# Git - Manual de usuario

Version 1

Cooperativa deTrabajo Fiqus LTDA
http://www.fiqus.com

# Contents

# Introducción: Entendiendo que es y como funciona Git.

### ¿Qué es un sistema de control de versiones?

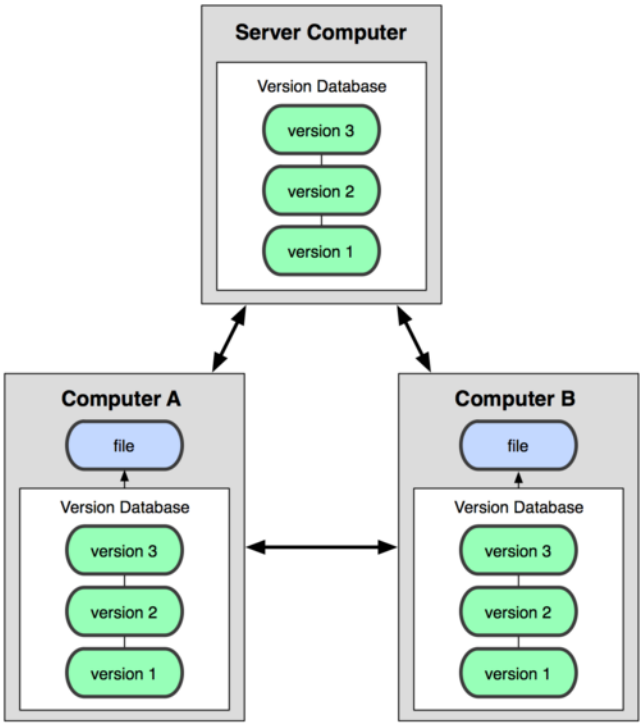
Un sistema de control de versiones (CVS) te permite realizar un seguimiento de la historia de una colección de archivos y además incluye la funcionalidad de revertir la colección de archivos actual hacia una versión anterior. Cada versión podría considerarse como una fotografía del estado de la colección en un momento determinado de tiempo. La colección de archivos usualmente es código fuente de algún lenguaje de programación, sin embargo, un sistema de control de versiones funciona con cualquier tipo de archivo. Tanto la colección de archivos como su historia completa están guardados en un repositorio.

### ¿Qué es un sistema de control de versiones distribuida?

En un sistema de control de versiones distribuida hay un servidor central para almacenar el repositorio y cada usuario puede hacer una copia completa del repositorio central mediante un proceso llamado "clonación". Cada repositorio clonado es una copia completa del repositorio central y por ser una copia completa posee la mismas funcionalidades que el repositorio original, es decir, contiene la historia completa de la colección de archivos. Cada repositorio clonado puede a su vez intercambiar las versiones de sus archivos con otros repositorios clonados del mismo nodo padre (ya que deben tener la misma estructura), enviando sus cambios y recibiendo los del otro en forma directa o a través del repositorio central.

**¿Qué es Git?**

Git es un sistema de control de versiones distribuida que se origina a partir del desarrollo del kernel de Linux y es usado por muchos proyectos populares Open Source como ser Android o Eclipse, así como tantos otros proyectos comerciales. Entonces, la principal diferencia entre Git y cualquier otro sistema de control de versiones es cómo Git modela sus datos. Conceptualmente, la mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas modelan la información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo. En cambio, Git modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente hace una foto del estado 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.

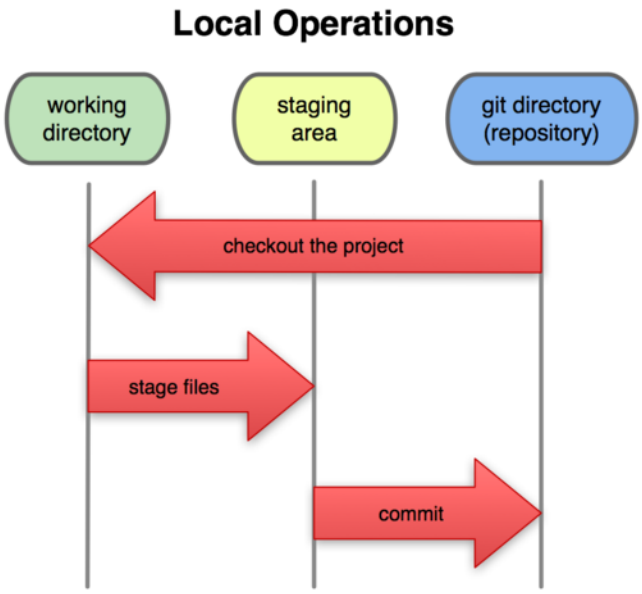



En la imagen se ve claramente como en un sistema de control de versiones distribuida, cada repositorio clonado es una copia exacta del repositorio padre.

**El repositorio local**

Luego de clonar o crear un repositorio el usuario tiene una copia completa del repositorio, y puede realizar operaciones de control de versiones contra este repositorio local, como por ejemplo crear nuevas versiones, revertir cambios, etc. El flujo de trabajo básico en Git es algo así:

- Modificas una serie de archivos en tu directorio de trabajo (working directory).
- Añadís instantáneas de los archivos a tu área de preparación (staging area).
- Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esa instantánea de manera permanente en tu directorio de Git (git directory).




Hay dos tipos de repositorios en Git:

- Repositorios "bare", son generalmente usados en los servidores, aunque puede generarse en cualquier directorio local de quien desee crearlo. Se utiliza para compartir cambios provenientes de diferentes desarrolladores, no se utiliza para crear u compartir cambios desde sí mismo, ya que no cuenta con esas funcionalidades.
- Repositorios de trabajo, permiten crear archivos nuevos, generar cambios a los archivos existentes y a su vez crear nuevas versiones en el repositorio.

Si se desea eliminar el repositorio local, sólo debe borrarse la carpeta en la que se encuentra.

### Repositorios remotos

Git permite a los usuarios sincronizar el repositorio local con otros repositorios remotos al ambiente de trabajo local. Los usuarios que posean los permisos necesarios pueden hacer un "push" (comando utilizado para subir cambios locales) de los cambios locales al repositorio remoto. A su vez también pueden hacer un "fetch" o "pull" (comandos para buscar cambios) de los cambios realizados en repositorios remotos al local.

### Repaso de algunos conceptos útiles

- Repositorio: Un repositorio contiene la historia, las diferentes versiones en el tiempo y todas las diferentes ramas. En Git cada copia del repositorio es un repositorio completo. Si el repositorio en el que estás trabajando no es creado con la opción "bare", entonces permite hacer un checkout de las revisiones que desees en tu repositorio local.
- Working tree: Posee el contenido de un commit que se puede obtener haciendo un checkout desde un repositorio git. Luego uno puede modificar ese contenido y hacer un nuevo commit con los cambios al repositorio.
- Branch (rama): Un branch es un puntero con un nombre determinado por el usuario que apunta a un commit. Posicionarse en un branch utilizando git es denominado como "hacer un checkout" de ese branch. Si estás trabajando en un determinado branch, la creación de un nuevo commit hace avanzar el puntero a esta nueva instancia. Cada commit conoce sus antecesores así como a sus sucesores en caso de tenerlos. Uno de los branches es el default, generalmente llamado master.
- Tag: Un tag apunta a un commit que unívocamente identifica una versión del repositorio. Con un tag, podés tener un puntero con nombre al que siempre puedas revertir los cambios. Por ejemplo, la versión de 25.01.2009 del branch "testing".
- Commit: Vos commiteas los cambios a un repositorio. Esto crea un nuevo objeto commit en el repositorio que unívocamente identifica una nueva versión del contenido del repositorio. Esta revisión puede ser consultada posteriormente, por ejemplo si uno quiere ver el código fuente de una versión anterior. Cada commit posee metadata que nos informa acerca del autor, la fecha y otros datos que nos pueden resultar prácticos a la hora de tratar de encontrar uno determinado.

- URL: Una URL en Git determina la ubicación de un repositorio. Revisión: Representa una versión del código fuente. Git implementa las revisiones de la misma manera que los objetos commit.
- HEAD: Es un objeto simbólico que apunta generalmente al branch sobre el que estamos trabajando (lo que también conocemos como "checked out branch"). Si uno cambia de un branch al otro el HEAD apunta al último commit del branch seleccionado. Si uno hace un checkout de un determinado commit, el HEAD apunta a ese commit.
- Staging area: Es el lugar en el que se almacenan los cambios del working tree previos al commit. Es decir, contiene el set de cambios relevantes para el próximo commit.
- Index: Es un término alternativo para referirnos al staging area.

# Comandos básicos en Git

## Obteniendo y creando proyectos

Como ya lo mencionamos previamente, para hacer cualquier cosa con Git, primero hay que tener un repositorio creado, que es donde Git almacena los cambios que haces al código. Hay dos maneras de obtener un repositorio Git. Una forma es simplemente inicializar uno nuevo desde un carpeta existente, como un nuevo proyecto. La segunda forma consiste en clonar un repositorio Git público, como si quisieras una copia o quisieras trabajar con alguien en un proyecto.

### Git init - inicializar una carpeta como un repositorio Git

Para crear un repositorio en una carpeta existente de archivos, podes ejecutar el comando git init en esa carpeta. Por ejemplo, digamos que tenemos una carpeta con algunos archivos adentro, así:

```
$ cd repositorio_nuevo
$ ls
README hello.rb
```

Se trata de un proyecto en el que estamos escribiendo ejemplos del "Hola Mundo" en todos los idiomas. Hasta el momento, sólo tenemos el código escrito en Ruby. Para iniciar el control de versión con Git, podemos ejecutar git init.

```
$ git init
Initialized empty Git repository in /repositorio_nuevo/.git/
```

Ahora se puede ver que hay una subcarpeta oculta llamada ".git" en el proyecto. Este es tu repositorio donde se almacenan todos los cambios del proyecto.

```
$ ls -a
.  ..  .git  README  hello.rb
```

En pocas palabras, se usa "git init" para convertir una carpeta existente en un nuevo repositorio Git. Se puede hacer esto en cualquier carpeta en cualquier momento.

> Nota: Cuando se crea un repositorio con init, la primera vez que haces un push, tenes que correr `git push origin master`. Además hay que crear el branch master y hacerle un commit con los archivos agregados para que aparezca.

### Git clone - copiar un repositorio Git

Si tenes que colaborar con alguien en un proyecto, o si deseas obtener una copia de un proyecto para poder ver o usar el código, debes clonarlo. Para lograrlo sólo tenes que ejecutar el comando git clone [url] con la URL del proyecto que deseas copiar.

```
$ git clone git://github.com/schacon/simplegit.git

Initialized empty Git repository in /private/tmp/simplegit/.git/
remote: Counting objects: 100, done.
remote: Compressing objects: 100% (86/86), done.
remote: Total 100 (delta 35), reused 0 (delta 0)
Receiving objects: 100% (100/100), 9.51 KiB, done.
Resolving deltas: 100% (35/35), done.

$ cd simplegit/
$ ls
README  Rakefile lib
```

Esto copiará toda la historia de este proyecto, con lo cual lo tendrás a nivel local y te dará una carpeta de trabajo de la rama principal de ese proyecto para que puedas ver el código o empezar a editarlo. Si cambias a la nueva carpeta, podes ver la subcarpeta .git (ahí es donde están todos los datos del proyecto).

```
$ ls -a
.  ..  .git  README  Rakefile lib
$ cd .git
```

```
$ ls
HEAD  description info  packed-refs
branches  hooks  logs  refs
config  index  objects
```

Por defecto, Git va a crear una carpeta que tiene el mismo nombre que el proyecto en la dirección que le indiques - básicamente cualquier cosa que aparezca después de la última barra de la URL. Si queres un nombre diferente, podes ponerlo al final del comando, después de la URL. Entonces, se utiliza el comando git clone para obtener una copia local de un repositorio Git, para que puedas verlo y empezar a modificarlo.

## Agregando y subiendo cambios

Un concepto que no se debe perder de vista es que Git tiene un índice (index), que funciona como una especie de área de desarrollo (staging area) para los cambios que uno va generando en los archivos a medida que va trabajando. Esto permite terminar de darle forma a los cambios que uno va realizando en su área de trabajo, en lugar de que la herramienta automáticamente te arme los commits.

### git add - agregar los contenidos de archivos al staging area

En Git tenes que agregar previamente los cambios realizados al staging area para luego poder hacer el commit correspondiente (confirmar los cambios). Si el archivo que estás agregando es nuevo, entonces tenes que correr el comando git add para añadirlo inicialmente en tu staging area. Si el archivo ya está en "seguimiento" también tenes que correr el mismo comando (git add), no para agregar el archivo, sino para agregar las nuevas modificaciones en tu staging area. Volviendo al ejemplo de Hola Mundo, una vez iniciado el proyecto, empezaremos a agregarle archivos y para ello correremos el comando git add. Podemos usar el comando git status para ver en que estado está nuestro proyecto.

```
$ git status -s
?? README
?? hello.rb
```

En este momento tenemos dos archivos que no están bajo el seguimiento de Git. Ahora podemos agregarlos.

```
$ git add README hello.rb
```

Ahora si corremos el comando git status nuevamente, veremos que fueron agregados.

```
$ git status -s
A  README
A  hello.rb
```

Es también común agregar en forma recursiva todos los archivos en un nuevo proyecto especificando sólo el directorio de trabajo que se desea agregar, como por ejemplo:

```
$ git add .
```

De esta manera, Git comenzará a agregar todos los archivos que se encuentran en el directorio especificado. En el ejemplo particular que estamos viendo, "git add ." hubiese hecho lo mismo que al escribir:

```
$ git add README hello.rb
```

Para el caso también "git add *" hubiese hecho lo mismo, además de meterse en subdirectorios que estuviesen dentro del directorio actual. Entonces, si ahora editamos uno de estos archivos y corremos el comando "git status" nuevamente, veríamos algo como lo siguiente:

```
$ vim README
$ git status -s
AM README
A  hello.rb
```

El estado 'AM' significa que el archivo fue modificado en el disco desde que lo agregamos. Esto significa que si hacemos un commit de nuestro cambios hasta este momento, estaríamos grabando la versión del archivo que teníamos en el momento que corrimos git add, no la versión que está en nuestro disco. Git no asume que lo que uno quiere subir es explícitamente lo que tenemos en el disco, uno le tiene que avisar a Git con el comando git add nuevamente.

En pocas palabras, uno corre git add en un archivo cuando quiere incluir cualquier cambio que le hayas hecho a tu próximo commit. Cualquier cambio que no hayas agregado con este comando, no será incluído en el commit, esta particularidad nos ofrece la posibilidad de armar los commits de una forma más certera, con un alto nivel de detalle acerca de qué queremos incluir y qué no.

### git status – estado de los archivos en el directorio de trabajo y en el staging area

Tal como lo mencionamos en la sección anterior, uno puede usar este comando para ver el estado de su staging area comparado con el código que se encuentra en

su directorio de trabajo. Usando la opción "-s" nos mostrará la forma abreviada del informe, en cambio si no usamos esa opción, nos mostrará más información del contexto. Aquí hay un ejemplo del resultado con y sin la opción:

```
$ git status -s
AM README
A  hello.rb
```

Sin la opción se vería algo así:

```
$ git status -s
$ git status
# On branch master
# Initial commit
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
# new file:   README
# new file:   hello.rb
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
# modified:   README
```

Se ve que la salida corta es más compacta, pero la salida larga tiene algunos datos útiles que nos pueden llegar a servir en algún momento. Git además te informará de los archivos que fueron borrados, agregados o modificados desde tu último commit.

```
$ git status -s
M  README
D hello.rb
```

En el siguiente ejemplo se pueden ver dos columnas, la primera es para el staging area, la segunda es para el directorio de trabajo. Entonces, por ejemplo si tenés el README file en el stage y después lo modificas nuevamente sin correr "git add" nuevamente, vas a ver algo así:

```
$ git status -s
MM README
 D hello.rb
```

**git diff – muestra diferencias entre el stage y lo modificado que no esta en el stage**

Hay dos usos diferentes para el comando git diff. El uso que veremos a continuación lo vamos a usar para describir los cambios que están en stage o que fueron modificados pero no están en el stage.

**git diff - muestra las diferencias de cambios que no están en el stage.** Sin ningún argumento adicional, un simple git diff mostrará en formato unificado, qué código o contenido ha cambiado en el proyecto desde el último commit, que todavía no se metió en el stage area (no se le hizo el add) para el próximo commit.

```
$ vim hello.rb
$ git status -s
 M hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index d62ac43..8d15d50 100644
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 class HelloWorld
   def self.hello
-    puts "hello world"   #indica que se reemplazó esta línea por la de abajo. (- por +)
+    puts "hola mundo"
   end
 end
```

Entonces, "git diff" te muestra cuáles son los cambios, línea por línea. Por lo general es un buen comando de seguimiento de git status.

**git diff HEAD – muestra las diferencias de todos los cambios, estén o no en el staging** ste comando nos permite básicamente ver la diferencia entre nuestro directorio de trabajo y el último commit, ignorando el staging area. Si hacemos un nuevo cambio en nuestro archivo hello.rb, entonces tendremos algunos cambios en el directorio de trabajo y otros en el último commit, si hacemos un "diff HEAD" entonces nos mostrará las diferencias entre lo último subido y el directorio, sin tener en cuenta lo que está en el staging area.

**git diff –stat – muestra una lista de cambios en lugar de un diff completo por archivo** Si no queremos la salida completa del diff, pero queremos más que la salida del git status, podemos utilizar la opción - stat, lo que nos

dará un resumen de los cambios en su lugar. Acá está el mismo ejemplo anterior, pero con la opción – stat.

```
$ git status -s
MM hello.rb
$ git diff --stat
 hello.rb |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

También se puede proporcionar una ruta de archivo al final de cualquiera de estas opciones para limitar la salida del diff a un archivo o subdirectorio.

### git commit – graba una instantánea del staging area

Una vez que agregaste los cambios al staging area, vas a necesitar confirmar esos cambios con el comando add, pero para poder hacerlo git necesita grabar tus datos personales para poder hacer un seguimiento de los cambios. Entonces, el primer paso es configurar estos datos:

```
$ git config --global user.name 'Tu nombre'
$ git config --global user.email tumail@algundominio.com
```

A continuación vamos a meter en el stage los cambios y luego a confirmarlos, en este ejemplo se usa la opción -m para agregar el mensaje al mismo tiempo que confirmamos los cambios:

```
$ git add hello.rb
$ git status -s
M  hello.rb
$ git commit -m 'mis cambios al archivo hola mundo'
[master 68aa034] my hola mundo changes
 1 files changed, 2 insertions(+), 1 deletions(-)
```

Ahora ya grabamos la instantánea del archivo. Si corremos el comando git status nuevamente, veremos que tenemos nuevamente un directorio de trabajo "limpio", lo que significa que no hicimos ningún cambio desde nuestro último commit.

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Si no ingresamos la opción -m git va a intentar abrir un editor de texto para que escribas el mensaje. Algo como esto:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   hello.rb
#
~
~
".git/COMMIT_EDITMSG" 9L, 257C
```

En este punto se agrega el mensaje de cambios en la parte superior del documento. Todas las líneas que comienzan con '#' se ignoran. En general, es muy importante escribir un buen mensaje de confirmación.

**git commit –a – agrega automáticamente en el staging area los cambios de archivos que están en seguimiento** Si te parece que correr el comando git add cada vez que modificas un archivo es demasiado engorroso, Git te permite saltar esta parte con la opción -a. Esto le dice a Git, que ejecute el git add en cualquier archivo que ya esta en "seguimiento" - es decir, cualquier archivo que se encontraba en tu último commit y después fue modificado. Esto te permite hacer un flujo de trabajo más al estilo de Subversion, sólo tenés que editar los archivos y luego ejecutar git commit -a cuando queres que todo lo que ha cambiado entre en el commit. Igualmente siempre tenes que ejecutar git add para iniciar el seguimiento de nuevos archivos.

```
$ vim hello.rb
$ git status -s
 M  hello.rb
$ git commit -m 'cambios en el archivo hola'
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   hello.rb
#
no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -am 'changes to hello file'
[master 78b2670] changes to hello file
 1 files changed, 2 insertions(+), 1 deletions(-)
```

Este comando te permite cerrar el workflow de trabajo con git: haces cambios a tu archivos, después usas git add para meter esos cambios en el staging area,

con git status y git diff consultas las diferencias y los cambios y finalmente con git commit guardas una instantánea con el estado del conjunto de archivos en ese momento determinado.

### git reset – deshace cambios y commits

Este es probablemente el comando más confuso, pero puede ser muy útil cuando cometas algún error.

**git reset HEAD - deshacer el último commit y sacar del staging area los archivos modificados.** En general, se usa para sacar algo que agregaste al staging area (le hiciste el add) por accidente. Si ya hiciste el commit o sólo si lo agregaste, entonces este comando te ayuda a deshacer el commit (en caso de que lo hayas hecho) y a sacar del índice a los archivos que le indiques. Supongamos que modificaste dos archivos, y queres grabarlos en dos commits diferentes, entonces tenes que agregar y confirmar uno y después agregar y confirmar el otro. Si accidentalmente agregas los dos al mismo tiempo, ¿Cómo sacas uno? Lo haces con git reset HEAD –nombreArchivo. Técnicamente no es necesario añadir el — pero se usa para avisarle a Git que terminaste de listar las opciones y empezas a enumerar las rutas de los archivos. Veamos un ejemplo, en el que tenemos dos archivos que fueron modificados desde el último commit, vamos a hacer un commit con ambos y luego sacaremos uno de los dos:

```
$ git status -s
 M README
 M hello.rb
$ git add .
$ git status -s
M  README
M  hello.rb
$ git reset HEAD -- hello.rb
Unstaged changes after reset:
M hello.rb
$ git status -s
M  README
 M hello.rb
```

Ahora sí, podes correr git commit para grabar los cambios al archivo README, no al archivo hello.rb. En caso de que te preguntes que está haciendo, lo que hace en realidad es resetear las entradas del archivo ese en el “index” para que sea igual que en el commit anterior.

Si queres cambiar el nombre del comando porque te resulta más fácil de entender, sólo tenes que grabarle un alias: git config –global alias.unstage “reset HEAD”.

Una vez que corras eso, podes correr git unstage [file]. Cuando corres git reset sin especificar una opción, por defecto es –mixed. Las otras opciones son –soft y –hard.

**git reset –soft - deshacer el último commit (no saca del los archivos del staging)** Lo primero que git reset hace es deshacer el último commit y poner los archivos de nuevo en el índice. Si se incluye la opción –soft HEAD~ se puede deshacer el commit y los archivos modificados estarán de vuelta en el stage.

```
$ git status -s
M  hello.rb
$ git commit -am 'hello with a flower'
[master 5857ac1] hello with a flower
 1 files changed, 3 insertions(+), 1 deletions(-)
$ git status
# On branch master
nothing to commit (working directory clean)
$ git reset --soft HEAD~
$ git status -s
M  hello.rb
```

Esto es, básicamente, hacer lo mismo que git commit –amend, que te permite hacer más cambios y meterlos en el mismo último commit.

> Nota: Nunca hagas un amend de un commit al que ya le hiciste el push, porque puede traer problemas.

**git reset –hard - deshace el último commit, saca del stage los cambios y pisa los cambios del directorio de trabajo** Esta es la opción más peligrosa, ya que si lo corres por error perdes los cambios que venías realizando en tu directorio de trabajo.

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
# modified:   README
# 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:   README
$ git reset --hard HEAD
```

```
HEAD is now at 5857ac1 hello with a flower
$ git status
# On branch master
nothing to commit (working directory clean)
```

En el ejemplo anterior, teníamos los cambios para meterlos en el stage y luego hacer el commit, pero un git reset –hard los pisó. Además de eso, el último commit se ha deshecho. Se puede reemplazar HEAD con un el número de un commit (SHA-1) u otra referencia para restablecer el directorio de trabajo a ese punto específico.

### git rm - remueve archivos del staging area

Esto es un poco diferente a git reset HEAD que saca los archivos del stage. Sacarlo del stage significa revertir el staging area a la versión anterior a que empecemos a modificar archivos en el directorio de trabajo. En cambio el comando git rm saca la entrada del archivo en el index, de manera que no va a ser tenido más en cuenta para el próximo commit.

Por defecto, git rm [filename] va a remover el archivo del index y del disco, para sacarlo del index pero dejarlo en el directorio de trabajo hay que agregar la opción –cached.

### git stash - guardar cambios realizados en el stage y el index para confirmarlos más tarde

Si estás en medio de algunos cambios y de repente aparece un bug urgente que no podes dejar de atender, pero no podes hacer un commit con algo que no anda y tampoco queres perder los cambios, entonces este comando es la solución.

**git stash – agrega los cambios actuales a la pila** Este comando agarra todos los cambios del directorio de trabajo y del index y los pone en una lista para poder buscarlos posteriormente. Después de correrlo te deja un directorio de trabajo limpio y te devuelve al estado del último commit realizado.

```
$ git status -s
M hello.rb
$ git stash
Saved working directory and index state WIP on master: 5857ac1 hello with a flower
HEAD is now at 5857ac1 hello with a flower
$ git status
# On branch master
nothing to commit (working directory clean)
```

**git stash list – ver los stash que están en la lista** Al correr este comando nos devolverá una lista con los stashes realizados.

```
$ git stash list
stash@{0}: WIP on master: 5857ac1 hello with a flower
```

El último ítem agregado en el stash será referenciado por stash@{0} e incrementará a los que estén en una unidad.

```
$ vim hello.rb
$ git commit -am 'it stops raining'
[master ee2d2c6] it stops raining
1 files changed, 1 insertions(+), 1 deletions(-)
$ vim hello.rb
$ git stash
Saved working directory and index state WIP on master: ee2d2c6 it stops raining
HEAD is now at ee2d2c6 it stops raining
$ git stash list
stash@{0}: WIP on master: ee2d2c6 it stops raining
stash@{1}: WIP on master: 5857ac1 hello with a flower
```

**git stash apply – toma un ítem de la lista de stashes y lo aplica al directorio de trabajo** Cuando estés listo para continuar desde donde dejaste, ejecuta el comando git stash para traer de vuelta los cambios guardados en el directorio de trabajo.

```
$ git stash apply
# On branch master
# 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:   hello.rb
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Si además queres borrar de la lista el ítem que tomaste, debes ejecutar git stash pop.

**git stash drop - remueve un ítem de la lista de stashes** Si corres el comando sin ningún parámetro entonces removerá el último stash, en cambio podes especificar un ítem particular de la lista. En el siguiente ejemplo, se borra el anterior al último:

```
$ git stash drop stash@{1}
Dropped stash@{1} (0b1478540189f30fef9804684673907c65865d8f)
```

Si queres borrar toda la lista entonces debes ejecutar el comando git stash clear.

## Creando y administrando ramas

Resulta útil pensar en los branches como contextos, ya que es la forma en que se usan más a menudo. Cuando realizas cambios de ramas, se cambian los contextos en que se están trabajando y podés rápidamente cambiar de contexto entre las diferentes ramas.

En pocas palabras podés crear una rama / branch con: git branch(nombre de la rama / branch), cambiar a ese contexto con git checkout (nombre de la rama / branch), grabar los cambios en ese contexto, y a continuación alternar a otra rama / branch fácilmente. Cuando cambiás de ramas, Git reemplaza el directorio de trabajo con las versiones de los archivos del último commit en la rama. Se fusionan las ramas con git merge. Podés fusionar (mergear) fácilmente varias veces la misma rama a lo largo del tiempo, o podés optar por eliminar una rama inmediatamente después de su fusión.

### git branch - Listar, crear y administrar los contextos de trabajo

Sin argumentos, git branch listará las ramas locales que tengas. La rama sobre la que se está trabajando actualmente tendrá una estrella al lado de él y si tenés la coloración habilitada, se mostrará la rama actual en verde.

```
$ git branch
* master
```

Esto significa que tenemos una rama 'master' y actualmente estamos en ella. Al ejecutar git init automáticamente creará una rama 'master' de forma predeterminada, sin embargo no hay nada de especial en el nombre, no hace falta tener una rama 'master', pero ya que es el valor por defecto que se crea, la mayoría de los proyectos la tienen.

#### git branch (nombre del branch) - crea una nueva rama

Vamos a empezar por crear una nueva rama y cambiar a ella. Puede hacerlo ejecutando git branch [branchName].

```
$ git branch pruebas
$ git branch
* master
  pruebas
```

Ahora podemos ver que tenemos una nueva rama. Cuando se crea una rama de esta manera se crea a partir del último commit, por lo que si haces algunos cambios y los grabas en este punto y luego cambias a la rama 'pruebas', volverás al contexto de trabajo de cuando se creó la rama en primer lugar, se puede pensar a la rama como un marcador / etiqueta de donde se encuentra actualmente. Vamos a ver esto en acción - usamos git checkout (rama) para cambiar la rama en la que actualmente estamos.

```
$ ls
README hello.rb
$ echo 'contenido de la prueba'> test.txt
$ echo "más contenido"> more.txt
$ git add *. Txt
$ git commit-m 'added two files'
[master 8bd6d8b] 'added two files
 2 files changed, 2 insertions (+), 0 deletions(-)
create mode 100.644 more.txt
create mode 100.644 test.txt
$ ls
README hello.rb more.txt test.txt
$ git checkout pruebas
Switched to branch 'pruebas'
$ ls
README hello.rb
```

Así que ahora podemos ver que cuando nos cambiamos a la rama 'testing', se eliminaron los archivos nuevos. Podríamos volver a la rama 'master' y verlos reaparecer.

```
$ ls
README hello.rb
$ git checkout master
Switched to branch 'master'
$ ls
README hello.rb more.txt test.txt
```

### git branch -v - ver la última confirmación en cada rama

Si queremos ver los últimos commits en cada rama podemos ejecutar git branch -v para verlos.

```
$ git branch -v
* master      54b417d fix javascript issue
 development 74c111d modify component.json file
 testing     62a557a update test scripts
```

### git branch -d [branchName] - eliminar una rama

Si queremos eliminar una rama (como la rama 'pruebas' en el ejemplo anterior, ya que no hay un trabajo único en ella), podemos ejecutar git branch -d (branch) para eliminarlo.

```
$ git branch
* master
 testing
$ git branch -d testing
Deleted branch testing (was 78b2670).
$ git branch
* master
```

## git checkout -b (nombre del branch) - crear y cambiar inmediatamente a una rama

En la mayoría de los casos vas a querer cambiar de rama de inmediato, por lo que podés hacer el trabajo en ella y luego la fusión en una rama que contiene solamente un trabajo estable (como el "master") en un momento posterior, cuando el trabajo en su nueva rama es estable. Podés hacer esto con bastante facilidad con git branch newbranch, git checkout newbranch, pero Git te da un acceso directo para esto: git checkout-b newbranch.

```
$ git branch
* master
$ ls
README   hello.rb more.txt test.txt
$ git checkout -b removals
Switched to a new branch 'removals'
$ git rm more.txt
rm 'more.txt'
$ git rm test.txt
rm 'test.txt'
$ ls
README   hello.rb
$ git commit -am 'removed useless files'
[removals 8f7c949] removed useless files
```

```
2 files changed, 0 insertions(+), 2 deletions(-)
delete mode 100644 more.txt
delete mode 100644 test.txt
$ git checkout master
Switched to branch 'master'
$ ls
README   hello.rb more.txt test.txt
```

Acá podés ver como creamos una rama, eliminado algunos de los archivos, mientras que estábamos en el contexto de esa rama, y luego cambiamos a nuestra rama principal y vemos que algunos archivos volvieron a aparecer. La ramificación aísla con seguridad el trabajo que hacemos en contextos entre los que podemos cambiar. Si comenzás a trabajar, es muy útil empezar siempre en una rama (porque es rápido y fácil de hacer) y luego fusionarlo y eliminar la rama cuando haya terminado. De esta forma, si lo que estás haciendo no se resuelve fácilmente podés descartarlo y si tenés que cambiar de nuevo a un contexto más estable, el trabajo en curso es fácil de poner a un lado y luego volver.

> Nota: Cuando creas una rama hay que agregarla al archivo ".git/config" y configurar contra que hace el merge, para que una vez que haces el push y luego el pull sepa contra que mergea lo que trae de la rama creada en el repositorio remoto. Por ejemplo, si creaste la rama dev, entonces tendrás que agregar esto a tu config: [branch "dev"] remote = origin merge = refs/heads/dev

## git push (nombre-remoto): (nombre de branch) - eliminar una rama remota

Cuando hayas terminado de trabajar con una rama remota, ya sea porque se fusionó con el master remoto o si deseas abandonarla y eliminarla, se emite un comando git push con un símbolo de dos puntos para eliminar esa rama.

```
$ git push origin :tidy-cutlery
To git@github.com:octocat/Spoon-Knife.git
- [deleted]         tidy-cutlery
```

En el ejemplo anterior se ha borrado la rama "tidy-cutlery" del branch remoto "origin". Una manera de recordar esto es pensar en la sintaáxis git push remote-name local-branch: remote-branch. Esto indica que desea hacer push de tu rama local para que coincida con la de la rama remota. Cuando se quita la parte local-branch se intenta hacer coincidir nada con la rama remota, "diciéndole" la rama remota que se convierta en "nada". Como alternativa, podés ejecutar git push remote-name –delete En pocas palabras se usa git branch para listar sus ramas actuales, crear nuevas ramas y eliminar ramas innecesarias o ya fusionadas.

## git merge - fusionar una rama en tu contexto actual

Una vez que tengas el trabajo aislado en una rama, es muy probable que desees incorporarlo en la rama principal. Podés combinar cualquiera de las ramas en su rama actual con el comando git merge. Tomemos como ejemplo la rama "removals" de arriba. Si creamos una rama, eliminamos archivos en ella y commiteamos los cambios de 'removals' en esa rama, se aísla de nuestra rama principal ("master", en este caso). Para incluir las eliminaciones en la rama 'master', podés fusionar en ella la rama 'removals'.

```
$ git branch
* master
 removals
$ ls
README   hello.rb more.txt test.txt
$ git merge removals
Updating 8bd6d8b..8f7c949
Fast-forward
more.txt |    1 -
test.txt |    1 -
2 files changed, 0 insertions(+), 2 deletions(-)
delete mode 100644 more.txt
delete mode 100644 test.txt
$ ls
README   hello.rb
```

Fusiones mas complejas: Por supuesto, esto no sólo funciona para simples adiciones y supresiones de archivos; Git fusionará las modificaciones de archivos. Por ejemplo, vamos a ver lo que sucede cuando editamos un archivo en una rama, en otra rama lo renombramos y lo editamos; y luego fusionar ambas ramas.

```
$ git branch
* master
$ cat hello.rb
class HelloWorld
 def self.hello
   puts "Hello World"
 end
end
HelloWorld.hello
```

Así que primero vamos a crear una nueva rama llamada "change_class" y cambiar a ella por lo que sus cambios de nombre van a quedar aislados. Vamos a cambiar cada instancia de Hello World' a 'HiWorld'.

```
$ git checkout -b change_class
Switched to a new branch 'change_class'
$ vim hello.rb
$ head -1 hello.rb
class HiWorld
$ git commit -am 'changed the class name'
[change_class 3467b0a] changed the class name
1 files changed, 2 insertions(+), 4 deletions(-)
```

Así que ahora que hemos commiteado los cambios de la rama 'change_class'; para volver a la rama 'master' el nombre de clase volverá a lo que era antes de que cambiemos ramas. Aquí podemos cambiar algo diferente (en este caso la salida impresa) y al mismo tiempo cambiar el nombre del archivo desde hello.rb a ruby.rb.

```
$ git checkout master
Switched to branch 'master'
$ git mv hello.rb ruby.rb
$ vim ruby.rb
$ git diff
diff --git a/ruby.rb b/ruby.rb
index 2aabb6e..bf64b17 100644
--- a/ruby.rb
+++ b/ruby.rb
@@ -1,7 +1,7 @@
class HelloWorld

  def self.hello
-    puts "Hello World"
+    puts "Hello World from Ruby"
  end
end
$ git commit -am 'added from ruby'
[master b7ae93b] added from ruby
1 files changed, 1 insertions(+), 1 deletions(-)
rename hello.rb => ruby.rb (65%)
```

Ahora, esos cambios se guardan en la rama 'master'. Observe que el nombre de la clase es de nuevo a 'Hello World', no 'HiWorld'. Para incorporar el cambio "HiWorld' podemos hacer el merge hacia la rama' change_class '. Sin embargo, el nombre del archivo ha cambiado desde que creamos las ramas, que hará Git?

```
$ git branch
 change_class
* master
```

```
$ git merge change_class
Renaming hello.rb => ruby.rb
Auto-merging ruby.rb
Merge made by recursive.
ruby.rb |    6 ++----
1 files changed, 2 insertions(+), 4 deletions(-)
$ cat ruby.rb
class HiWorld
 def self.hello
   puts "Hello World from Ruby"
 end
end

HiWorld.hello
```

Observa que no hay conflictos de merge y el archivo que se había renombrado, ahora tiene el nombre de clase 'HiWorld' que se hizo en la otra rama.

Conflictos de merge / fusión: En situaciones en las que el mismo bloque de código se edita en diferentes ramas no hay manera de que una computadora lo resuelva, así que depende de nosotros. Vamos a ver otro ejemplo donde se cambia la misma línea en dos ramas.

```
$ git branch
* master
$ git checkout -b fix_readme
Switched to a new branch 'fix_readme'
$ vim README
$ git commit -am 'fixed readme title'
[fix_readme 3ac015d] fixed readme title
1 files changed, 1 insertions(+), 1 deletions(-)
```

Ahora hemos commiteado un cambio a una línea en nuestro archivo README en una rama. Vamos a cambiar la misma línea de una manera diferente en nuestra rama 'master'.

```
$ git checkout master
Switched to branch 'master'
$ vim README
$ git commit -am 'fixed readme title differently'
[master 3cbb6aa] fixed readme title differently
1 files changed, 1 insertions(+), 1 deletions(-)
```

Vamos a combinar la primera rama en nuestra rama principal, provocando un conflicto de merge.

```
$ git merge fix_readme
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
$ cat README
<<<<<<< HEAD
Many Hello World Examples
=======
Hello World Lang Examples
>>>>>>> fix_readme
This project has examples of hello world in
```

Podemos ver que Git inserta marcas de conflicto estándares, en los archivos cuando hay un conflicto de merge. Ahora nos toca a nosotros resolverlos. Vamos a hacerlo de forma manual, pero mirá opciones para git mergetool si queres que Git use una herramienta de merging gráfica (como kdiff3, emerja, p4merge, etc).

```
$ vim README  # here I'm fixing the conflict
$ git diff
diff --cc README
index 9103e27,69cad1a..0000000
--- a/README
+++ b/README
@@@ -1,4 -1,4 +1,4 @@@
- Many Hello World Examples
-Hello World Lang Examples
++Many Hello World Lang Examples
 This project has examples of hello world in
```

Un buen consejo para la resolución de conflictos de merge en Git es que si ejecutas git diff, te mostrará los dos lados del conflicto y cómo se ha resuelto (como se muestra aquí). Ahora es el momento de marcar el archivo como resuelto. En Git lo hacemos con git add - para decirle a Git que el archivo se ha resuelto tenés que hacerle un commit.

```
$ git status -s
UU README
$ git add README
$ git status -s
M  README
$ git commit
[master 8d585ea] Merge branch 'fix_readme'
```

Y ahora ya resolvimos con éxito nuestro conflicto de merge y commiteado el resultado. En pocas palabras, se usa git merge para combinar otra rama en tu

rama actual. Puede ser que se de cuenta de forma automática cómo combinar de la mejor manera los diferentes cambios de ambas ramas y si no puede sólo, tendrás que resolver los conflictos manualmente.

## git log - mostrar la historia de commits en un branch

Hasta ahora hemos estado haciendo commits de cambios de tu proyecto y cambiando entre diferentes ramas aisladas, pero ¿Qué pasa si nos hemos olvidado de cómo llegamos a donde estamos? ¿Y si queremos saber en qué difiere una rama de otra? Git te proveé de una herramienta que te muestra todos los mensajes de commit que te llevan al estado actual de la rama, este comando se llama git log. Para entender el comando git log, hay que entender que información se almacena cuando se ejecuta el comando git commit para guardar los cambios realizados. Además de mostrar los archivos modificados y los mensajes de commit, también se muestra información sobre la persona que hizo el commit, Git también almacena el commit en que se ha basado el estado actual de la rama. Es decir, si se clona un proyecto, ¿Cuál es el commit que modificaste para llegar al commit que subiste / pusheaste? Esto es útil para saber cómo el proyecto llegó a donde está y permite a Git averiguar quién lo cambió. Si Git tiene commit que se guardó y en el que se basó, entonces se puede calcular de forma automática lo que ha cambiado. El commit en que se basó un nuevo commit se llama el "padre / parent". Para ver una lista cronológica de los padres de cualquier rama, puede ejecutar git log cuando estás en esa rama. Por ejemplo, si ejecutamos git log en el proyecto Hello World en que hemos estado trabajando en esta sección, vamos a ver todos los commits que hemos hecho. Para ver una versión más compacta de la misma historia, podemos utilizar la opción - oneline.

```
$ git log --oneline
8d585ea Merge branch 'fix_readme'
3cbb6aa fixed readme title differently
3ac015d fixed readme title
558151a Merge branch 'change_class'
b7ae93b added from ruby
3467b0a changed the class name
17f4acf first commit
```

Lo que nos muestra es la historia del desarrollo de este proyecto. Si los mensajes de los commits son adecuadamente descriptivos, ésto nos puede informar sobre todos los cambios que se han aplicado o han influido en el estado actual de la rama y por lo tanto lo que hay en ella. También podemos utilizarlo para ver cuando la historia se bifurca y se fusionó / mergeó con la opción –graph. Aquí es el mismo comando pero con el gráfico de topología encendido:

```
$ git log --oneline --graph
*   8d585ea Merge branch 'fix_readme'
```

```
|\
| * 3ac015d fixed readme title
* | 3cbb6aa fixed readme title differently
|/
*   558151a Merge branch 'change_class'
|\
| * 3467b0a changed the class name
* | b7ae93b added from ruby
|/
* 17f4acf first commit
```

Ahora podemos ver más claramente cuando las ramas se separaron y luego cuando se fusionaron de nuevo. Esto es muy bueno para ver lo que ha sucedido o qué cambios se aplicaron; también es muy útil para la gestión de las ramas. Vamos a crear una nueva rama, hacer algún trabajo en ella y luego volver a nuestra rama principal para hacer algo de trabajo; finalmente vamos a ver cómo el comando log puede ayudarnos a entender lo que está sucediendo en cada uno. En primer lugar vamos a crear una nueva rama para añadir el lenguaje de programació Erlang al ejemplo Hello World, queremos hacer esto en una rama para no "ensuciar" nuestra rama estable con código que puede no funcionar por un tiempo, para que podamos cambiar "limpiamente" hacia y desde ella.

```
$ git checkout -b erlang
Switched to a new branch 'erlang'
$ vim erlang_hw.erl
$ git add erlang_hw.erl
$ git commit -m 'added erlang'
[erlang ab5ab4c] added erlang
1 files changed, 5 insertions(+), 0 deletions(-)
create mode 100644 erlang_hw.erl
```

También agregamos un programa de ejemplo Haskell mientras que todavía en la rama llamada "erlang".

```
$ vim haskell.hs
$ git add haskell.hs
$ git commit -m 'added haskell'
[erlang 1834130] added haskell
1 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 haskell.hs
```

Finalmente, decidimos que queremos cambiar el nombre de clase de nuestro programa de Rubi a como era antes. Por lo tanto, podemos volver a la rama principal, cambiar eso y commitear directamente en la rama principal en lugar de crear otra rama.

```
$ git checkout master
Switched to branch 'master'
$ ls
README  ruby.rb
$ vim ruby.rb
$ git commit -am 'reverted to old class name'
[master 594f90b] reverted to old class name
1 files changed, 2 insertions(+), 2 deletions(-)
```

Así que ahora supongamos que no trabajamos en el proyecto durante un tiempo porque tenemos otras cosas que hacer. Cuando regresemos queremos saber en qué estado está la rama 'erlang' y que hemos dejado fuera de la rama principal. Con sólo mirar el nombre de la rama, no podemos saber qué cambios hemos hecho en Haskell, pero usando git log podemos saberlo fácilmente. Si le das un nombre de rama a Git, te mostrará sólo los commits que son "accesibles" en la historia de esa rama, que son los commits que influyeron en el estado actual.

```
$ git log --oneline erlang
1834130 added haskell
ab5ab4c added erlang
8d585ea Merge branch 'fix_readme'
3cbb6aa fixed readme title differently
3ac015d fixed readme title
558151a Merge branch 'change_class'
b7ae93b added from ruby
3467b0a changed the class name
17f4acf first commit
```

De esta manera, es muy fácil ver que código tenemos incluído en la rama Haskell (resaltado en la salida). Lo que es aún más genial es que podemos decirle fácilmente a Git que sólo estamos interesados en los commits que son "accesibles" desde una rama y no desde otra, es decir, qué commits son exclusivos de una rama en comparación con otra.

En este caso, si estamos interesados en mergear la rama "erlang", queremos ver qué commits van a afectar nuestra rama cuando hagamos el correspondiente merge. La forma en que le decimos a Git es agregando un ^ delante de la rama que no queremos ver. Por ejemplo, si queremos ver los commits que se hay en la rama 'erlang' y que no están en la rama 'master', podemos hacer erlang ^ master, o viceversa. Tenga en cuenta que en línea de comandos, Windows trata a ^ como un carácter especial, en cuyo caso tendrá que rodear ^ master entre comillas.

```
$ git log --oneline erlang ^master
1834130 added haskell
```

```
ab5ab4c added erlang
$ git log --oneline master ^erlang
594f90b reverted to old class name
```

Esto nos da una buena y simple herramienta de gestión de ramas. Nos permite ver fácilmente que commits son únicos y a que ramas pertenecen, así sabemos lo que nos falta y lo que íbamos a mergear si tuviéramos que hacer el merge correspondiente. En pocas palabras se usa git log para hacer una lista de la historia de commits o una lista de personas que han hecho cambios que han llevado al estado actual de la rama. Esto permite ver cómo el proyecto llegó al estado en que se encuentra actualmente.

## git tag - marcar un punto en la historia como importante

Si se llega a un punto que es importante y que se quiere recordar por siempre ese commit en específico, podés etiquetarlo con git tag. El comando tag, básicamente, consiste en poner un marcador permanente en un commit en concreto para que pueda ser usado para comparar con otro commit en el futuro. Esto se hace a menudo cuando se libera una release (versión) o se hace una entrega.

Digamos que queremos lanzar nuestro proyecto Hello World como la versión "1.0". Podemos marcar el último commit (HEAD) como "v1.0" ejecutando git tag -a v1.0. La opción -a significa "hacer una etiqueta anotada", lo que te permite agregar un mensaje a la etiqueta, que es lo que casi siempre se quiere hacer. La ejecución sin -a también funciona, pero no graba cuando fue etiquetado, quien lo ha marcado, o no te permite agregar un mensaje a la etiqueta. Se recomienda crear siempre las etiquetas anotadas.

```
$ git tag -a v1.0
```

Cuando se ejecuta el comando git tag -a, Git abrirá el editor y se puede escribir un mensaje para la etiqueta, al igual que como se escribe un mensaje de commit. Ahora, cuando corremos git log –decorate, podemos ver nuestra etiqueta allí.

```
$ git log --oneline --decorate --graph
* 594f90b (HEAD, tag: v1.0, master) reverted to old class name
*   8d585ea Merge branch 'fix_readme'
|\
| * 3ac015d (fix_readme) fixed readme title
* | 3cbb6aa fixed readme title differently
|/
*   558151a Merge branch 'change_class'
|\
| * 3467b0a changed the class name
```

```
* | b7ae93b added from ruby
|/
* 17f4acf first commit
```

Si hacemos más commits, la etiqueta estará justo en ese commit, así que tenemos un commit específico marcado para siempre y siempre se pueden comparar futuros commits al mismo.

Nosotros no tenemos que etiquetar el commit en el que estamos, sin embargo, si nos olvidamos de etiquetar un commit que hemos pusheado / subido, podemos etiquetarlo ejecutando el mismo comando, pero con el SHA commit al final. Por ejemplo, digamos que habíamos lanzado el commit 558151a (varios commits atrás), pero se nos olvidó etiquetarlos en el momento. Los podemos marcar ahora:

```
$ git tag -a v0.9 558151a
$ git log --oneline --decorate --graph
* 594f90b (HEAD, tag: v1.0, master) reverted to old class name
*   8d585ea Merge branch 'fix_readme'
|\
| * 3ac015d (fix_readme) fixed readme title
* | 3cbb6aa fixed readme title differently
|/
*   558151a (tag: v0.9) Merge branch 'change_class'
|\
| * 3467b0a changed the class name
* | b7ae93b added from ruby
|/
* 17f4acf first commit
```

Las etiquetas apuntando a los objetos trackeados desde las cabezas de las ramas se descargarán automáticamente al hacer un fetch de un repositorio remoto. Sin embargo, las etiquetas que no son accesibles desde las cabezas de las ramas se omitirán. Si desea asegurarse de que todas las etiquetas se incluyen siempre, debe incluir la opción –tags.

```
$ git fetch origin --tags
remote: Counting objects: 1832, done.
remote: Compressing objects: 100% (726/726), done.
remote: Total 1519 (delta 1000), reused 1202 (delta 764)
Receiving objects: 100% (1519/1519), 1.30 MiB | 1.21 MiB/s, done.
Resolving deltas: 100% (1000/1000), completed with 182 local objects.
From git://github.com:example-user/example-repo
* [new tag]         v1.0       -> v1.0
* [new tag]         v1.1       -> v1.1
```

Si lo que desea es una sola etiqueta, usar git fetch tag . Por defecto, las etiquetas no están incluidas cuando haces un push a un repositorio remoto. Con el fin de actualizar explícitamente esto, se debe incluir la opción –tags usando git push. En pocas palabras se utiliza git tag para marcar un commit o marcar tu reporitorio como importante. Esto también te permite referirte a ese commit con una referencia más fácil de recordar que un SHA.

## Intercambio y actualización de Proyectos

Git no tiene un servidor central como Subversion. Todos los comandos hasta el momento se han realizado a nivel local, sólo la actualización de una base de datos local. Para poder colaborar con otros desarrolladores en Git, tenes que poner todos los datos en un servidor en el que los otros desarrolladores tengan acceso. La forma en que Git hace esto es sincronizando los datos con otro repositorio remoto. En estos casos no hay ninguna diferencia real entre un servidor y un cliente - un repositorio Git es un repositorio Git y se puede sincronizar con otro fácilmente.

En general, vas a hacer una serie de commits a nivel local, y luego a buscar a los datos del repositorio compartido en línea que clonaste, combinar cualquier nuevo trabajo que se agregue al tuyo, y por último hacer un push de todo esto al repositorio remoto.

### git pull bajar nuevas ramas y datos del repositorio remoto

El comando git pull busca en un repositorio remoto y trata de combinar en la rama actual lo que trajo del remoto. Git tiene dos comandos para actualizar el repositorio local con un repositorio remoto. El primer comando es git fetch y se usa para sincronizar el repositorio local con otro remoto, bajando los datos no tenes a nivel local. Las ramas que se encuentran en el servidor remoto se llaman “ramas remotas” y son idénticas a las ramas locales, con excepción de que Git no te permitirá que cambies tu entorno hacia ellas - sin embargo, podes combinarlas con las tuyas locales (hacer un merge de esas ramas a las tuyas), hacerles un diff con otras ramas, etc. Pero todos esos comandos los corres locales después de sincronizar el repositorio en el que estás trabajando.

El segundo comando que busca los nuevos datos de un servidor remoto es “git pull”. Este comando básicamente ejecuta un git fetch inmediatamente seguido de un git merge de la rama que esta en ese repositorio remoto y que está siendo seguida por la rama en la que te encontras parado en tu repositorio local. Algunos autores insisten en correr los comandos fetch y merge por separado, sin embargo la mayoría de la gente que utiliza git hace directamente un git pull cada vez que quiere actualizar su repositorio local con el repositorio remoto.

Asumiendo que tenes un repositorio remoto configurado y queres bajarte actualizaciones, tendrías que correr el comando git fetch [alias] para avisarle a Git que baje toda las actualizaciones, y luego deberás correr git merge [alias]/[branch] para hacer un merge (sincronicar los cambios) en la rama en la que te encontras parado, cualquier actualización que veas en el servidor. Entonces, si por ejemplo, estuviste trabajando en un proyecto de Hello World en conjunto con muchas otras personas y queres traerte todos los cambios que se subieron desde que te actualizaste por última vez, tendrías que hacer algo similar al siguiente ejemplo, en que tomamos a github como el alias (un alias es una forma amigable de llamar a una URL de un repositorio git):

```
$ git fetch github
remote: Counting objects: 4006, done.
remote: Compressing objects: 100% (1322/1322), done.
remote: Total 2783 (delta 1526), reused 2587 (delta 1387)
Receiving objects: 100% (2783/2783), 1.23 MiB | 10 KiB/s, done.
Resolving deltas: 100% (1526/1526), completed with 387 local objects.
From github.com:schacon/hw
   8e29b09..c7c5a10  master     -> github/master
   0709fdc..d4ccf73  c-langs    -> github/c-langs
   6684f82..ae06d2b  java       -> github/java
 * [new branch]      ada        -> github/ada
* [new branch]      lisp       -> github/lisp
```

Acá podemos ver que desde la última vez que actualizamos nuestro repositorio se agregaron o actualizaron cinco ramas. Las ramas 'ada' y 'lisp' son nuevas, y las ramas 'master', 'c-langs' y 'java' fueron actualizadas. En nuestro ejemplo, otros desarrolladores están subiendo cambios a ramas remotas para ser controladas antes de ser mergeadas a la rama master. También podes ver los mapeos que hace git, la rama 'master' en el repositorio remoto, se ve como una rama llamada 'github/master' en el local. De esta forma podes hacer un merge de la rama 'master' remota en la rama 'master' local, corriendo el comando git merge github/master. O podes ver que commits nuevos hay en esa rama corriendo git log github/master ^master. Si tu repositorio remoto se llama 'origin', entonces será origin/master en tal caso. Si tenes más de un repositorio remoto, podes hacer un fecth desde uno específico corriendo git fetch [alias] o podes decirle a git que sincronice con todos los remotos corriendo git fetch –all.

## git push subir las nuevas ramas y cambios a un repositorio remoto

Para compartir tus commits con otros, necesitas hacer un push de tus cambios a un repositorio remoto. Para hacer esto, tenes que correr el comando git push [alias] [branch] que intentará hacer tu [branch] el nuevo [branch] en el [alias]

remoto. Veamos un ejemplo tratando de hacer un push inicial (por primera vez) de nuestra rama 'master' al alias remoto 'github'.

```
$ git push github master
Counting objects: 25, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (25/25), done.
Writing objects: 100% (25/25), 2.43 KiB, done.
Total 25 (delta 4), reused 0 (delta 0)
To git@github.com:schacon/hw.git
* [new branch]      master -> master
```

Ahora si alguien clona el repositorio, entonces tendrá exactamente lo mismo que acabamos de subir junto con toda la historia. Ahora, ¿Que pasa si tenes un branch como el llamado 'erlang' del ejemplo anterior y queres compartir sólo ese? Podes hacer un push únicamente de ese branch:

```
$ git push github erlang
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 652 bytes, done.
Total 6 (delta 1), reused 0 (delta 0)
To git@github.com:schacon/hw.git
 * [new branch]      erlang -> erlang
```

Ahora, cuando la gente haga un clon o un fetch sobre ese repositorio, entonces obtendrán un branch llamado 'erlang' que podrán ver y del cual podrán sincronizar. Podes hacer un push de cualquier branch a cualquier repositorio remoto al que tengas permisos de escritura. Si tu branch ya se encuentra en el repositorio remoto, git va a tratar de actualizarlo, si no se encuentra, entonces lo va a crear. Por último, un problema frecuente se da cuando queres hacer un push de algún cambio sobre una rama que alguien actualizó antes que vos y vos no tenes esa actualización. Si tratas de hacer un push en ese caso Git por default no te va a dejar pisar esas actualizaciones. Para resolver este problema, tenes que actualizar tu copia haciendo un pull de los cambios (o un fetch-merge) y luego hacer el push. Esto es lo que pasará si intentas de hacer un push en una rama que tenes desactualizada con respecto al servidor remoto:

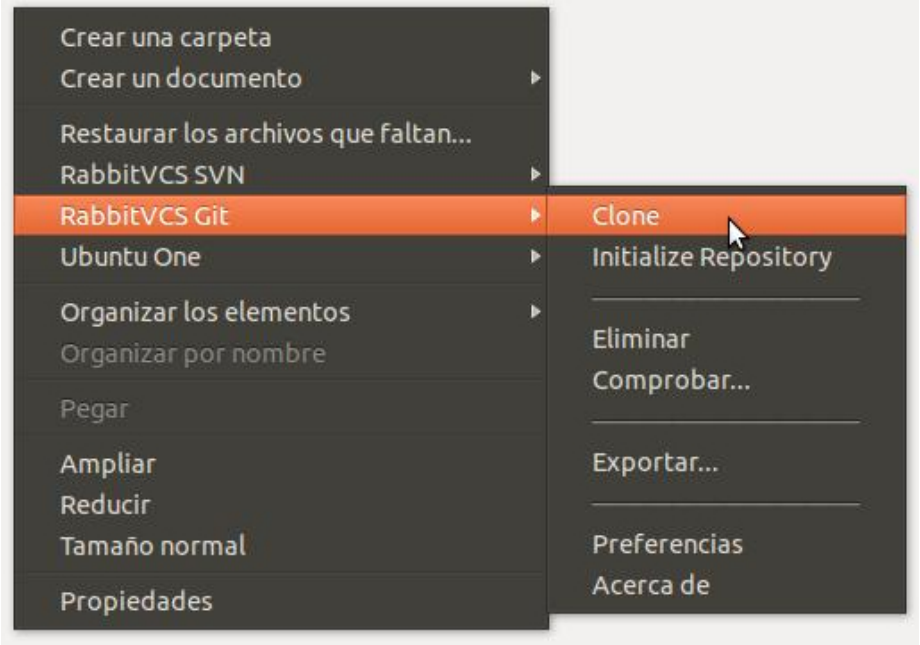
```
$ git push github master
To git@github.com:schacon/hw.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'git@github.com:schacon/hw.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes before pushing again.  See the 'Note about
fast-forwards' section of 'git push --help' for details.
```

Entonces podes arreglar esto corriendo git fetch github; git merge github/master y luego haciendo nuevamente el push.

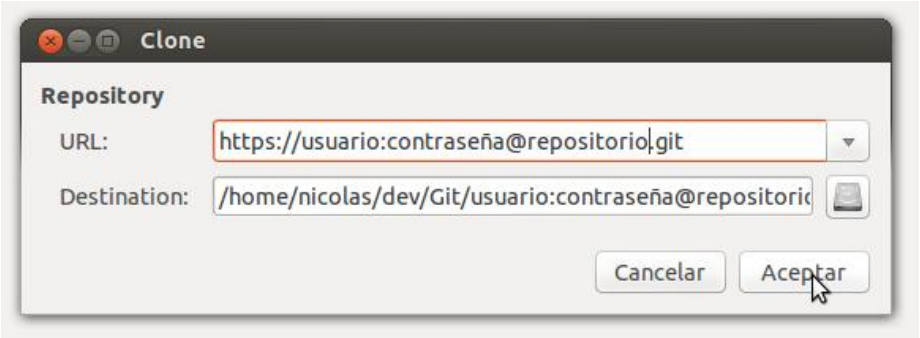
## Herramienta visual: Rabbit VCS

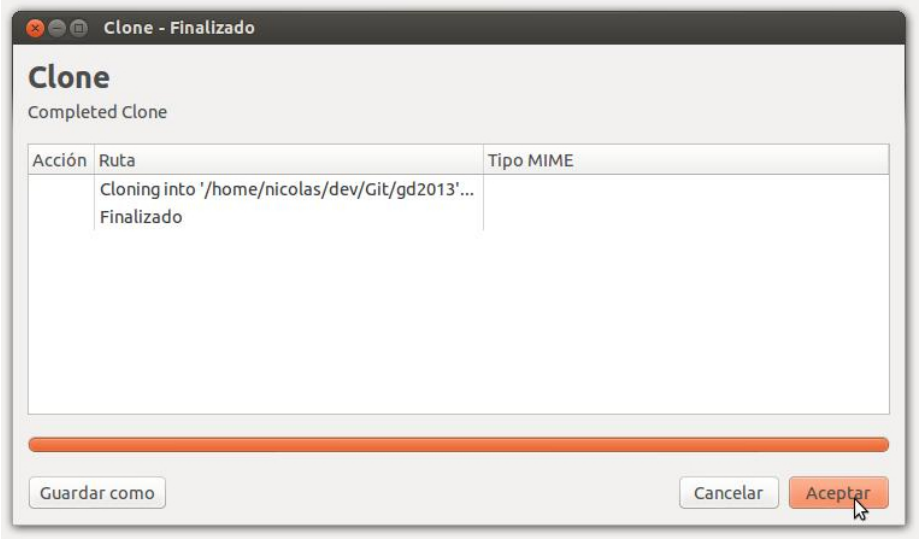
Si bien recomendamos particularmente la línea de comandos para trabajar con git, también existen herramientas visuales que nos permiten trabajar de una forma más "amigable". Una herramienta muy conocida y usada es RabbitVCS, a continuación mostramos paso a paso como sería uno de los flujos explicados anteriormente con una herramienta visual como esta:

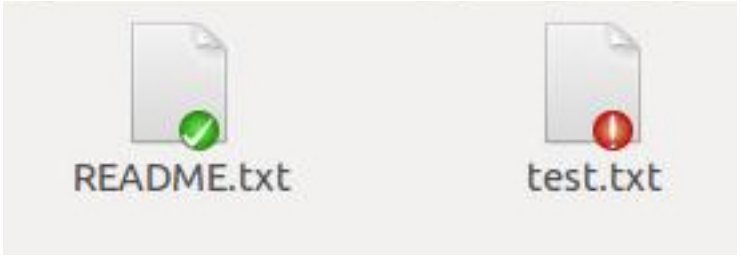
- Clonamos un repositorio: Especificamos la URL, el usuario y la contraseña para que Git verifique los permisos.

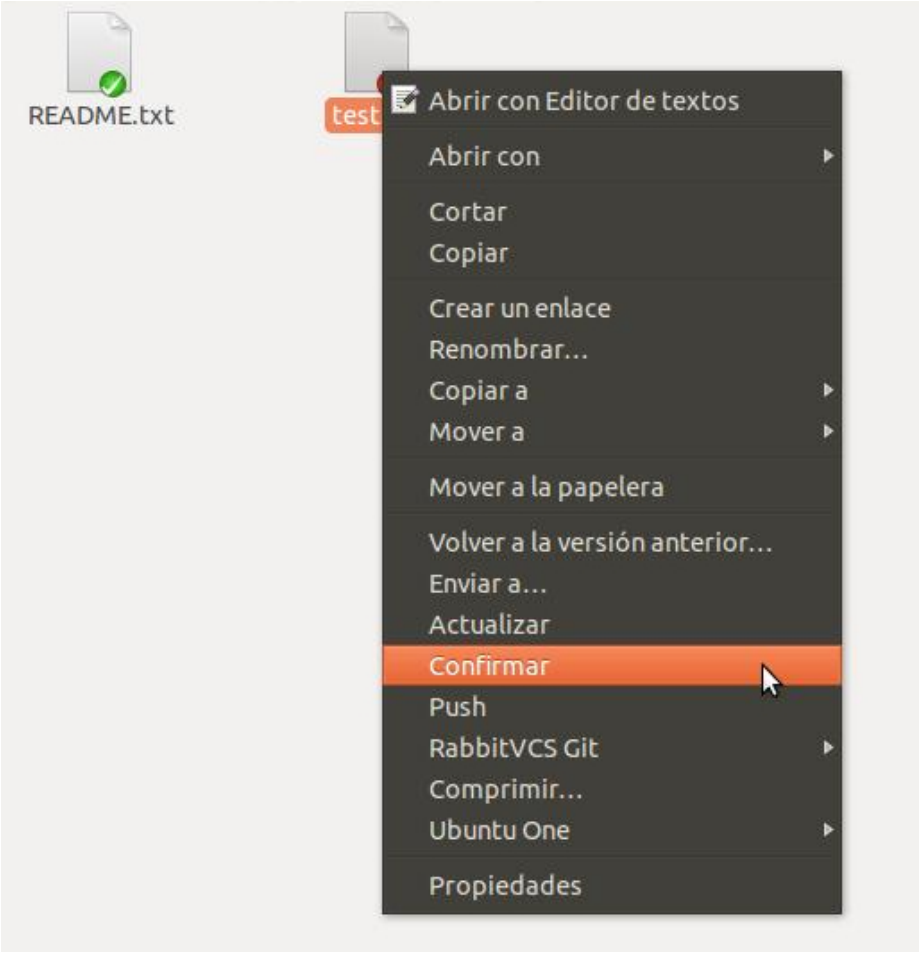


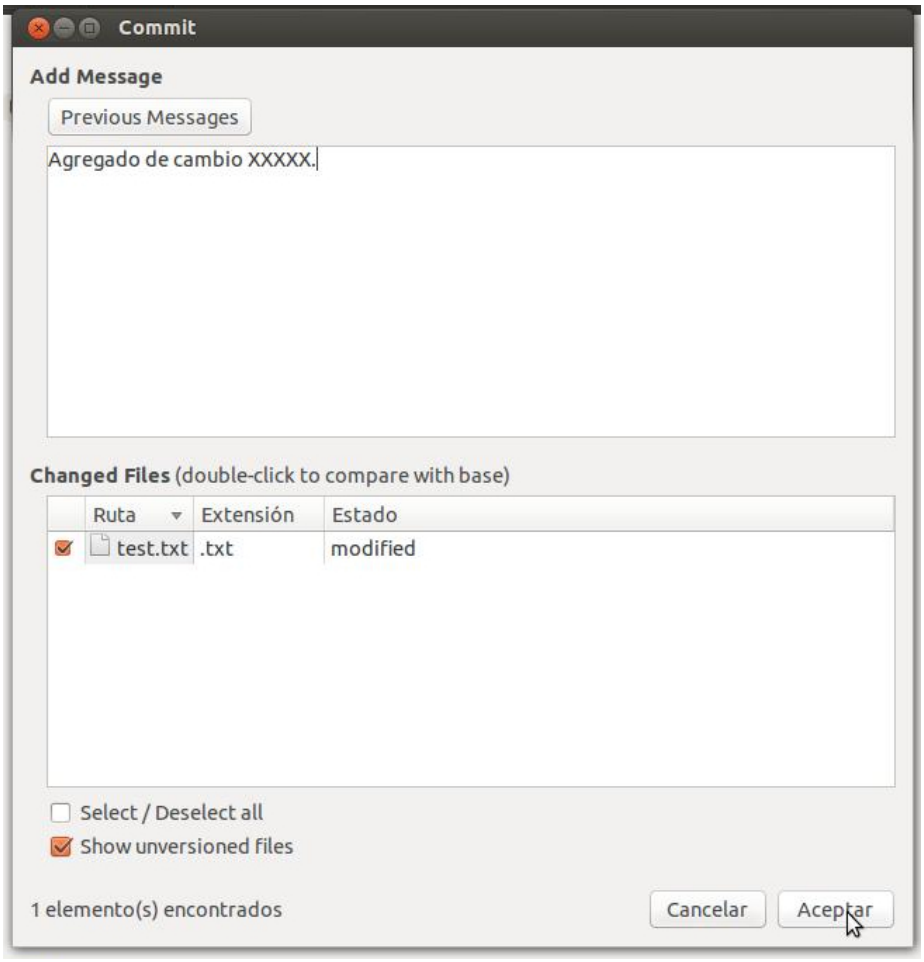

- Realizamos un cambio en un archivo: Podemos ver que la herramienta nos marca los archivos modificados con un icono color rojo.
- Hacemos un commit del cambio realizado: elegimos la opción "confirmar" en caso de que tengamos lenguaje español.
- Confirmamos el commit: Le agregamos un mensaje que describa lo que estuvimos haciendo y confirmamos.

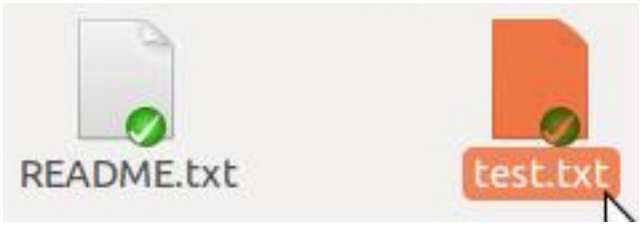

Clone
Repository
URL:
https://usuario:contraseña@repositorio.git
Destination:
/home/nicolas/dev/Git/usuario:contraseña@repositorio
Cancelar
Aceptar

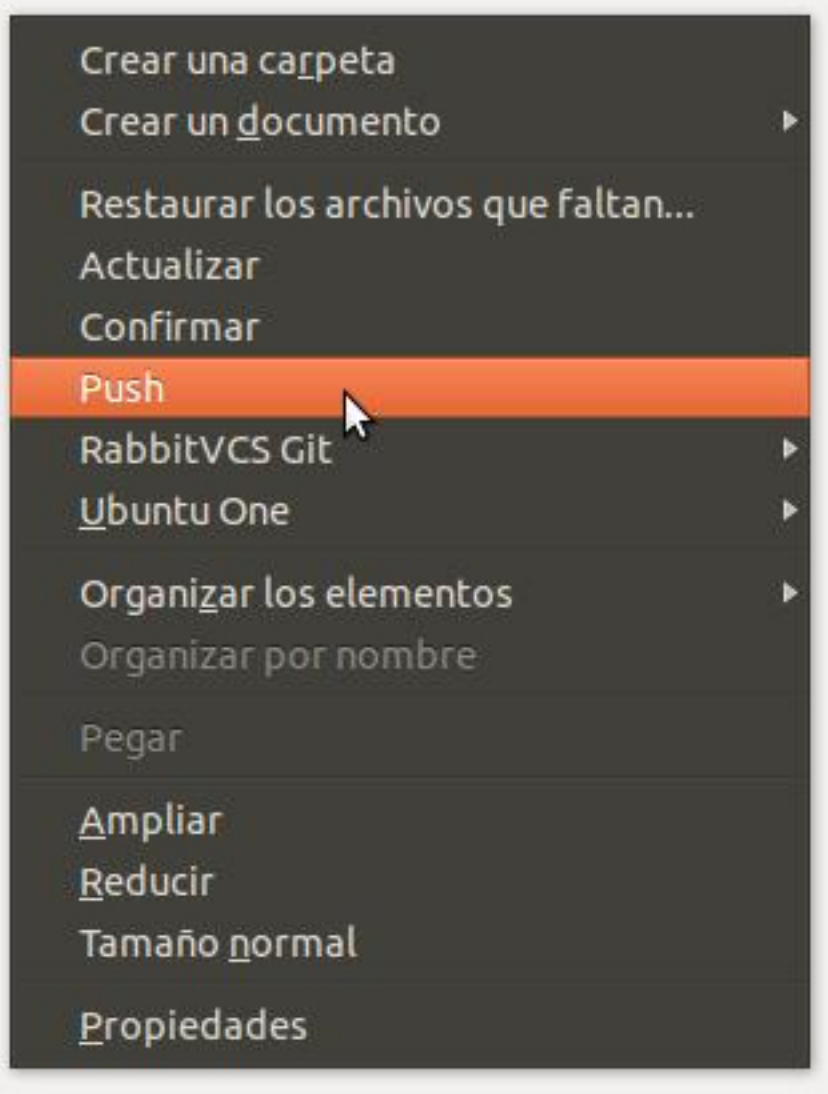

Clone - Finalizado
Clone
Completed Clone
Acción
Ruta
Tipo MIME
Cloning into '/home/nicolas/dev/Git/gd2013'...
Finalizado
Guardar como
Cancelar
Aceptar


README.txt
test.txt


README.txt
test
Abrir con Editor de textos
Abrir con
Cortar
Copiar
Crear un enlace
Renombrar…
Copiar a
Mover a
Mover a la papelera
Volver a la versión anterior…
Enviar a…
Actualizar
Confirmar
Push
RabbitVCS Git
Comprimir…
Ubuntu One
Propiedades


Commit
Add Message
Previous Messages
Agregado de cambio XXXXX.
Changed Files (double-click to compare with base)
Ruta
Extensión
Estado
test.txt
.txt
modified
Select / Deselect all
Show unversioned files
1 elemento(s) encontrados
Cancelar
Aceptar


README.txt
test.txt


Crear una carpeta
Crear un documento
Restaurar los archivos que faltan...
Actualizar
Confirmar
Push
RabbitVCS Git
Ubuntu One
Organizar los elementos
Organizar por nombre
Pegar
Ampliar
Reducir
Tamaño normal
Propiedades

- Hacemos un push: Se subirá al servidor remoto todo lo commiteado en el repositorio.

Para más información y para bajarse la herramienta visitar el sitio oficial de la herramienta: http://rabbitvcs.org/

## Referencias

- Se utilizaron algunas imágenes cuya fuente provienen del sitio git-scm.com.
- La información troncal del documento se redactó basándose en el sitio oficial de gitref.
- El contenido está bajo la licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional.