**APUNTES GIT**

**Conceptos básicos**

**git --version:** Ver la versión de git instalada

**git config --global user.name *Usuario:***Define el nombre de usuario en la configuración global

**git config –global user.email *usuario@email.com***: Define el mail del usuario en al config global

**git help *comand:*** Abre un html con la ayuda asociada al comando solicitado

**ssh-keygen –t rsa –C*”Comentario para este ssh...“:*** Generar un ssh (si es que no tenemos uno ya)

En Windows, por defecto, quedará almacenado en C:/Usuarios/.ssh/

**git init:** Crea un repositorio en el directorio en que nos encontremos

**git init *nombre\_repositorio*:** Crea un repositorio (y el correspondiente directorio) que inicialmente estará vacío.

**git status:** Indica el estado del repo

Por defecto aparecerán en rojo los archivos “Untracked”, es decir los que todavía no han sido movidos a la “staging area”. En verde los que sí lo están (“Tracked”)

**git add *nombre\_archivo***: Añade el archivo a la “staging area”. (Cambia el color de rojo a verde)

**git add .** : Añade todos los archivos a la “staging area”. (Los archivos renombrados o eliminados no se tendrán en cuenta. Tan solo añade archivos nuevos o con modificaciones en su contenido)

**git add –p:** Añade los archivos a la “staging area”, pero va preguntando uno a uno para confirmar cada cambio

**git commit –m *‘comentarios’* :** Hace un commit con los comentarios que se indican entre comillas. Si no se añade comentario, por defecto se abrirar una ventana con un editor de texto (vim típicamente) para que se escriba un comentario asociado al commit. Los archivos pasan de la “staging area” al repositorio local. Si no se le pasa –m*’comentarios’* nos forzará a ponerlos abriéndonos el editor que tenga configurado por defecto (habitualmente vi o vim)

Para hacer un commit sin pasar previamente por el add, podemos hacerlo todo de una con git commit -a (-m *‘comentarios’*). Cuidado porque solo sirve para archivos modificados. Si añadimos algún archivo nuevo, tenemos que hacer el add obligatoriamente.

Para cambiar el editor cuando hacemos por ejemplo un commit: export EDITOR=notepad. Por defecto está puesto el vi o el vim. <http://iie.fing.edu.uy/~vagonbar/unixbas/editorvi.htm>

Con unos pocos comandos básicos se puede ya trabajar en vi editando y salvando un texto:

**vi *arch1*** arranca en modo comando editando el archivo arch1

**i** inserta texto a la izquierda del cursor

**a** agrega texto a la derecha del cursor

**ESC** vuelve a modo comando

**x** borra el caracter bajo el cursor

**dd** borra una línea

**h o flecha izquierda** mueve el cursor un caracter a la izquierda

**j o flecha abajo**  mueve el cursor una línea hacia abajo

**k o flecha arriba** mueve el cursor una línea hacia arriba

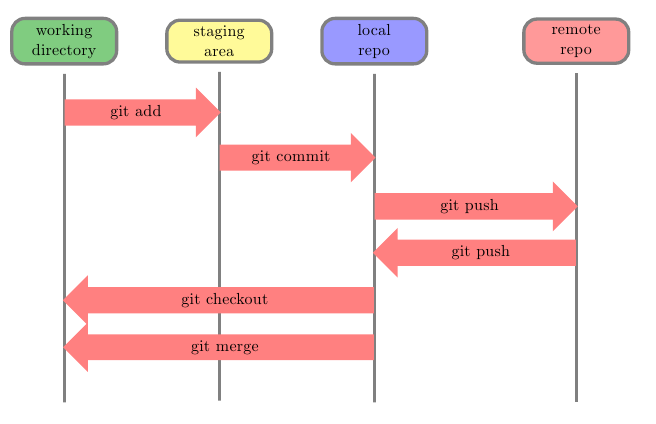
**l o flecha derecha** mueve el cursor un caracter a la derecha

**:w** salva el archivo (graba en disco)

**:q** sale del editor (debe salvarse primero)

**git add . –A:** Añade los cambios a la “staging area” tenienedo en cuenta los archivos renombrados y borrados (-A = --all)

Nota: Si queremos renombrar un archivo y ademas cambiar su contenido, es aconsejable hacerlo en dos pasos (dos commits diferentes), para ayudar a la herramienta a identificar los cambios y por mayor claridad a la hora de ver el historial de cambios.



**git rm *nombre\_archivo*:** Elimina del repositorio el archivo. (También lo elimina localmente)

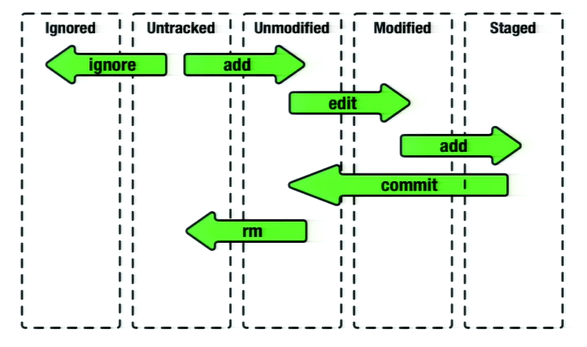
Para ignorar varios archivos simultáneamente, crear el fichero .gitignore. Dentro se puede utilizar por ejemplo \*.log para ignorar todos los .log de la carpeta actual. Para ignorar en subcarpetas, utilizar \*\*/

Para ignorar un directorio completo, simplemente escribir su nombre, por ejemplo target.

El archivo .gitignore aplica al directorio en el que se encuentra, y todos sus subidrectorios.

Hacer ignore sobre un tipo de archivo que ya está en el repositorio, no lo elimina del repositorio ni hace que se deje de “trackear”. La herramienta seguirá considerándolo trackeado (puesto que fue añadido antes de que cambiara el .gitignore). Sin embargo, archivos con esa misma extensión creados posteriromente sí serán ignorados.

**git rm --cached *<file or regexp>***git no realiza un seguimiento del archivo, pero los deja en el directorio de trabajo. Útil cuando se olvida añadir archivos al .gitignore y ya hemos agregado dichos archivos al repositorio.



**git push:** sube el contenido al repositorio remoto

**git clone *address*:** Clona el contenido de un repositorio remoto en un repositorio en local. (Se le puede añadir un parámetro adicional al final para renombrarlo localmente)

Git soporta los siguientes protocolos de clonado:

* ssh
* http
* git
* file

git clone ssh://user@server:project.git (o bien… git clone user@server:project.git)

git clone <http://server/project.git> (No utiliza protocolo de autenticación)

git clone git://server/project.git (Port : 9418)

git clone <file://myrepos/project> (o bien... git clone /myrepos/project)

**git remote add *origin git@github.com:Username/Project.git* :** Añade donde está el repositorio remoto. Normalmente se usa la primera vez si no hemos utilizado la opción de clonar el repositorio.

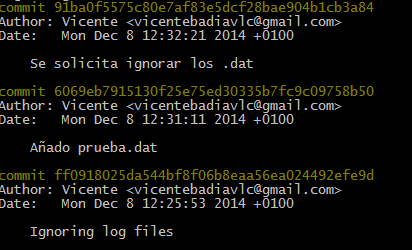
**git remote set-url *origin git@github.com:Username/Project.git*** : Cambia la url del repositorio remoto a una de tipo SSH. La dirección se puede meter en el portapapeles desde github

**git remote –v:** Muestra el origen del repositorio remoto (http://...)

Si el ssh nos pide una clave y la queremos eliminar: **ssh-keygen -f */c/user/direccion/…/id\_rsa* –p**. Nos pedira la clave antigua, al darle la nueva presionamos directamente intro y ya no nos volverá a hacer falta introducirla para trabajar con el repositorio remoto.

**Almacenamiento y hashes**

A cada commit se le asocia un hash único que lo identifica.



Habitualmente no es necesario utilizar todos lo caracteres para identificarlo en un grupo de trabajo. Con emplear los 4 o 5 primeros suele ser suficiente. Para referirnos al último commit se emplea el término HEAD.

**git log:** Lista todos los commits, informando de su correspondiente hash, autor y fecha

**git log --pretty=oneline:** Lista todos los commits de forma compacta

**git log --graph:** Muestra el historial de forma gráfica. Se le puede añadir al final (--pretty=oneline) para que tenga el mismo efecto que en el ejemplo anterior

**git log --stat:** Muestra el historial indicando el número de cambios en cada commit

**git log *hash*:** Lista los commits anteriores al hash que se le pasa. (No es necesario escribir el hash entero, los primeros caracteres es suficiente)

**git log *hash\_inicio\_rango..hash\_fin\_rango:*** Lista los commits entre dos rangos. Los rangos se pueden pasar utilizando el principio del hash (a5bc...) o comodines como HEAD o el símbolo ^. Por ejemplo git log HEAD^^^..HEAD, nos mostraría los tres últimos commits. (El símbolo ^ repetido n veces puede ser reemplazado por ~n, por ejemplo HEAD^^^^ equivale a HEAD~4)

**git show:** muestra información sobre el último commit (hash, autor, fecha, comentario y cambios realizados)

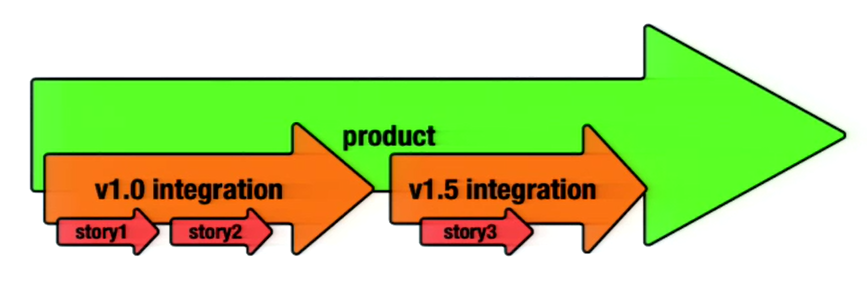
**Trabajando con Branches (ramas)**

Categorizar las branches por lifetime (de menor a mayor):

-Short lived: (la más corta de las ramas)

- Feature: (rama de tamaño medio)

- Release: (rama de mayor duración)



**git remote add pb *dirección*:** Añade el repositorio remoto

**git pull:** Actualiza el contenido local con lo más reciente del repositorio remoto (utiliza la misma dirección que utilizó el clone). Importante asegurarse de hacerlo antes de empezar a crear ramas.

**git branch *nombre*:** Crea una nueva rama. (Aunque está creada no estamos dentro de ella, así que habrá que hacer un checkout para entrar dentro de la rama)

**git checkout *nombre*:** Empezamos a trabajar en la nueva rama

Se puede unificar los dos comandos anteriores usando **git checkout –b *nombre****,* de esta forma, de un solo paso, creamos la rama y nos metemos en ella para empezar a trabajar.

**git checkout -** : Vuelve al anterior

**git branch –a:** Muestra una lista con todas las ramas locales (verde) y remotas(rojo)

**git push origin *nombre*:** Hace un push al repositorio remoto “origin” de la rama “nombre”

**git branch --set-upstream *branch origin/branch:*** Permite que la rama actual sea bidireccional, es decir, no solo nosotros vamos a subir cosas a esa rama, sino que vamos a permitir que otros usuarios también suban cosas para que nosotros las tomemos. Si no, por defecto, una rama creada por mi, solo podría ser editada por mi, el resto de usuarios solo harían pulls pero no push.

**git push origin *branch:nueva\_branch:*** Crea una nueva rama en el repositorio remoto igual a la rama branch, pero con otro nombre. (No es necesario estar en la rama para hacerlo)

**git branch –d *nombre:*** Elimina la rama

**Remotes**

El nombre por defecto cuando se clona un remote es **origin**. Sin embargo podemos renombrarlo o incluso añadir otros adicionales. Por ejemplo podemos tener un remote para ssh (con una pareja fetch y push) y otro remote para https (también con su respectiva pareja de fetch y push).

**git remote add *nombre\_del\_remote git@github.com:Username/Project.git* :** Añade un nuevo remote con el nombre especificado.

**git fetch *nombre\_del\_remote*:** Obtiene todo el contenido de un repositorio remoto en concreto, pero sin realizar el merge. En ese punto podríamos ver las diferencias entre una de mis ramas y las del remote. Si me quiero quedar con el contenido debería hacer un pull o un merge. (Nota: pull hace un fecth combinado con un merge).

**git diff *branch1* *branch2*:** Muestra las diferencias entre 2 ramas. Puede resultar muy útil después de hacer un fetch, de forma que en *branch1* comparamos lo que tenemos en local con lo que acabamos de tomar del servidor *(branch2).* Por tanto este branch2 sería de la forma remote/branch.

Podemos añadir al final del comando anterior **--word-diff**. De esta forma en lugar de mostrar la linea completa antes y después del cambio, muestra los cambios palabra a palabra. Por ejemplo, en modo normal mostraría:

**- Some random text: hi**

**+ Some random text: hello**

Mientras que con el --word-diff mostraría **Some random text: [-hello-]{+hi+}**

**Tagging**

Podemos distinguir dos tipos de tags:

Los tags “light” son empleados para llevar un cierto control, pero nos permiten cierta flexibilidad, en el sentido de que se pueden modificar sus características a posteriori si nos conviene. Estos tags no requieren un mensaje.

Los “hard” tags son aquellos que por su naturaleza no pueden editare. Por ejemplo cuando se trabaja en algún proyecto para una aplicación de forma profesional o existe alguna obligación contractual, crearemos uno de estos tags que son invariables. Al crearlos es además obligatorio incluir un mensaje al estilo de los commits.

Nota: Habitualmente se utilizan letras mayusculas y subguiones para nombrar los tags, aunque no es obligatorio.

**git tag *TAGNAME:***Crea un tag en modo light (puede reeidtarse autor, fecha,…).

**git tag –a *TAGNAME:***Crea un tag en modo heavy (no puede editarse autor, fecha…)

Por defecto los tags no se suben al remoto

**git push -- tags:** Hace un push indicando que queremos subir los tags.

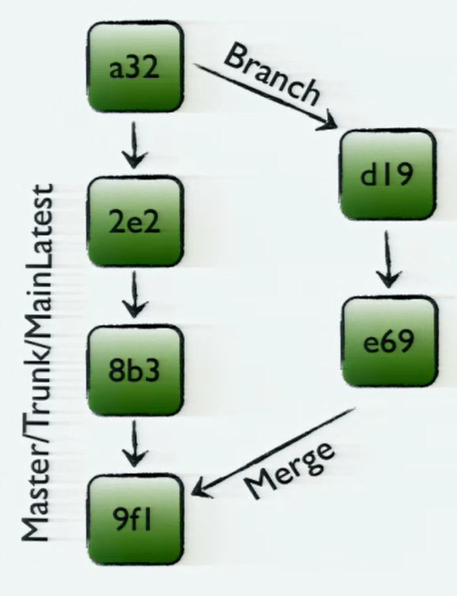
**git tag –d *TAGNAME:*** Elimina el tag. (Aunque si está en el remote, al volver a hacer pull, reaparecerá)

**git tag –f *TAGNAME:*** Fuerza a crear un tag, aunque ya haya uno con ese mismo nombre. Por tanto lo sobrescribirá.

**Merging**

Consiste en combinar el contenido de una rama con el de otra (típicamente con la master). En algunas ocasiones es automático, pero en otras es complicado, si se han hecho cambios en los mismos archivos, en las mismas lineas.

Para hacer el merge, primero nos ponemos en la rama de destino (típicamente **git checkout master**). Y haremos un **git merge *rama.***

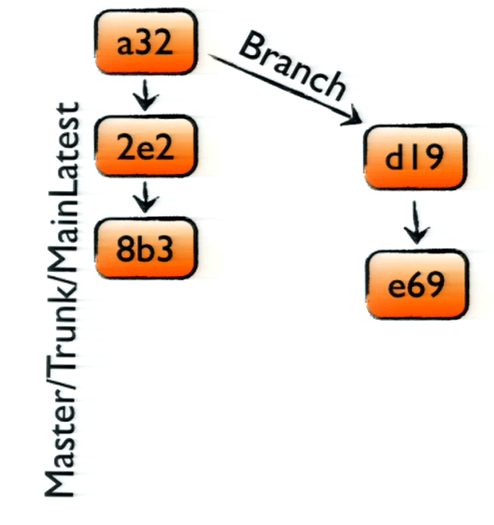
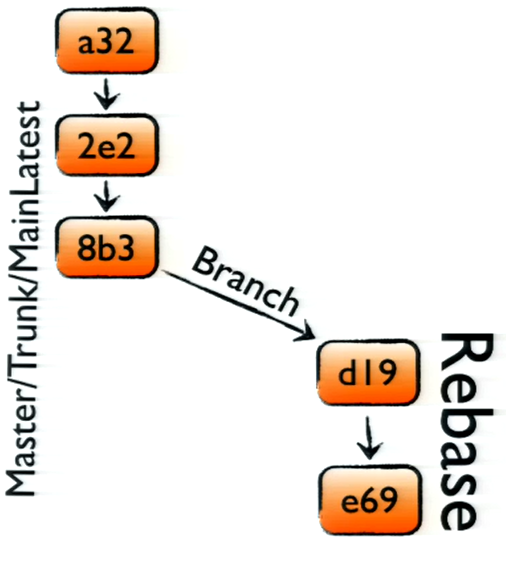
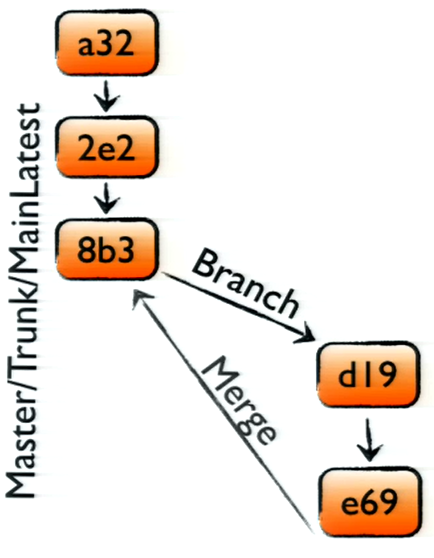
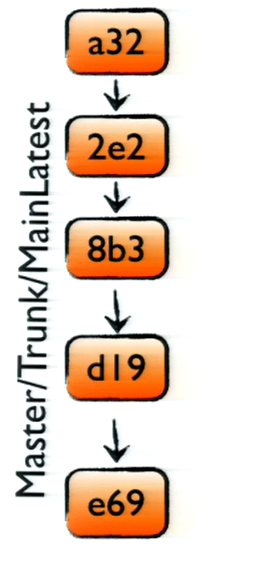


**git checkout - - theirs:** Para quedarme en un merge con la versión de un conflicto de la otra rama y no la mía. (Se utilizaría ours para quedarme con la mía. Para que solo afecte a un archivo, habría que indicar a continuación cual. Si no, se aplica a todos)

**Rebasing**

No se trata de un simple merge. Rebasing cambia la historia. Simula miembros de un equipo tomando turnos. Esto permite que el historial sea más sencillo de seguir en el futuro. Como puede verse en el ejemplo, aunque el trabajo se ha hecho en paralelo (en master y en una rama), al hacer un rebase reescribimos el historial como si el trabajo de master se hubiera hecho primero, y a continuación el de la rama (que finalmente desaparece por simplicidad para la lectura del historial en el futuro)

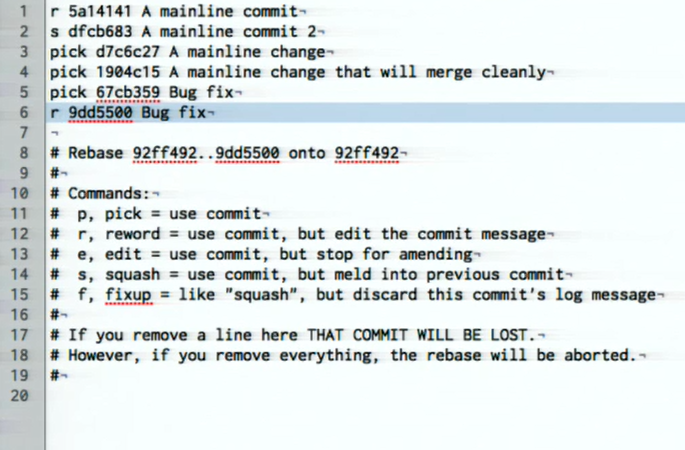
Es muy importante que el Rebase se haga en el repositorio local, para que el resultado final que llegue al repositorio remoto tenga el aspecto de haber trabajado “en turnos” y no un conjunto de ramas.

Para hacer un rebase, primero hacemos un checkout sobre la rama que queremos modificar

Rebase también permite reorganizar un grupo de commits (locales) que hemos realizado de forma consecutiva y que quiza sería más conveniente agrupar o corregir sus comentarios asociados.

**git rebase HEAD^^^^ -i:** Se abrirá un editor en el que se podrán reorganizar los commits elegidos (en este caso los cuatro últimos). Con r podemos reescribir el comentario del commit, con s combinar un commit con el anterior, borrando la linea podemos eliminar el commit, etc… Las instrucciones vienen en el propio editor.



Tras realizar un rebase, si se necesita que resolvamos problemas manualmente, tendremos que ir escribiendo **git rebase --continue**, para ir avanzando en los sucesivos pasos de la ejecucción del rebase. (Para abortar el rebase haríamos **git rebase –abort**)

**Deshacer**

**git commit --amend:** Permite corregir el último commit, reescribiendo el comentario si fuera necesario, y reactualizando al estado actual de la staging área.

**git revert *9ab844*:** Hace un revert del commit *9ab844*. Una vez acabada la edición del revert, se debe hacer un commit. Revert no borrara de la historia el commit *9ab844*, simplemente añadirá otro que avisa del cambio en el anterior. (También se puede utilizar para revertir un rango de varios commits por ejemplo entre 9ab844..65bc81. O incluso con comodines como HEAD^^^^..HEAD^)

**git reset --hard:** Vuelve al estado anterior (al estado del último commit). Así que si he estado algunos archivos y me arrepiento, puedo hacer un reset y vuelvo a un estado limpio exactamente igual al último commit.

Se puede volver a un estado aún más antiguo y por tanto eliminar commits intermedios, por ejemplo con el modificador ^. Así el comando, git reset --hard HEAD^^^ nos devuelve al estado de hace tres commits. Si tras hacer esto intentaramos un push, git nos va advertir que vamos a modificar el historial del remote. Si aún así estamos seguros tendríamos que utilizar –f con el push para forzarle.

Nota: Para recuperar un commit “perdido“

**git reflog:** Muestra el historial de los ultimos HEADs. A continuación con git reset --hard <hash\_id> se puede recuperar el commit que previamente se había borrado con otro reset

**git stash:** Mueve todo el progreso desde el último commit a un “directorio” aparte (dirty) y devuelve el estado al del último commit. Este directorio “sucio” puede posteriormente recuperarse. Un ejemplo de uso podría ser que estamos trabajando y de golpe el jefe nos pide hacer algo urgentemente ahora mismo sobre el último estado. En ese caso hacemos un stash, lo que estabamos haciendo se queda aparte (dirty), volvemos al estado del último commit. Hacemos la tarea urgente, y luego podemos recuperar lo que estabamos haciendo.

**git pull origin +master:master:** Fuerzo a que el contenido de mi master sea exactamente igual al del repositorio remote. (Antes de hacer cualquier push es necesario haber bajado previamente el último estado del repositorio remoto)

**git clean -f:** Elimina los archivos que no estén “trackeados”. Hay que añadir –f porque si no, por defecto no los borraría. Los archivos que no estén “trackeados” (por ejemplo usando .gitignore) no serán tocados por este comando. Para que elimine subdirectorio hay que añadir -d

**git config --global alias. *nombre\_alias* “comando git”:** Asigna a un determinado alias un comando de git, por ejemplo: git --config alias. arbol “log --graph pretty=oneline”. Ahora al teclear “árbol” ejecutara todo el comando log en modo grafico y una línea.

**git notes add:** Añade una nota al ultimo commit, para incluir aclaraciones.

Por defecto git no sigue la traza de los directorios vacíos. Si quisiéramos que los tuviera en cuenta podemos crear dentro del directorio un .gitignore y dejarlo vacio. A partir de ese momento, git ya considerará que ese directorio debe ser tenido en cuenta.

**CONSOLA**

**ls –al:** Lista el directorio actual con las propiedades (l) incluyendo archivos ocultos (a)

**touch *nombre*:** Crea un archivo

**cat *nombre*:** Muestra el contenido del archivo

**echo *texto que se escribe* >> *archivo*** : Concatena el texto escrito dentro del archivo

**mv *nombre\_antiguo nombre\_nuevo*:** Renombra el archivo

**notepad *nombre*:** Abre el archivo en el bloc de notas

**pwd:** Devuelve la ruta completa de la carpeta en la que estamos

**clip < *ruta\_del\_archivo*:** Copia el contenido del archivo al portapapeles

**Git Avanzado**

**Configuración**

Se puede aplicar la configuración a tres niveles: system, global o local. (Por ejemplo git config --system, git config --global o git config --local). Rara vez se usa system. Lo normal es usar global de manera genérica y local para un uso particular en un repositorio. Si despues de config no se pone nada, se entiende que es --local.

Si ejecutamos git config --system realizará los cambios en /etc/gitconfig. (Está asociado al sistema, se aplicará a todos los usuarios que utilicen el ordeandor)

Si hacemos git config --global salvará los cambios en ~/.gitconfig. (Está asociado a la cuenta de usuario) En el caso de windows se suele almacenar en C:/Usuarios/Vicente/.gitconfig

Si hacemos git config --local lo almacenará en .git/config. (En el propio repositorio local)

**git config --list:** Devuelve la configuración actual por consola.

Para obtener un parámetro concreto podemos usar git config section.key (por ejemplo git config color.ui). O git config section.subsection.key.

Para configurar un parámetro se usa git config section.key value. Se pueden crear secciones y keys al gusto del usuario y luego utilizarlas para algún tipo de scripting.

**Pull Options**

El comando pull hace un merge por defecto. Una opción interesante es utilizarlo haciendo un rebase.

**git pull --rebase:**

**git config branch.autosetuprebase allways:**

**git branch –r:** Muestra todas las ramas en el repositorio remoto

**Rerere**

Reused recorded resolution (of conflicted merges). Es una herramienta local que permite almacenar resoluciones de conflictos en merges que se sospecha que se van a repetir habitualmente.

**git config rerere.enabled true:** Habilita la resolución automática de conflictos que se repiten.

**git rerere status:** (Estando en la resolución de un merge) Muestra los archivos que están en conflicto (si tenemos habilitado rerere). La forma de resolver los conflictos de estos archivos quedará almacenada al acabar.

**git rerere diff (|less):** Estando en un conflicto, muestra los cambios. Podemos incluir less para que no muestre tanta info.

Al acabar una resolución avisa de que ha grabado la resolución para los archivos que hemos resuleto.

**git rerere gc:** Borra autoresoluciones memorizadas (en caso de que tuvieramos problemas de performances, lo cual es poco probable)

Se crea un archivo rr-cache donde se almacenan estas resoluciones automáticas. Esta información está almacenada en local.

La próxima vez que ocurra en conflicto similar, aplicará la resolución que ha memorizado. Aunque no los pondrá en la staging area. Tendremos que hacer un add.

**Push Options**

¿Que es lo que se sube al repositorio remoto cuando no se le pasan argumento al push?

**git config --global push.default matching:** (Este es el modo de operar de git por defecto hasta la versión 2.0. Ahora es “simple”). Sube todas las ramas que coincidan en nombre con el del repositorio remoto.

**git config --global push.default nothing:** Solo sube si le especifico el nombre de la rama. (Las demás ramas no se subiran)

**git config --global push.default current:** Solo sube la rama en la que estoy

**git config --global push.default tracking:** Sube la rama actual a lo que está trackeando (para el caso en que tengamos varios repositorio remotos)

**Bundle**

**git bundle create *filename.bundle rama(s)*:** Crea un bundle. Normalmente se crea fuera del repositorio (por ejemplo indicando ../filename.bundle para crearlo en un nivel superior). Se pueden pasar una o varias ramas para hacer el bundle.

Una vez creado el bundle, se le puede pasar a otro usuario a través de un usb, o por la red, etc…

**git bundle list-heads *filename.bundle:*** Muestra los heads de las ramas que están metidas en el bundle.

**git ls-remote *filename.bundle:*** Tiene el mismo efecto que ejecutar este comando sobre origin, lo que demuestra que a efectos de información el bundle puede contener exactamente lo mismo que un repositorio remoto. (Evidentemente, el bundle no dispone de las ventajas de tener la información en remoto accesible).

**git clone *filename.bundle name:*** También se puede clonar desde un bundle, aunque con algunas limitaciones. Se da un warning que avisa de que HEAD se refiere a algo no existente si en el bundle no estaba incluido. Para crearlo es necesario hacer a continuación un checkout (o bienm haberlo incluido en el bundle)

**git fecth *filename.bundle rama*:** Se creará un FETCH\_HEAD asociado al bundle.

**git merge FETCH\_HEAD:** Hará un merge de lo que acabo de obtener con el fetch sobre la rama en la que estoy.

**git remote add *filename.bundle*:** Añade un bundle como si fuera un repositorio remoto. (Util si se van a estar pasando bundles con el mismo nombre. De esta forma me ahorro tener que escribir el nombre cada vez.) Este tipo de remote solo permite hacer pulls (push no es posible, puesto que no tiene sentido)

**Patches**

Útil para transporte offline. Permite asociar firmas. Funciona fuera de repositorios Git.

Hay tres posibles operaciones. Crear el patch con git format-patch, aplicarlo con git apply y aplicarlo con git am:

**git format-patch HEAD~2 patches:** Crea una carpeta llamada patches con los archivos que contienelos patches asociados a los últimos dos commits.

**git apply patches/\*:** aplica todos los patches contenidos en la carpeta patches. Genera por tanto un commit.

**git apply ../pacthes/\* --index:** Aplica todos los cambios de los patches del nivel superior y lo deja en la staging area, como si los hubiera modificado yo mismo y hubiera hecho un git add .

**git am ../patches/\*:** Aplica patches desde el correo. Genera por tanto un commit. Se incluye información asociada el correo de donde vino el patch (hora, asunto, versión de git…). Las siglas “am” vienen de apply mailbox

**git am ../patches/\* --signoff:** Además de lo anterior añade una firma al commit.

**git format-patch --stdout > unified.patch:**

**git diff > ../diff.patch:**  Añade las diferencias que he aplicado desde el último commit a un patch. Posteriormente se puede hacer un git apply a este archivo y creará los cambios definidos en el patch.

**Ramas huérfanas**

No tienen una referencia a un “parent”. Útil para documentación, recursos, archivos que cambian poco, etc… (No es muy habitual su uso)

**git checkout --orphan *docs*:** Crea la rama docs, que no pertenece a ninguna rama padre. Se empleará para archivos que no van a cambiar mucho, como por ejemplo la documentación.

Si se utiliza Github, las ramas (huérfanas) que se llamen gh-pages crean páginas web para la documentación. La dirección tendrá el formato *http://username.github.com/project\_name.* Para más información ver <https://help.github.com/articles/creating-project-pages-manually/>