**CONTROL DE VERSIONES V: GIT AMPLIADO, HERRAMIENTAS GRÁFICAS Y COMANDOS BÁSICOS**

[*Git*](http://git-scm.com/) *es un Sistema de Control de Versiones (*VCS) *distribuido* [*libre y de Código Abierto*](http://www.opensource.org/docs/osd)escrito en una combinación de Perl, C y scripts de shell.

Git se ha impuesto a todos los demás sistemas de control de versiones (especialmente si se escoge un sistema distribuido) junto a su plataforma GitHub.

La principal diferencia entre Git y cualquier otro VCS (Subversión, entre otros…), es que no almacenan la información como una lista de cambios en los archivos, (que con el tiempo, en desarrollos grandes y con muchos desarrolladores, se puede convertir en algo lento), sino que lo almacena como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que se confirma un cambio, o guarda el estado del proyecto en Git, básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado

- No almacena las diferencias entre archivos, sino instantáneas del proyecto completo.

- La mayoría de las operaciones son locales, por lo tanto no necesitas estar conectado a ninguna red para trabajar con Git la mayor parte del tiempo (aunque en algún momento se tendrán que incorporar los cambios al proyecto).

- Tiene integridad,una revisión no tiene un número, sino un largo número hexadecimal *hash*. Esto es para que no se altere la integridad del *commit*. El *hash* se calcula a base de los datos que componen un *commit*. Cambiar un byte cambia el *hash*.

Vocabulario

|  |  |
| --- | --- |
| Repositorio ó Repository | Un repositorio contiene la historia, las diferentes versiones en el tiempo y todas las diferentes ramas (branches) y etiquetas (Tags). Cada copia del repositorio en Git es un repositorio completo. |
| Ramas ó Branches | Una rama o branch es una línea de código por separado con su propia historia. Puede crear una nueva rama de una existente y cambiar el código de forma independiente de otras ramas. Una de las ramas es el valor por defecto (normalmente llamado master ó maestro). |
| Etiquetas ó Tags | Una etiqueta ó Tag apunta específicamente a cierto espacio en el tiempo en una rama ó branch específico. Con un tag, puede tener un punto de referencia al que siempre puede revertir, por ejemplo, el código del 15.05.2012, en el branch “test” o el código de la versión 1.5 en el brach “master”. |
| Commit | Confirmar cambios. Crea una nueva revisión, que puede ser recuperada más tarde. Cada commit contiene el autor y el committer o quien realiza los cambios, por lo que es posible identificar la fuente del cambio. |
| Revisión | Representa una versión del código fuente. Git identifica las revisiones con los identificadores de SHA1. Los SHA1 son los identificadores de 160 bits de longitud y están representados en hexadecimal. La última versión se puede abordar a través de “HEAD”, la versión antes de que a través de “HEAD ~ 1” y así sucesivamente. |

Instalación

Linux: A partir de los repositorios oficiales de cada distribución ó:

Debian/Ubuntu: **$ apt-get install git-core**

##### Fedora: $ yum install git

Gentoo: **$ emerge –ask –verbose dev-vcs/git**

…

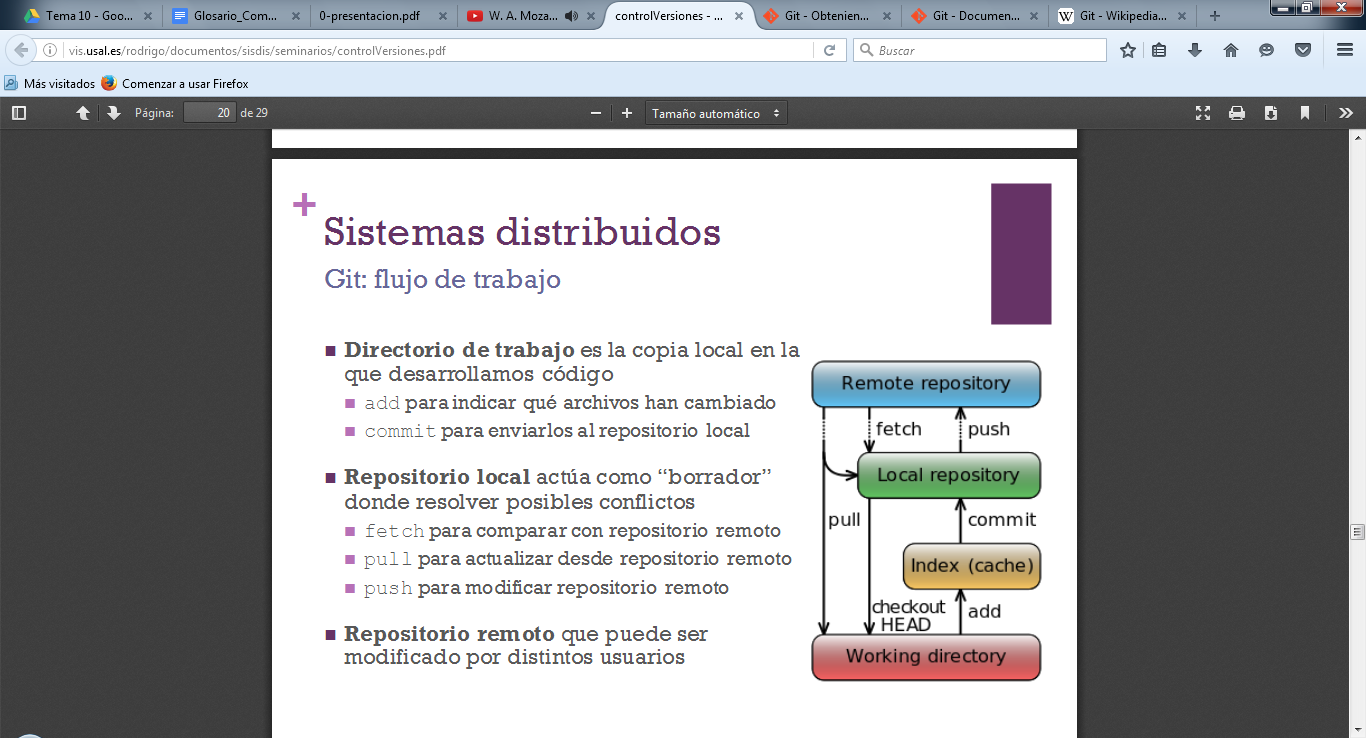
Windows Instalar desde el siguiente enlace [GitHub en Windows](https://git-for-windows.github.io/)

Veamos otras características que, como las anteriores, están sacadas del [manual que se incluye en la página oficial de Git](https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Fundamentos-de-Git)

Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged).

* Confirmado significa que los datos están almacenados de manera segura en tu base de datos local.
* Modificado significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos.
* Preparado significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git: El directorio de Git (Git directory), el directorio de trabajo (working directory), y el área de preparación (staging area).

* El directorio de Git, la parte más importante, donde se almacenan los metadatos y la base de datos de objetos. Es lo que se copia cuando se clona un repositorio..
* El directorio de trabajo es una copia de una versión del proyecto con la que el usuario trabaja en local.
* El área de preparación, es un archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en la próxima confirmación. A veces se le denomina índice (“index”).

El flujo de trabajo básico en Git sería:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiéndolos a tu área de preparación.
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de Git.

**INTERFACES GRÁFICAS PARA TRABAJAR CON GIT**

La interfaz nativa de [Git](https://git-scm.com/book/es/v2/Git-en-otros-entornos-Interfaces-gráficas) es la línea de comandos, hay que tener en cuenta que es el único lugar en donde puedes ejecutar **todos** los comandos de Git, porque las herramientas gráficas están enfocadas a realizar una u otra parte del trabajo. En todo caso, cuando se instala Git, también se pueden instalar herramientas gráficas:

* gitk es un visor gráfico del histórico.
* git-gui es principalmente una herramienta para elaborar commits.

Existen muchas más y están incluidas en la misma [página oficial de Git](https://git-scm.com/downloads/guis), muy bien clasificadas por sistema operativo y precio. En la [wiki de Git,](https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools) por su parte hay una lista exhaustiva de herramientas, extensiones, etc.. de todo tipo para Git.

**Gratuitas:**

[**SourceTree**](https://www.sourcetreeapp.com/)

**Windows y Mac**

De Atlassian (como Bitbucket): En principio solo estaba disponible para Mac pero actualmente también se puede utilizar desde Windows. Es un cliente para Git y también para Mercurial. Sirve para manejar las operaciones más habituales y también es una ayuda para migraciones desde Subversion (con [SubGit](http://www.genbetadev.com/sistemas-de-control-de-versiones/subgit-migrar-de-subversion-a-git-sin-quebraderos-de-cabeza)). En principio es gratuito.

[**GitHub Desktop**](https://desktop.github.com/)

**Windows y Mac**

GitHub ha creado dos clientes Git: uno para Windows y otro para Mac. Estos clientes son un claro ejemplo de herramientas orientadas a flujo de trabajo, en vez de proporcionar toda la funcionalidad de Git, se centran en las funcionalidades más utilizadas.

[**TortoiseGit**](https://tortoisegit.org/) **Windows** Gratuita y Opensource, específica para Windows.

Está basada en TortoiseSVN, la gui más popular de Subversion, y como ésta, no se integra con ningún IDE en concreto, sino con el sistema de archivos de Windows.

[**GitKraken**](https://www.gitkraken.com/) **Windows, Mac y Linux (**Una de las herramientas más reciente, de 2016. Es gratuita pero no open source).

[**Git-cola**](https://git-cola.github.io/) **Windows, Mac y Linux** Desarrollado en python con el toolkit QT.

[**giggle**](https://wiki.gnome.org/Apps/giggle/) **Linux** Frontend que usa el entorno de desarrollo GTK+.

[**GitEye**](http://www.giteyeapp.com/) **Windows, Mac y Linux**

[**RabbitVCS**](http://rabbitvcs.org/) **Linux**

[**Egit**](http://www.eclipse.org/egit/)Egit es un plugin para el entorno de desarrollo Eclipse, o como indica el faq de Eclipse "EGit es un proveedor de Eclipse Team con licencia [EPL](http://www.eclipse.org/org/documents/epl-v10.php) para [Git](http://git-scm.com/), construido sobre JGit.(...) JGit es una biblioteca JAVA pura, ligera, con licencia [EDL/BSD](http://www.eclipse.org/org/documents/edl-v10.php) que implementa las rutinas de acceso al sistema de control de versiones [Git](http://git-scm.com/), sus protocolos de red y algoritmos nucleares de control de versiones. " Se puede bajar directamente desde Eclipse, junto a ayuda y un manual en castellano para utilizarlo.

**Comerciales**

[**SmartGit**](https://www.syntevo.com/smartgit/) **Windows y Mac y Linux** De pago, pero gratuito para usos no comerciales.

[**Tower**](https://www.git-tower.com/) **Windows y Mac** De pago, pero con período de prueba gratuito.

Se pueden seguir dos caminos, iniciar la operativa en el entorno distribuido y llevarlo a local o bien comenzar la operativa en local y pasarlo después a entorno distribuido entre los que destacar GitHub , Bitbucket, GitLab ó Coding por citar los más conocidos, todos ellos ofrecen un entorno colaborativo de desarrollo. Si bien todos ellos ofrecen las mismas operaciones básicas:

* Solicitud de extracción
* Revisión de código
* Edición en línea
* Seguimiento de problemas
* Soporte de Markdown
* Autenticaciones de dos factores
* Gestión avanzada de permisos
* Páginas web estáticas alojadas
* API de funciones avanzadas
* Contenedor de repositorios / clon

Para trabajar con Git o Github hay dos formas de trabajar:

1) Trabajar en local, en un repositorio que se crea en mi máquina.

2) Clonar un repositorio de Github (u otro hosting de repositorios) para traerse a local el repositorio completo y empezar a trabajar con ese proyecto.

La opción 1) permite comenzar desde cero y es con la que se puede apreciar mejor cuáles son las operaciones básicas con Git. Así, cualquier operación que se realiza con Git tiene que comenzar mediante el trabajo en local, por lo que hay que comenzar por crear el repositorio en tu propia máquina. Incluso si el objetivo es simplemente subir el repositorio a Github para que otras personas lo puedan acceder a través del hosting remoto de repositorios, hay que comenzar trabajando en local.

Lo lógico es iniciar el desarrollo en GIT y posteriormente extenderlo a un entorno distribuido. ([Manual de GIT](https://git-scm.com/book/es/v1))

Comprobamos si está instalado con “git --version”

Así, una vez instalado lo primero es configurar, con “git config” que permite obtener y establecer variables de configuración,almacenables en tres sitios distintos:

/etc/gitconfig: Para todos los usuarios y todos sus repositorios.“git config --system”

~/.gitconfig : Específico a tu usuario. “git config –global”

.git/config : de cada repositorio en particular

Cada nivel sobrescribe los valores del nivel anterior, por lo que los valores propios del repositorio tienen máxima preferencia.

Crear una identidad, usuario y e-mail, las confirmaciones (commits) la usan.

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

Si se realiza con --global, sólo será necesario la primera vez y se usará para todos los proyectos, para cambiarlo en proyectos específicos se ejecutará sin la opción --global desde el directorio específico del proyecto.

Con “git config -l” nos mostrará que valores se van a utilizar.

El editor de texto por defecto cuando Git necesite que se introduzca un mensaje será el editor por defecto del sistema, (Vi o Vim usualmente). Para cambiarlo, por Emacs, por ej.sería con:

$ git config --global core.editor emacs

La herramienta de diferencias por defecto, usada para resolver conflictos de unión (merge)

$ git config --global merge.tool vimdiff

Git acepta kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, y opendiff.

\* Por defecto no viene instalado ninguno para usar vimdiff es necesario instalar antes vim.

Finalmente se comprueba la configuración,con git config --list para listar los valores configurados. Pueden verse claves repetidas, pues Git lee la misma clave de los distintos archivos presentes , utilizando el último valor para cada clave, que es única.

Se puede obtener ayuda de 3 formas:

$ git help <comando>

$ git <comando> --help

$ man git-<comando>

\* Para crear el repositorio, se crea un directorio (mkdir directorio) y desde él se ejecuta git init

Se crea así un subdirectorio .git con todos los archivos necesarios del repositorio —un esqueleto de un repositorio Git. Creamos un fichero README de descripción del proyecto.

Para empezar a seguir todos los ficheros que se incorporen a este directorio se utiliza git add

$ git add README // Se añade sólo el fichero descripción del proyecto.

$ git add file1 file2.txt file3.c // Se añaden los 3 ficheros especificados.

$ git add \*.\* // Incorpora todos los ficheros presentes.

$ git commit –m 'versión inicial del proyecto' // Se confirma el cambio.

A la acción de guardar los archivos en el repositorio se llama, en la jerga de Git, hacer un "commit". En este caso el commit se está haciendo en local, porque los archivos se están enviando al repositorio local que está en la máquina.

Para comenzar con una copia de un proyecto existente, se utiliza: $ git clone [url]

$ git clone git://github.com/schacon/nombreRepositorio.git

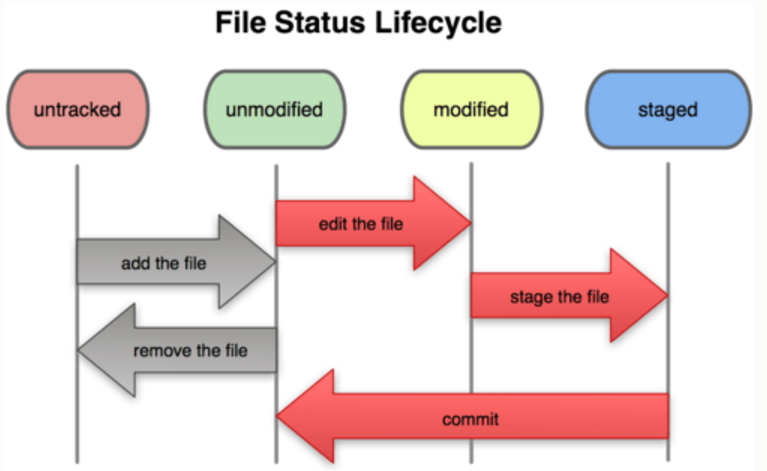
Y respetará el nombre original del repositorio clonado, para cambiárselo será :

$ git clone git://github.com/schacon/nomreRepositorio.git myRepositorio

En ambos casos se creará un directorio .git en cuyo interior, descarga toda la información del repositorio, y saca una copia de trabajo de la última versión.

Git permite usar distintos protocolos de transferencia, http y https, requiriéndose la autenticación ó configuración del ssh en este ultimo caso.

Cada archivo del directorio de trabajo puede estar bajo seguimiento (tracked), o sin seguimiento (untracked). Los archivos bajo seguimiento son los que existían en la última instantánea; que a su vez pueden estar sin modificaciones, modificados, ó preparados. Los archivos sin seguimiento son todos los demás —cualquier archivo del directorio que no estuviese en la última instantánea ni en el área de preparación—.



\* Para conocer el estado de los archivos se usa git status . Si se hace nada más clonar:

$ git status

# On branch master nothing to commit, working directory clean

Sería un directorio de trabajo limpio, sin archivos bajo seguimiento ni modificados.

A menudo existirán archivos que no se quiere que Git añada, estos se indicarán, nominalmente, o por medio de reglas en .gitignore (fichero creado en el dir de trabajo)

Ahora con git commit se confirman los cambios incorporados y abrirá el editor por defecto para pedir un comentario salvo que se ejecute con la opción -m “Indicando en un entrecomillado el texto comentario, obligatorio, a adjuntar al cambio subido”.

El siguiente gráfico muestra como es el paso de estados entre las distintas áreas.

|  |  |
| --- | --- |
|  |  |

Para ver las confirmaciones realizadas se utiliza git log que muestra las confirmaciones hechas sobre el repositorio en orden cronológico inverso. Es decir, las más recientes primero.

Con la opción -p muestra las diferencias introducidas en cada confirmación.

Con la opción -n muestra las n últimas confirmaciones únicamente.

Con la opción -stat imprime tras cada confirmación una lista de archivos modificados, indicando cuántos han sido modificados y cuántas líneas han sido añadidas y eliminadas para cada uno de ellos, y un resumen de toda esta información.

La opción más interesante es --pretty, que modifica el formato de la salida que puede ser combinado con format, que te permite especificar tu propio formato puede resultar especialmente útil si estás generando una salida para que sea analizada por otro programa, con la opción oneline que imprime cada confirmación en una única línea, etc...

git log --pretty=format:"%h - %an, %ar : %s"

Las opciones onelineyformatson especialmente útiles combinadas con otra opción –graph que añade un pequeño gráfico ASCII mostrando tu histórico de ramificaciones y uniones. $ git log --pretty=format:"%h %s" –graph

Para ver lo que has modificado pero aún no has preparado,  git diff que compara lo que hay en el directorio de trabajo con lo que hay en tu área de preparación.Es importante indicar que git diff por sí solo no muestra todos los cambios hechos desde tu última confirmación —sólo los cambios que todavía no están preparados—. Esto puede resultar desconcertante, porque si has preparado todos los cambios, git diff no mostrará nada. Y git diff --cached para ver los cambios que llevas preparados hasta ahora, también con git diff –-staged compara tus cambios preparados con tu última confirmación. Recordar que cualquier cosa que todavía esté sin preparar —cualquier archivo creado o modificado, y sobre el que no se haya ejecutado git add desde su última edición— no se incluirá en la confirmación. Se mantendrán como modificados en tu disco.

Si quiero ver diferencias entre dos commit sucesivos: git diff –head

Git no hace un seguimiento explicito del movimiento de archivos. Si renombras un archivo, en Git no se almacena ningún metadato que indique que lo has renombrado, para renombrar un archivo en Git $ git mv file\_from file\_to esto es equivalente a ejecutar algo así:

$ mv README.txt README

$ git rm README.txt

$ git add README

Para eliminar un archivo de Git, se debe eliminar de los archivos bajo seguimiento (se debe eliminar del área de preparación), y después confirmar. El comando git rm se encarga de eso, y también elimina el archivo del directorio de trabajo, para que no aparezca entre los archivos sin seguimiento.

Otra cosa que puede interesar es mantener el archivo en el directorio de trabajo, pero eliminarlo del área de preparación. (que quieras mantener el archivo en tu disco duro, pero interrumpir su seguimiento por parte de Git). Esto resulta útil cuando se olvidó de añadir al archivo .gitignore y se incorporó accidentalmente. Para esto, usa la opción --cached. $ git rm --cached readme.txt

El comando git rm acepta archivos, directorios, y patrones glob. Es decir, se podría hacer algo así:$ git rm log/\\*.log Atención al (\) antes del \*. Es necesario debido a que Git hace su propia expansión de rutas, además de la expansión que hace la shell. En la consola del sistema de Windows, esta barra debe de ser omitida. Este comando elimina todos los archivos con la extensión .log en el directorio log/.

$ git rm \\*~ elimina todos los archivos que terminan en ~.

**Realizando cambios.**

El comentario, obligatorio del commit, puede ser cambiado ejecutando de nuevo commit con la opción –amend, si no se ha vuelto a realizar un commit desde la vez anterior.

$ git commit -m 'initial commit'

$ git add forgotten\_file

$ git commit --amend

Para eliminar un archivo de Git, se debe eliminar de los archivos bajo seguimiento y después confirmarlo, para ello se usa el comando si no se hace así, git rm no se eliminará del repositorio y volverá a traerlo.

El comando que se usa para determinar el estado de ambas áreas te recuerda como deshacer sus modificaciones.

$ git add .

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: README.txt

# modified: benchmarks.rb

Te dice que uses git reset HEAD <archivo>... para sacar un archivo del área de preparación.$ git reset HEAD fichero.extension

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory) #

# modified: benchmarks.rb

También como, si no se quiere mantener las modificaciones que has hecho sobre el archivo afectado. Puedes deshacerlas fácilmente y revertir el archivo al estado en el que se encontraba cuando se hizo la última confirmación. git checkout – fichero.extension Es importante entender que git checkout -- [archivo] es un comando peligroso. Cualquier cambio realizado a ese archivo desaparecerá - se sobreescribe con otro archivo. Nunca se utiliza este comando a menos de estar absolutamente seguro de que ya no se quiere el archivo. El comando checkout se usa para copiar archivos de los commits (o desde stage) al directorio de trabajo, y para opcionalmente intercambiar branches.

**Repositorios remotos**

git remote. Permite ver qué repositorios remotos están configurados, la opción-v, muestra la URL asociada a cada repositorio remoto:

$ git remote -v

origin git://github.com/schacon/ticgit.git (fetch)

origin git://github.com/schacon/ticgit.git (push)

Añadir repositorio git remote add [url]:

Añadir repositorio con alias git remote add [nombre] [url]:

el alias permite referenciarlo en la línea de comandos, sin tener que usar toda la URL.

$ git remote add pb git://github.com/paulboone/ticgit.git

$ git remote -v

origin git://github.com/schacon/ticgit.git

pb git://github.com/paulboone/ticgit.git

Ahora puedes usar la cadena "pb", en lugar de toda la URL.

git fetch pb

**Recibiendo de tus repositorios remotos**

$ git fetch [remote-name]

Así se recuperan todos los datos del proyecto remoto que no se tengan todavía. Después de hacer esto, se debería tener referencias a todas las ramas del repositorio remoto.

Si se clona un repositorio, el comando añade automáticamente ese repositorio remoto con el nombre de "origin". Por tanto, git fetch origin recupera toda la información enviada a ese servidor desde que se clonó (o desde la última vez que se ejecutó fetch). Es importante tener en cuenta que el comando fetch sólo recupera la información y la pone en el repositorio local —no la une automáticamente con tu trabajo ni modifica aquello en lo que estás trabajando.

El comando git clone automáticamente configura la rama local maestra para que siga la rama remota maestra del servidor del cual se clonó (asumiendo que el repositorio remoto tiene una rama maestra).

Si se ha configurado una rama para seguir otra rama remota, se puede usar el comando git pull para recuperar y unir automáticamente la rama remota con la rama actual. Al ejecutar git pull, por lo general se recupera la información del servidor del que se clonó y automáticamente se intenta unir con el código con el que se está trabajando actualmente.

**Enviando a tus repositorios remotos**

Para poder compartir un proyecto hay que enviarlo a un repositorio remoto esto se hace con: git push [nombre-remoto][nombre-rama]. Pero si lo que quieres enviar es la rama maestra (master) al servidor origen (origin) se realiza con $ git push origin master

Esto funcionará solo si el punto de partida fue clonando desde un servidor en el que se tienes permiso de escritura, y nadie ha enviado información mientras tanto. Si otra persona y tú clonáis a la vez, y él envía su información y luego envías tú la tuya, tu envío será rechazado. Primero tendrías que bajarte su trabajo e incorporarlo en el tuyo para que se te permita hacer un envío.

Para ver más información acerca de un repositorio remoto en particular, se puede usar el comando git remote show [nombre]. Si lo ejecutas pasándole origin como nombre de un repositorio, obtienes algo así:

$ git remote show origin

\* remote origin URL: git://github.com/schacon/ticgit.git

Remote branch merged with 'git pull' while on branch master master

Tracked remote branches master ticgit

Esto lista la URL del repositorio remoto, así como información sobre las ramas bajo seguimiento. Este comando te recuerda que si se está en la rama maestra y se ejecuta git pull, automáticamente unirá los cambios a la rama maestra del remoto después de haber recuperado todas las referencias remotas. También lista todas las referencias remotas que ha recibido. (Este es un ejemplo sencillo, el más habitual, pero usando Git de forma avanzada, puede mostrar mucha más información).

**Eliminar y renombrar repositorios remotos**

git remote rename permite renombrar una referencia a un repositorio remoto.

git remote rm permite eliminar una referencia en un repositorio remoto ( se ha movido el servidor o ya no se está usando un determinado mirror, o un contribuidor ha dejado de contribuir).

|  |  |
| --- | --- |
| $ git remote rename pb paul | $ git remote rm paul |
| $ git remote | $ git remote |
| origin | origin |
| paul |  |

**Etiquetas**

Git tiene la habilidad de etiquetar (tag) que son puntos específicos. Generalmente la se usa esta funcionalidad para marcar puntos donde se ha lanzado alguna versión.

Listar las etiquetas disponibles  git tag

Buscar etiquetas de acuerdo a un patrón en particular.

$ git tag -l 'v1.4.2.\*'

v1.4.2.1

v1.4.2.2

v1.4.2.3

v1.4.2.4

Git dispone de etiquetas ligeras y anotadas. Una etiqueta ligera es un puntero a una confirmación específica. Las etiquetas anotadas son objetos completos que tienen suma de comprobación, nombre del etiquetador, correo electrónico, fecha y mensaje de etiquetado; pueden estar firmadas y verificadas con GPG (GNU Privacy Guard). Se recomienda crear etiquetas anotadas para disponer de toda esta información.

Se crea una etiqueta anotada con git tag -a se puede firmar la etiqueta con GPG, si se tiene una clave privada, usando -s en vez de -a. El parámetro -m especifica el mensaje a almacenar.

$ git tag -a v1.4 -m 'my version 1.4'

$ git tag

v0.1

v1.3

v1.4

Se pueden ver los datos de la etiqueta junto a la confirmación que fue etiquetada con: git show

$ git show v1.4

tag v1.4

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 14:45:11 2009 -0800

my version 1.4

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'

Para crear una etiqueta ligera no se añade ninguna opción (-a, -s o -m) al crearla

$ git tag v1.4-l

Al ejecutar git show en la etiqueta, no se verá información extra, sólo la confirmación.

$ git show v1.4-l

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'

Pueden etiquetarse commits antiguos, basta con incorporar la suma de comprobación de la confirmación (o una parte de la misma) al final del comando:

$ git log --pretty=oneline

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment' 9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile 964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo 8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme

$ git tag -a v1.2 -m 'version 1.2' 9fceb02

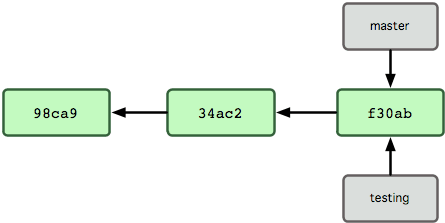
Por defecto, el comando git push no transfiere las etiquetas a servidores remotos. Hay que enviarlas explicitamente a un servidor compartido después de haberlas creado. Este proceso es igual a compartir ramas remotas  git push origin [tagname].

Si hay muchas etiquetas que quieres enviar, la opción --tags en el comando git push. transfiere todas las etiquetas que no estén aún en el servidor remoto.

Cuando se crea una confirmación con git commit, Git realiza sumas de control de cada subcarpeta (si las hay), y guarda en el repositorio Git una copia de cada uno de los archivos contenidos en ella/s. Después, Git crea un objeto de confirmación con los metadatos pertinentes y un apuntador al nodo correspondiente del árbol de proyecto. Esto permitirá poder regenerar posteriormente dicha instantánea cuando sea necesario.

Una rama Git es simplemente un apuntador móvil apuntando a una de esas confirmaciones. La rama por defecto de Git es la rama master. Con la primera confirmación de cambios que realicemos, se creará esta rama principal master apuntando a dicha confirmación. En cada confirmación de cambios que realicemos, la rama irá avanzando automáticamente. Y la rama master apuntará siempre a la última confirmación realizada. Cuando se crea una nueva rama, simplemente se crea un nuevo apuntador para que se pueda mover libremente. para crear una nueva rama se usa git branch

$ git branch testing



Git mediante un apuntador especial denominado HEAD, comentar que este HEAD es totalmente distinto al concepto de HEAD en otros sistemas de control de cambios como Subversion o CVS, sabe en que rama se encuentra.

\* Para saltar de una rama a otra, se utiliza el comando git checkout

$ git checkout testing

|  |  |
| --- | --- |
|  |  |

Tras realizar otra confirmación de cambios:

$ vim test.rb

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

Esto hace que la rama testing avanza, mientras que la rama master permanece en la confirmación donde estaba cuando se lanzó el comando git checkout para saltar.

|  |  |
| --- | --- |
|  | Volver a master: $ git checkout master |

Este comando realiza dos acciones: Mueve el apuntador HEAD de nuevo a la rama master, y revierte los archivos del directorio de trabajo; los deja como estaban en la última instantánea confirmada en dicha rama master. Esto supone que los cambios desde este momento en adelante divergirán de la antigua versión del proyecto.

$ vim test.rb

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

|  |
| --- |
|  |

Debido a que una rama Git es realmente un simple archivo que contiene los 40 caracteres de una suma de control SHA-1, (representando la confirmación de cambios a la que apunta), no cuesta nada el crear y destruir ramas en Git. Crear una nueva rama es tan rápido y simple como escribir 41 bytes en un archivo, (40 caracteres y un retorno de carro).

Esto contrasta fuertemente con los métodos de ramificación usados por otros sistemas de control de versiones. En los que crear una nueva rama supone el copiar todos los archivos del proyecto a una nueva carpeta adicional. Lo que puede llevar segundos o incluso minutos, dependiendo del tamaño del proyecto. Mientras que en Git el proceso es siempre instantáneo.

Es habitual que según se está trabajando en un apartado de un proyecto, donde normalmente el espacio de trabajo suele estar en un estado inconsistente se necesite cambiar de rama durante un tiempo para solucionar algún otro tema urgente. Esto plantea el problema de confirmar cambios en un trabajo medio hecho, simplemente para poder volver a ese punto más tarde. Y su solución es el comando 'git stash'. Este comando de guardado rápido (stashing) toma el estado del espacio de trabajo, con todas las modificaciones en los archivos bajo control de cambios, y lo guarda en una pila provisional. Posteriormente se podrán recuperar y aplicar de nuevo al espacio de trabajo.

Así si tengo que abandonar un trabajo en un momento dado con 'git status', se podría observar el estado inconsistente actual tal como:

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: index.html

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: lib/simplegit.rb

#

Si necesito en este momento cambiar de rama, pero sin confirmar los cambios realizados hasta ahora, la solución es un guardado rápido provisional utilizando el comando 'git stash' y enviando el nuevo grupo de cambios a la pila de guardado rápido:

$ git stash

Saved working directory and index state \

"WIP on master: 049d078 added the index file"

HEAD is now at 049d078 added the index file

(To restore them type "git stash apply")

Con ello, se limpia el área de trabajo:

$ git status

# On branch master

nothing to commit, working directory clean

Y puedo cambiar de rama para ponerse a trabajar en cualquier otra parte.

Puedo más tarde volver y con git stash apply retomar mi actividad.

Para ver el contenido de dicha pila de guardado rápido mediante 'git stash list':

$ git stash list

stash@{0}: WIP on master: 049d078 added the index file

stash@{1}: WIP on master: c264051... Revert "added file\_size"

stash@{2}: WIP on master: 21d80a5... added number to log

En este ejemplo, se habían realizado dos guardados rápidos anteriores, por lo que se ven tres grupos de cambios guardados en la pila. Con el comando 'git stash apply', tal y como se indica en la salida del comando stash original, se pueden volver a aplicar los últimos cambios recién guardados. Si lo que se desea es reaplicar alguno de los grupos más antiguos de cambios, se ha de indicar expresamente: git stash apply stash@{2} Si no se indica ningún grupo concreto, Git asume que se desea reaplicar el grupo de cambios más reciente de entre los guardados en la pila.

$ git stash apply

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: index.html # modified: lib/simplegit.rb

#

Como se ve en la salida del comando, Git vueve a aplicar los correspondientes cambios en los archivos que estaban modificados. Pero no conserva la información de lo que estaba o no estaba añadido al área de preparación. Las modificaciones sobre los archivos serán aplicadas; pero no así el estado de preparación. Para conseguir esto último, es necesario emplear la opción --index del comando git stash apply. Con ella se le indica que debe intentar reaplicar también el estado de preparación de los archivos. Y asi se puede conseguir volver exactamente al punto original:

Los comandos git stash apply tan solo recupera los cambios almacenados en la pila de guardado rápido, sin afectar al estado de la pila. Es decir, los cambios siguen estando guardados en la pila. Para quitarlos de ahí, es necesario lanzar expresamente el comando git stash drope indicar el número de guardado a borrar de la pila ó es posible utilizar el comando git stash pop, que aplica cambios de un guardado y lo retira inmediatamente de la pila.

Si se almacena rápidamente (stash) un cierto trabajo, se deja en la pila durante bastante tiempo, y se continua mientras tanto con otros trabajos sobre la misma rama. Es muy posible que se presenten problemas al tratar de reaplicar los cambios guardados tiempo atrás. Si para recuperar esos cambios se ha de modificar un archivo que también haya sido modificado en los trabajos posteriores, se dará un conflicto de fusión (merge conflict) y será preciso resolverlo manualmente. Una forma más sencilla de reaplicar cambios es utilizando el comando git stash branch. Este comando crea una nueva rama, extrayendo (checkout) la confirmación de cambios original en la que se estaba cuando los cambios fueron guardados en la pila, reaplica estos sobre dicha rama y los borra de la pila si se consigue completar el proceso con éxito.

**Recordatorio de comandos mas usuales**

git clone <https://github.com/libgit2/libgit2> Clonar un repositorio:

git fetch: Descarga los cambios realizados en el repositorio remoto.

git merge *<nombre\_rama>*: Fusiona en la rama en la que estás situado los cambios realizados en la rama “nombre\_rama”.

git pull: Unifica los comandos *fetch* y *merge* en un único comando.

git commit -am "<mensaje>" Confirma los cambios realizados. El “mensaje” generalmente se usa para asociar al *commit* una breve descripción de los cambios realizados.

git **push** origin *<nombre\_rama>*: Sube la rama “nombre\_rama” al servidor remoto.

git status: Muestra el estado actual de la rama, como los cambios que hay (de los que no se ha hecho commit todavía)

git branch: Lista todas las ramas locales.

git remote prune origin: Actualiza tu repositorio remoto en caso que algún otro desarrollador haya eliminado alguna rama remota.

git reset --hard HEAD: Elimina los cambios realizados que aún no se hayan hecho *commit*.

git revert *<hash\_commit>*: Revierte el *commit* realizado, identificado por el código “hash\_commit”.

git stash Realiza un [grabado rápido provisional](https://git-scm.com/book/es/v1/Las-herramientas-de-Git-Guardado-rápido-provisional) de los cambios

git stash pop, aplica cambios de un guardado (recupearamos a donde lo dejamos)y lo retira de la pila.

**Enlaces y tutoriales**

Lo mejor es ir al [libro oficial de Git](https://git-scm.com/book/es/v1) ya mencionado y empezar por el segundo capítulo. Está en castellano.

Este es un tutorial independiente que parece bastante claro

[http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/es/apa.html#\_microsoft\_windows](http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/es/apa.html" \l "_microsoft_windows)

Otro tutorial que presume de sencillo (

<http://rogerdudler.github.io/git-guide/index.es.html>

Tutorial on-line de GITHUB :<https://try.github.io/levels/1/challenges/1>