos
- git log -p: muestra las diferencias introducidas en cada confirmación.

git show: muestra los últimos cambios sobre un archivo.
git diff: compara lo que hay en tu directorio de trabajo con lo que hay en tu área de preparación. El resultado te indica los cambios que has hecho y que todavía no has preparado. Puedo ver los cambios entre un commit y otro de un archivo. Muestra los cambios que todavía no están preparados.
git rm --cached: Elimina los archivos del área de Staging y del próximo commit, pero los mantiene en el disco duro.
git rm --force: Elimina los archivos de git y del disco duro.

****En caso de eliminar un archivo****
Para conservar los datos de los otros archivos debería dejarlos en staging y luego hacer
git reset --soft <commit>: Vuelvo al commit donde estaba el archivo

git checkout HEAD <archivo>: Saco el archivo borrado del staging.
git checkout <archivo>: recupero el archivo borrado

git commit para agregar los archivos de staging

En caso que solo lo haya borrado pero no haya hecho commit
git reset <commit>: Vuelvo a un commit anterior donde estaba el archivo
git restore <archivo>: Recupero archivo.

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 agregará al commit anterior.

- git commit -m “Primer commit”
- git add index.html (me olvidé de mandar este archivo en el commit)
- git commit --amend (Se abrirá el editor para agregar algo mas al mensaje del commit)

git reset HEAD <archivo>: sacar un archivo del área de preparación
git reset: Me permite volver en el tiempo a otra versión.

- **git reset *n° de commit* --hard:** cambia todo y vuelve a la versión del commit indicado.
- **git reset *n° de commit* --soft:** Vuelve al commit indicado pero mantiene lo que hay en staging. No toca el archivo de índice o el árbol de trabajo en absoluto (pero reinicia la cabeza <commit>, al igual que todos los modos). Esto deja todos los archivos modificados como "Cambios para confirmar", como lo git status diría.

git checkout: traer cambios a tu carpeta.

- **git checkout *n° de commit nombre de archivo***: trae el archivo elegido del commit a una rama HEAD distinta.
- **git switch -c *nombre de rama*:** paso el commit que use con checkout a una nueva rama para seguir trabajando sobre el mismo

Ramas
git branch *nombre de la rama:* Crea una rama nueva distinta a Master con el último commit de master.
git checkout *nombre de la rama:* Para moverme a una rama en particular.

git merge: une ramas. Debo estar en master y desde allí hacer merge trayendo la rama que quiero fusionar.
Puede haber conflictos cuando se hace el merge si se modificaron las lineas iguales en ambas ramas.

Para ver el historial de todas las ramas:
git show-branch --all

Mostrar ramas de forma visual:
gitk

Historial gráfico
git log --all --graph --decorate --oneline
Me muestra reducido y gráfico los movimientos de los commits y las ramas.

Alias
Para crear un alias que ejecute el comando:
alias arbolito= "git log --all --graph --decorate --oneline"

GITHUB
Readme: Un archivo que explica que es el repositorio.

Agregar el origen del repositorio de github
git remote add origin *ruta de github donde subiré mi repositorio local*

git remote -v: muestra el origen para enviar y recibir el repositorio
git push origin master: envió la rama master al repositorio de github
git pull origin master: Traigo la actualización, si la hubiera, del repositorio de Github. Es buena práctica.
git pull origin master --allow-unrelated-histories: Para fusionar las historias de los commit.

Llaves públicas y privadas
En mi entorno creo la llave privada y pública y envió la llave pública a Github.
Las llaves deben crearse en el home.

Crear llaves:
ssh-keygen -t rsa -b 4096 -C *"correo electrónico con el que nos conectamos a github"*

Copiar llave pública a github.

Para ver si el servidor de llaves está funcionando:
eval $(ssh-agent-s)

Agregar mi llave:

ssh-add ~/.ssh/id_rsa

Para cambiar la url del repositorio origin por la ssh en el repositorio del proyecto:
git remote set-url origin *dirección de github ssh*

Hacer pull después de cambiar url del repositorio.

TAGS
Para colocar un tag en un commit particular:
git tag -a v0.1 -m “mensaje del tag” *n° del commit*
Generalmente se manejan los release o versiones.

Para ver los tags de los commits
git show-ref --tags

Enviar Tags a Github
git push origin --tags

Borrar un tag en Git
git tag -d *nombre del tag*

Borrar Tags en Github
git push origin :refs/tags/*nombre del tag*

Ramas en Github
Subir rama de git a Github:
git push origin *rama a enviar*

Trabajo colaborativo
Si el repositorio es público cualquiera puede clonarlo, pero para subir los cambios con push se necesita agregar al usuario al repositorio.
Vamos a los settings del repositorio->Manage access y agrego un colaborador. La persona debe tener una cuenta en Github.
Le llegará un correo para que acepte la invitación.

Forks
Entro al repo de github que quiero colaborar y hago clic en fork así clono el proyecto en mi Github.
Luego hago Clone para traermelo a mi PC

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
```

README.md
https://pandao.github.io/editor.md/en.html

Github pages

https://pages.github.com/
Para que cargue en la raiz debo renombrar el repositorio por el nombre de usuario.

Rebase
rebase reescribe la historia del repositorio, cambia la historia de donde comenzó la rama y **solo** debe ser usado de manera local.
Ejemplo:
rama experimental quiero fusionar con Master
estando en la rama experimental hago:
git rebase master

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 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 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 solo detecta archivos nuevos, no es necesario que se trate de una copia de otro archivo, suficiente con que sea un archivo nuevo que ustedes hayan creado.

****git clean --dry-run** **
Simula la eliminación de archivos, ¿tienes dudas de que archivos eliminará?, ejecuta git clean --dry-run, y cuando estes seguro de que desea eliminarlos, ejecutas **git clean -f**

Archivos que se hayan modificado o editados git clean no interviene aquí.

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.

En la rama Master traigo un commit de otra rama
git cherry-pick *hash del commit*

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 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.

Buscar en Git
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>

git log -S "cabecera" --> cuantas veces use la palabra cabecera en todos los commits.

grep–> para los archivos
log --> para los commits.

git shortlog -sn = muestra cuantos commit han hecho cada miembros del equipo.
git shortlog -sn --all = muestra cuantos commit han hecho cada miembros del equipo hasta los que han sido eliminado
g***it shortlog -sn --all --no-merge*** = muestra cuantos commit han hecho cada miembros quitando los eliminados sin los merges
git blame ARCHIVO = muestra quien hizo cada cosa linea por linea
git COMANDO --help = muestra como funciona el comando.
git blame ARCHIVO -Llinea_inicial,linea_final= muestra quien hizo cada cosa linea por linea indicándole 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