GUÍA RÁPIDA GIT – quick reference http://gitref.org/

ESTRUCTURA / ESQUEMA DE CAPAS

Un repositorio git está formado por:

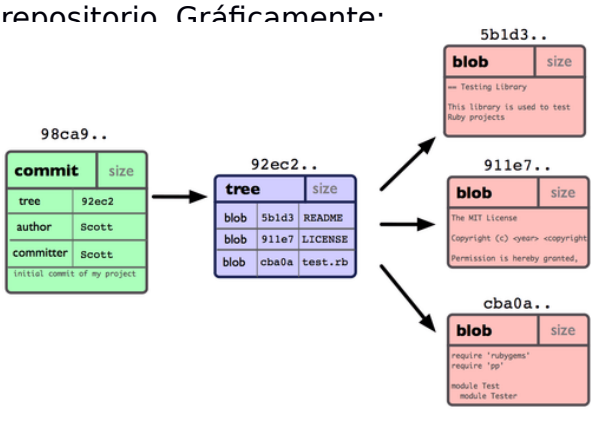
DIRECTORIO LOCAL / Working directory .- Donde el programador tiene su copia de código actual en la que está trabajando

HISTORY .- Es un espacio mantenido automáticamente donde se almacenan de forma íntegra y con carácter permanente los snapshots o las sucesivas versiones del proyecto

STAGING AREA / ÍNDICE.- Es un área de trabajo intermedia entre el directorio de trabajo y el history. Digamos que es una zona de aculación de cambios, donde se van almacenando temporalmente los cambios del directorio de trabajo respecto del history, para que cuando se grabe una nueva versión, se haga de golpe (con todos los ficheros modificados) y no se genere una nueva versión por cada fichero modificado (se haría una versión de uno en uno). Además, esta área, permite flexibilizar cambios, para hacerlos o deshacerlos antes de que sea definitivo.

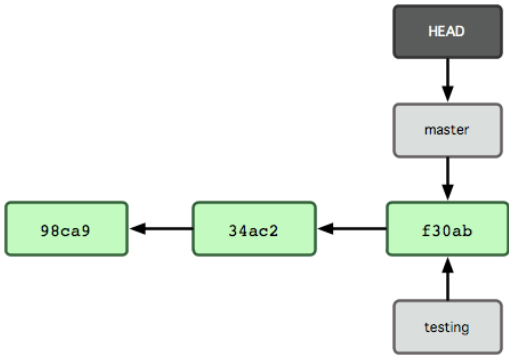
REMOTO .- Es un espacio de trabajo remoto opcional, donde a su vez, poder transmitir los cambios desde history, a un servidor de Internet.

Este es el esquema. Yo tengo 3 ficheros en mi working directory, hago add, y se calcula para ellos una firma SHA para cada elemento. De estas firmas, se obtiene otro checksum, que constituye un objeto tree, donde se almacenan las referencias a los ficheros contenidos en ése árbol y sus respectivas firmas. Eso sería, conceptualmente, una versión. Para poder tener varias versiones, hay que ir un paso más allá, e identificar al árbol, que a su vez referencia a los archivos y carpetas. Eso, se consigue mediante un objeto commit. El objeto commit, alberga una referencia a un árbol, tree o snapshot. Por cada commit que yo haga, este será el proceso, albergando cada commit, de forma indirecta, una versión íntegra del proyecto almacenado en el repositorio. Gráficamente:

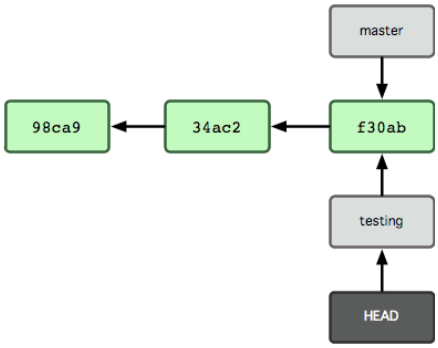


RAMA .- Si bien gracias a commit, tengo acceso a una versión del proyecto, ¿cómo puedo tener acceso a los diferentes commits o versiones del proyecto? Para ese fin, se crea el concepto de rama; que no es más que un "puntero" a un commit, y por tanto, a una versión concreta (snapshot) de nuestro proyecto.

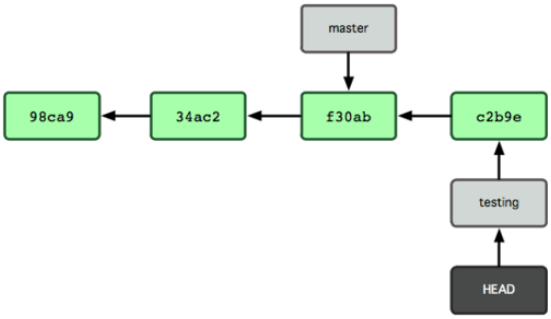
Una rama, por tanto, me permite acceder a una versión de un proyecto. ¿Cómo creo otra rama, para mantener el acceso a una versión distinta de un proyecto? Usando el git branch <nombre de la rama>. El objeto HEAD, apunta a la rama actual de trabajo. Con checkout <rama> muevo el HEAD donde necesite.



$ git branch testing

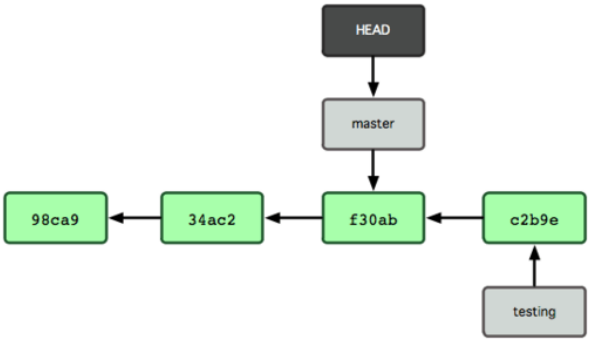


$ git checkout testing

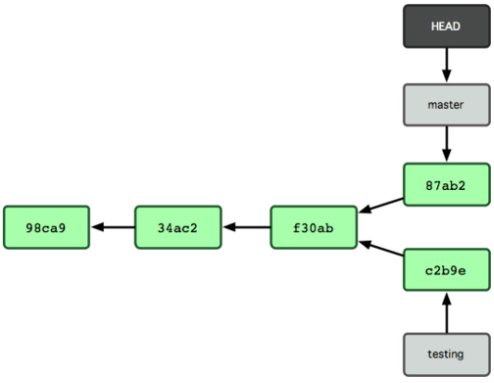


$ vim test.rb

$ git commit -a -m 'made a change'



$ git checkout master



$ vim test.rb

$ git commit -a -m 'made other changes'

MERGE y CONFLICTOS: Me creo una rama y me muevo a ella (git checkout –b nueva), toqueteo, hago los cambios en esa rama y me vuelvo a la master (checkout). Ahora, al hacer git merge nueva, desde la rama master, git incorpora automáticamente los cambios de nueva en master, del siguiente modo:

- Si en la versión de nueva, hay un fichero de menos, este no se elimina en el master
- Si en la versión nueva, hay un fichero de más, este da conflicto, aunque lo copia –es decir, aparece en el working directory de master, aunque conflictos-, por lo que simplemente salvando ese fichero haciendo commit en master, se resuelve
- Si en la versión de nueva, (rama alterada) hay un fichero con una línea de más o de menos, esa versión se asume como la buena y sobrescribe a la versión de ese mismo fichero en master, tal cual
- El problema viene cuando la misma parte (misma fila y columnas), del mismo fichero, se han hecho una editado de forma concurrente en distintas ramas. Esto, va a generar un conflicto al intentar mezclar las versiones que se traduce en que en el fichero en cuestión, va aparecer editado en el espacio de trabajo del contexto (rama) actual con unas marcas

```
<<<<<<< HEAD (rama actual)
Güuevonazo
=======
huevonazoS
>>>>>>> <rama_mezclada>
```

El conflicto aparece y <<<<< significa la rama actual ====== limita hasta donde llegan la versión del conflicto en la rama actual y >>>>> indica el final del conflicto. Para resolverlo, basta eliminar las líneas de <<<, de >>> y de ===, dejando el texto que realmente quieres y después, guardar el fichero. También, toca hacer commit de esos cambios.

```
MANDA HUEVOS
```

Al realizar el commit ya se hacen los cambios persistentes en la rama actual.

SÓLO HAY UNA STAGE AREA COMPARTIDA POR TODAS LAS RAMAS: por lo que si saltas de una versión a otra, que apunta a commits distintos, que tienen cambios sin confirmar, el checkout te dará error.

TRABAJO EN REMOTO .- Puedes compartir tu trabajo con un repositorio remoto. Tan simple como hacer git add. Una vez pase eso, en el espacio remoto, se habrá creado una copia de tu rama actual. Además, puedes establecer el mapping o la correspondencia entre la rama local y remota. Hay una copia de la rama remota en local. Si haces pull, lo que te intentaría mezclar esa rama-local/remota con tu head inmediatamente después de hacer un fetch, puede haber conflictos. Lo que se refleja en la rama master local. Para subsanarlos, igual que antes, editas, guardas, eliminas los conflictos y ya haces add (te obliga), commit (para que quede resuelto en local) y push, para machar el remoto.

LA MEZCLA DE CONFLICTOS, LA DEBO RESOLVER LOCALMENTE. PORQUE SI INTENTO HACER PUSH Y falla, no voy a poder subsanarlo, hasta que me descargue la última versión de la rama remota, mezcle y arregle.

INICIANDO PROYECTOS – getting and creating projects

COMANDO	EJEMPLO	COMENTARIOS
init	git init	Se crea una subcarpeta oculta (.git) desde el directorio actual que representan el repositorio. El contenido de esta carpeta, pasa a formar parte de los archivos en supervisión
clone	git clone [url] git clone git://github.com/val/simple.git	Crea una subcarpeta desde el directorio actual con el nombre del repositorio (simplegit en este caso) y dentro crea el oculto .git y copia todo el contenido que hubiera

COMANDOS BÁSICOS – basics commands

COMANDO	EJEMPLO	COMENTARIOS
add	git add . git add * git add file1 file2	Pasa los archivos indicados al área de intercambio o index – zona intermedia entre la versión definitiva y el directorio actual de trabajo- Con . toma todos los cambios en todas las subcarpetas (recursivo). Con * toma todos los archivos modificados del directorio actual (no recursivo). También puede expresarse explícitamente el nombre de los archivos separados por espacio
status	git status git status –s	Muestra los ficheros modificados entre el índice y el history (HEAD actual), los modificados entre el

		directorio de trabajo y el índice, y los modificados en el directorio de trabajo que no han sido subidos al índice. Con el parámetro –s (short), da la información abreviada. Por cada fichero modificado, aparecen dos columnas que preceden al nombre. La primera, indica el estado del fichero en el índice respecto al history y la segunda, indica el estado del fichero respecto al history y respecto al índice. El estado se representa con las letras A, M y D A.- Representa que es un nuevo fichero, no presente en el history aún, pero sí en el stage M.- Representa que existe una versión ya en el history, pero que ha sido modificada en el área de trabajo. Si aparece en la primera posición, indica que está indizada. Si no, indica que el cambio no está indizado D.- Representa que se ha eliminado un archivo desde ?.- Si el fichero solo existe en el área de trabajo y no ha sido indizado por primera vez Ejemplo 1:

		MM prueba.txt prueba.txt existía ya en el history, se ha modificado en el working directory y se ha indizado (M primera) Luego se ha modificar en el working directory, sin indizarse (M segunda) Ejemplo 2: AM prueba3.txt Prueba3.txt no existía en el history. Se ha creado en el directorio de trabajo y se ha indizado (A). Posteriormente, se ha modificado en el espacio de trabajo, sin pasarlo al índice (M) Ejemplo 3: D prueba4.txt Prueba4.txt existía en el history, pero se ha eliminado del directorio de trabajo
diff	git diff git diff - - staged git diff HEAD git diff - -stat	Muestra los cambios del área de trabajo NO indizados --staged Muestra los cambios SÍ indizados HEAD Muestra todos los cambios. Indizados y no Si pongo - - stat al final de una de las tres comibinaciones, se obtiene una información más resumida
commit	git commit –m “Mensaje obligado” git commit –am “Del tiron”	Guardo los cambios en el history desde el índice. Con –a, los cambios pueden pasar directamente del

		working directory

reset	git reset HEAD
	git reset [<mode>] [<commit>]
	git reset - - soft
	git reset - - hard
	
	`git reset --hard HEAD~2`
	
	`git reset --hard HEAD^`
		Deshace los cambios HEAD limpia el índice (lo contrario a add)
		Soft: vuelve el history al commit que le digas, dejando el stage área, los cambios deshechos, pero el working directory intacto
		Hard: vuelve el history al commit que le digas y deshace los cambios también en el directorio de trabajo con ^me refiero al Anterior en el History y con `~ al penúltimo`

rm	git rm <file>	Elimina el archivo del índice y del working directory NOTA: para eliminar del working directory y dejar el índice rm de Linux normal
stash	git stash git stash list git stash apply stash@{1} git stash pop stash@{n} git stash drop stash@{n} git stash clear	Saca los cambios tanto indizados como no y te deja el working directory en el último estado de commit. SU uso es apropiado ante una situación de emergencia, donde haya que trabjar sobre la versión del último cambio y dejar de lado temporalmente el desarrollo actual. Los manda a una zona llamada stack, identificando a ese conjunto de cambios por el commit por el que son sustituídos. Con list, miras el contenido de la pila Con Apply, lo sacas de la pila y te lo vuelves a madar al working directory Con pop, extraes de la pila Con drop, puedes

		borrar una tanda / elemento de los enviados a la pila mediante stash Con clear, se limpia la pila por completo
RAMAS Y MEZCLA DE VERSIONES – branching and merging		
COMANDO	**EJEMPLO**	**COMENTARIOS**
branch	git branch git branch -a git branch –v git branch –d <nombre_rama> git branch <nombre_rama> git branch –dr <nombre_rama>	Lista las ramas actuales. -a muestra todas. Si añado –v, lista las ramas y sus respectivos commits referenciados por cada una de ella. Con –d elimino una rama Al dar un nombre crea una nueva rama Con –dr elimino una local/remota (la dejas de seguir, realmente no se elimina de remoto) -r me salen las remotas/locales
chekout	git checkout <nombre_rama> git checkout –b <nombre_rama>	“Cambia de contexto” Realiza una copia de la rama al directorio de trabajo, siendo sustituido el espacio de trabajo actual por el último commit referenciado por la rama a la que hacemos checkout. Visto de otra forma, mueve el HEAD. Si añade la opción b, además, crea la rama, pudiendo omitir así el comando branch Si lo haces sin la opción b, sobre una rama loca/remota, sin correspondencia local, pero ya captada por fetch te crea una nueva rama y te pone en ella

merge	git merge <nombre_rama>	Traslada los cambios (sumando o restando) de <nombre_rama>, a la rama actual (apuntada por HEAD). El problema está cuando se altera parte del mismo fichero en las distintas ramas
Log	git log git log --oneline --graph	Te saca el histórico de commits de la rama actual (HEAD) -- online te da resumen abreviado --graph te saca el grafo de versiones
Tag	git tag –a <nombre_version_sin_espacios> <commit> -m “mensaje descriptivo de la versión”	Le da un nombre lógico a una versión estable, por si queremos recordarla mejor que por un commit. “Etiquetamos” ese commit. Si se omite commit, se etiqueta el commit apuntado por la rama actual HEAD. Con sólo git tag, veo las versiones disponibles en el repositorio
Rebase	Git checkout mirama Git rebase master Git checkout master Git merge mirama	Rebase hace muchas cosas: 1º busca la versión común entre master y la rama actual (último checkout) 2º aplica todos los cambios desde ése punto en común a cada versión de la rama actual, desde la más antigua a las más moderna, aplicándole también los cambios en la rama master, desde la más antigua a la más moderna. 3º Finalmente, pone después de la versión master, las versiones de la rama actual (a

		las que ha ido añadiendo todas las modificaciones de cada versión de master) En ese momento, tengo mi trabajo, con la rama master al día. Pero eso no es la nueva rama máster. Para que esa última versión, figure como mi rama master, haré el merge mirama desde checkout master. ES IMPORTANTÍSIMO hacer rebase master estando en mirama y no al revés; ya que así, se parte del master (versión compartida por todos los programadores). No es lo mismo, coger el master y sumar mis cambios (cuya evolución, todos los programadores entenderían), Que sumar mis cambios a una versión prueba y después, añadírselos al master y hacer commit, lo que supondría una versión ininteligible para el resto de desarrolladores.
TRABAJO REMOTO –sharing and updating projects		
COMANDO	**EJEMPLO**	**COMENTARIOS**
remote	Git remote Git remote –v Git remote add <nombre_logico> <url> Git remote rm <nombre_logico>	A pelo, te muestra la lista de repositorio remotos conocidos -v te da detalle Add añade repositorio remotos con los que estar conectado para compartir o intereactuar RM, elimina remotos

fetch	Git fetch <nombre_logico>	Trae las ramas presentes en el repositorio remoto, con sus últimos cambios dejándolos en una carpeta especial “remote/bracnhes” en local, de la que ya podemos hacer merge. No permite commits ni trabajo normal con esta “remote/branch”-local. Si quieres trabajar con esa versión, debes hacer una nueva rama o mezclarla en tu rama de trabajo (HEAD) primero. **Si hay ramas nuevas (pull request en github), fetch te traerá las ramas nuevas. Para por ejemplo origin/rama-pruebas, una nueva rama traída con fetch, bastaría con hacer git checkout rama-pruebas, para que automáticamente se cree una rama y se esté en ella (como check out –b)
pull	Git pull <nombre_remoto>	Hace un fetch, y te intenta mezclar las ramas mapeadas automáticamente. Es como si tú hicieras un fetch y luego hicieras manualmente git merge origin/master
push	Git push <nombre_remoto> git push <remote-name> <branch-name-local>:<branch-name-remota> git push <remote_repo_name> --delete <branch_remote_name>	Me lleva el último commit, a convertirse en el HEAD del repositorio remoto. Si hay conflicto, debo bajarme primero el remoto con fetch y merge o hacer pull, editar los conflictos, add y commit antes de intentar nuevamente el push

		origin minueva-local:minueva-remota establece una nueva rama remota, creándola, a partir de la creada en local --delete elimina la rama tanto local, como local/remota (de coincidir el nombre), de verdad
BUSCANDO Y COMPARANDO VERSIONES – inspection and comparison		
COMANDO	**EJEMPLO**	**COMENTARIOS**

Log	`git log --author=Linus --oneline -5`
	
	git log --since --before
	`git log --oneline --before={3.weeks.ago} --after={2010-04-18} --no-merges`
	`git log --grep=P4EDITOR --no-merges`
	
	git log –stat -n
		Obtiene información de los cambios realizados por un autor o en una determinada fecha o hasta un determinado número de commits (-n)

Diff	`Git diff –stat rama1 rama2`
	`Git diff –stat rama1 commit`
	`Git diff –stat rama1 Versión`
		Para saber las difetencias (de forma abreviada con - - stat) entre dos ramas, una rama y un commit, y una rama y una versión. Al fin y al cabo, entre dos commis

`merge-base`	`git merge-base master erlang`	Calcula el commit que tienen en común dos ramas

CADA NODO ES UNA VERSIÓN DEL PRYECTO

Operativa para trabajar a diario

Hago fetch y/o pull del repositorio común

Creo mi rama de trabajo diaria, haciendo los commits cuando se alcance algún hito

Al final del día, vuelvo a hacer pull del repositorio remoto

Hago rebase de la rama master commit rama diaria

Mezclo mi rama con la master

Y push