

# Tutorial de Git

Fuente original: [http://hoth.entp.com/output/git\\_for\\_designers.html](http://hoth.entp.com/output/git_for_designers.html)

Traducido al español por **Rodolinux**: <http://www.rodolinux.com.ar>

Formato del texto e imágenes agregadas por [Pablo Luis Martinez](#) – Sgo del Estero

El control de versiones, también conocido como control de fuentes o control de revisiones es parte integral de cualquier flujo de trabajo (workflow) de desarrollo. ¿Porqué? Es esencialmente una herramienta de comunicación, como los emails o IM, pero trabaja con código en vez de conversaciones humanas.

Control de versiones:

- permite a los programadores comunicar fácilmente su trabajo a otros.
- le permite a un equipo compartir el código.
- mantener versiones separadas de “producción” que están siempre deployables.
- permite el desarrollo simultáneo de diferentes características en el mismo código base.
- mantiene la pista de todas las versiones viejas de archivos.
- previene que se sobrescriba trabajo.

## ¿Qué es el control de versiones?

El control de versiones, alternativamente conocido también como control de código fuente o administración de código fuente, es un sistema que mantiene versiones de los archivos en las etapas progresivas del desarrollo. El sistema de control de versiones es similar en teoría a respaldar tus archivos, pero más inteligente. Cada archivo en el sistema tiene una historia completa de cambios, y puede ser fácilmente restaurado a cualquier versión de su historia. Cada versión tiene un identificador único que luce como una cadena de letras y números. (443e63e6..).

Hay muchos programas diferentes de control de versiones. Este documento se basa en git, pero también puedes estudiar Subversion (svn), CVS, darcs, Mercurial y otros. Cada uno tiene una slightly different metaphor for operation.

## Estructura del Repositorio

La versión más simple del sistema de control de versiones consiste de un *repositorio* en donde todos los archivos y sus versiones viven. Simplemente, un repositorio trabaja como una base de datos; puede devolver cualquier versión de cualquier archivo dentro, o el historial de cambios de cualquier archivo, o quizás un historial de los cambios a través del proyecto entero.

```
#25 Joe Adjust user profile information
```

```
#24 Fred Add login box
```

```
#23 Mary Allow user photo uploads
```

#22 Joe Change the color of the header to yellow

#21 Mary Change the header to blue

El repositorio de usuarios puede *chequear una copia de trabajo*, la cual es una copia de los últimos archivos a los que los usuarios pueden aplicar cambios. Después de hacer algunos cambios, los usuarios pueden entonces ingresar (check in o commit) los cambios al repositorio, el cual crea una nueva versión con metadata referida a los cambios que fueron aplicados y a la persona que los hizo.

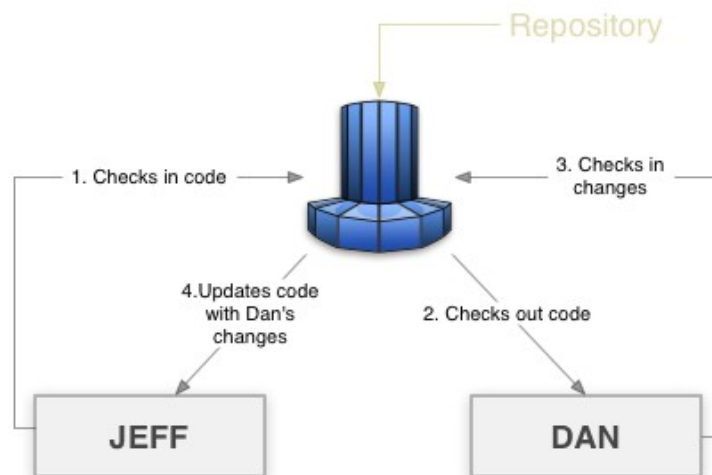


Figura 1: Un sistema básico de control de versiones

Si bien la forma más simple es tener un fuente canónico para el repositorio, esto no es necesario. Cada usuario tiene una copia completa del repositorio en su maquina local. Generalmente, tu aplicarás cambios a tu repositorio local, y una vez que esté completo, *empujaras* (push) tu trabajo al repositorio compartido de tu equipo. También puedes *tirar* (pull) de los cambios de otros repositorios.

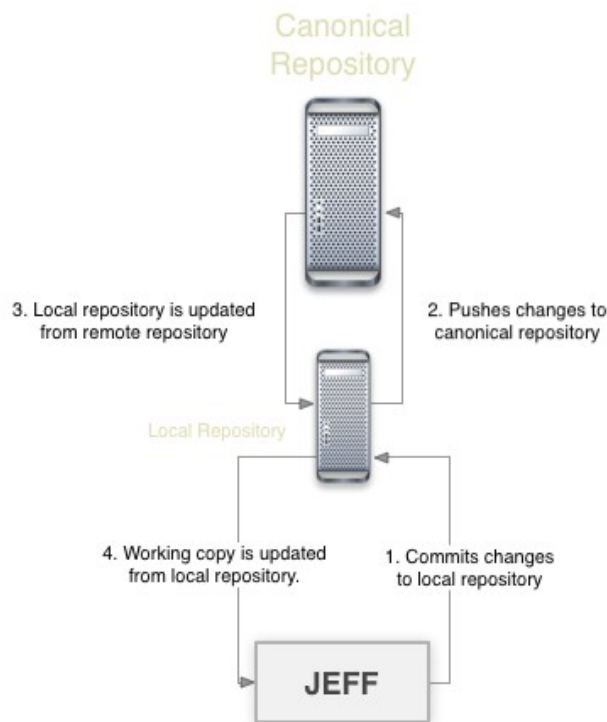


Figura 2: Un sistema de control de fuentes distribuido

## Ramas (Branches)

Las ramas cumplen el mismo rol que el borrador cuando escribes un mail. Puedes trabajar en un borrador, guardándolo frecuentemente hasta que esté completo.; entonces, cuando esté listo, envías el email, y el borrador es eliminado. En este caso, la carpeta de Enviados no se poluciona con tus cambios frecuentes, hasta que tu presionas “enviar”.

El branching es útil cuando desarrollas nuevas características, debido a que permite a la rama maestra –la carpeta de Enviados – estar trabajando siempre y deployable. Puede haber cualquier número de borradores –ramas experimentales- en desarrollo activo. Las ramas son fáciles crear y de intercambiar.

Una vez que el código en una rama se finaliza y la rama pasa sus tests, los cambios son mezclados (merged) en la rama maestra y la rama es removida, como ocurre con el borrador del correo. Si alguien aplica cambios de código (commit) a la rama maestra, es fácil actualizar la rama al último código maestro.

## Flujo de trabajo (Workflow)

### El ataque de los clones

Para tener una copia de trabajo de tu código base, necesitas clonar un repositorio remoto en tu máquina local. El clonado crea el repositorio y obtiene la última versión, la cual es referida como *HEAD*.

Clonemos un proyecto open-source.

```
$ git clone git://github.com/wycats/jspec.git
Initialized empty Git repository
```

Felicitaciones, has clonado tu primer repositorio. El comando clone setea unos pocos y convenientes ítems por tí; mantiene la dirección del repositorio original, y le da sobrenombre *origin*, de modo que puedas fácilmente enviar de regreso los cambios (si tienes autorización) al repositorio remoto.

Ahora tienes una carpeta **jspec** en el directorio corriente. Si haces **cd** hacia ese directorio, deberías poder ver el contenido del código fuente de JSpec (son solo unos pocos archivos)

Git puede correr sobre muchos protocolos, incluyendo “git://” como mas arriba (la mayoría de los proyectos públicos usaran git://). Por defecto, git usa el protocolo ssh, el cual requiere que tengas un acceso seguro al repositorio remoto.

```
$ git clone user@yourserver.com:thing.git
```

Puedes especificar los detalles de tu autorización para ssh como se ve arriba.

## Haciendo cambios

Ahora que tienes una copia de trabajo, puedes empezar a realizar cambios. No hay nada mágico respecto a la edición de archivos, por lo que todo lo que necesitarás hacer es editar los archivos sean cual sean y entonces salvarlos. Una vez que el archivo es salvado, necesitaras *agregar* (add) el cambio a la revisión actual (o más típicamente, harás los cambios a varios archivos y los agregarás a la revisión actual todo de una sola vez). Para hacer esto, necesitas hacer **git add** para agregar el archivo con cambios. Esto es conocido como “staging”.

```
$ git add index.html
```

O, también puedes agregar un directorio entero de una vez,

```
$ git add public/
```

Eso agregará cualquier archivo en el directorio **public/** a la revision, O, agregar el directorio actual:

```
git add .
```

Si haces cualquier cambio después del staging (antes de commit), necesitarás **git add** el archivo de nuevo.

El comando **git status** muestra el estado actual del repositorio.

```
ninja-owl:public courtenay$ git status
# On branch master
# Changes to be committed:
# (use “git reset HEAD <file>...” to unstage)
#
```

```
# modified: public/index.html
#
```

El comando **git diff** te muestra una vista diferencial de que es lo que ha cambiado. Por defecto, muestra los cambios que no han sido “estageados”. Agregando la bandera “-cached” mostrará los cambios “estageados” solamente.

```
ninja-owl:public courtenay$ git diff --cached
diff --git a/public/index.html b/public/index.html
index a04759f...754492a 100644
--- a/public/index.html
+++ b/public/index.html
@@ -8,7 +8,6 @@ revision control or source code management, is a system that
maintains versions
+This a line that I added to the file
- This is a line I removed from the file
```

Esta salida es denominada un **diff** o **match** y puede ser enviada por email a los otros desarrolladores de modo que ellos puedan aplicar tus cambios a sus códigos locales. Es también human-readable: muestra los nombres de archivos, los números de línea dentro del archivo, y los cambios con los símbolos + y -. Puedes también *entubar*(pipe) el diff en un archivo.

```
ninja-owl:public courtenay$ git diff --cached > line_modify.patch
```

## Comprometer (commit) algo en tu vida

Cuando ya tienes tus cambios de modo que quieras agregarlos a la revisión actual, entonces necesitas comprometer (commit) esta revisión a tu repositorio local. Para hacer esto, necesitarás correr **git commit**. Cuando ejecutas este comando, aparecerá un editor de textos, con una lista de los archivos que han cambiado y un poco de espacio vacío en la parte superior. En este espacio en blanco, necesitas describir lo que has cambiado de modo que tus colegas puedan decir a primera vista lo que tu has hecho. Necesitarás ingresar algo mejor que “cosa”, pero tampoco hay necesidad de escribir un testamento, y en cambio hacer algo como:

```
Changed line 434 in index.html to use spaces rather than tabs.
Changed line 800 in products.html.erb to have two spaces between the tags.
Changed line 343, 133, 203, 59, and 121 to have two spaces at the start rather than 9.
```

Una descripción corta de lo que haz cambiado será suficiente. Los mensajes commit son una forma de arte, como el haiku.

```
Minor formatting changes in the code.
```

Es aceptable escribir una línea de resumen (menos de 80 caracteres), una línea en blanco, y luego una tercer línea describiendo en mas detalle. Las segundas y terceras líneas son opcionales.

Una vez que hayas escrito el mensaje de commit, salva el archivo y sal del editor de textos. Esto se comprometerá a tu repositorio local, y puedes continuar con tu trabajo.

## Push back

Una vez que tus cambios hayan sido “comprometidos” a tu repositorio local, necesitarás empujarlos (push) al remoto. Para hacer esto, necesitarás ejecutar **git push**, el cual empujará todos los cambios desde tu repositorio local al remoto.

El push de git lleva varios argumentos: **git push <repository> <branch>** . En este caso, queremos empujar de regreso los cambios al repositorio original, el cual es “sobrenombrado” como origin, a la rama maestra.

```
$ git push origin master
```

Afortunadamente para nuestros dedos, git push ( y git pull) por defecto empujará y tirará de todas las ramas comunes al origen y al repositorio local.

Cuando ejecutas **push**, deberías ver una salida similar a la siguiente:

```
your-computer:git_project yourusername$ git push
updating 'refs/heads/master'
1.    from fdbdfe28397738d0d42eaca59c6866a87a0336e2
2.    to 1c9ec11f757c099680336875b825f817a992333e
Also local refs/remotes/origin/master
Generating pack...
Done counting 2 objects.
Deltifying 2 objects...
 100% (2/2) done
Writing 2 objects...
 100% (2/2) done
Total 2 (delta 3), reused 0 (delta 0)
refs/heads/master: fdbdfe28397738d0d42eaca59c6866a87a0336e2 → 1c9ec11f757c099680336875b825f817a992333e
```

Toda esta salida básicamente dice que tienes tus archivos listos para ser empujados (pushed) (**Generating pack**) y que el repositorio remoto ha recibido tus archivos (**Writing 2 objects**). Es entonces cuando el repositorio remoto ha actualizado su **head/master** (la rama “main” del repositorio) al punto de la revisión que has comprometido (commit) por lo que es conocido como el ultimo conjunto de cambios cometidos. Ahora otros pueden actualizar sus copias locales para estar sincronizados con los cambios que haz hecho. ¿Pero como haces esto?

## Obtener actualizaciones desde lejos

Para actualizar tu repositorio local y una copia de trabajo a la última revisión cometida al repositorio remoto, necesitas ejecutar **git pull**. Esto “tira” todos los cambios desde el repositorio remoto y los combina con tus cambios actuales (si hay alguno).

Cuando ejecutas un **git pull**, la salida debería lucir algo como lo siguiente:

```
remote: Generating pack...
```

```
remote: Done counting 12 objects.
```

```
remote: Result has 8 objects.
```

```
remote: Deltifying 8 objects...
```

```
remote: 100% (8/8) done
```

```
Unpacking 8 objects...
```

```
remote: Total 8 (delta 4), reused 0 (delta 0)
```

```
100% (8/8) done
```

3. \* refs/remotes/origin/master: fast forward to branch ‘master’ of git@yourco.com:git\_project  
old..new: 0c793fd..fdbdfe2

```
Auto-merged file.cpp
```

```
Merge made by recursive.
```

```
.gitignore | 2 ++
```

```
file.cpp | 8 ++++++--
```

```
src/things.html | 5 +++--
```

```
your_file.txt | 18 ++++++
```

```
4 files changed, 19 insertions(+), 4 deletions(-)
```

```
create mode 100644 .gitignore
```

```
create mode 100644 your_file.txt
```

Lo que ha sucedido es básicamente un **push** en reversa. El repositorio remoto ha preparado (**Generating pack**) y transferido los cambios a tu repositorio local (**Unpacking 8 objects**). Tu repositorio local entonces toma los cambios y los implementa en el mismo orden que fueron cometidos( ej. combinándolos como muestra el ejemplo para **file.cpp** o creándolos como **.gitignore** de tu **tu\_archivo.txt**).

**Nota:** El archivo **.gitignore** te permite decirle a Git que ignore ciertos archivos y directorios. Esta opción es útil para cosas como binarios generados, archivos log, o también aquellos con passwords. locales en ellos.

## Branching

Deberías siempre crear una rama antes de comenzar a trabajar en una nueva característica. De esta forma, el master estará siempre en un estado estable, y tu tendrás la posibilidad de trabajar aisladamente de los otros cambios. El crear nuevas ramas te permite tomar la rama **master**, “clonarla”, y aplicar cambios a ese “clonado”. Entonces cuando estás listo, puedes mezclar tu rama con la **master**; o, si ha habido cambios al master mientras tu estabas trabajando, mezclarlos a tu propia rama. Es como tirar y empujar, pero todo ocurre en el mismo directorio.

La figura inferior ilustra el proceso.

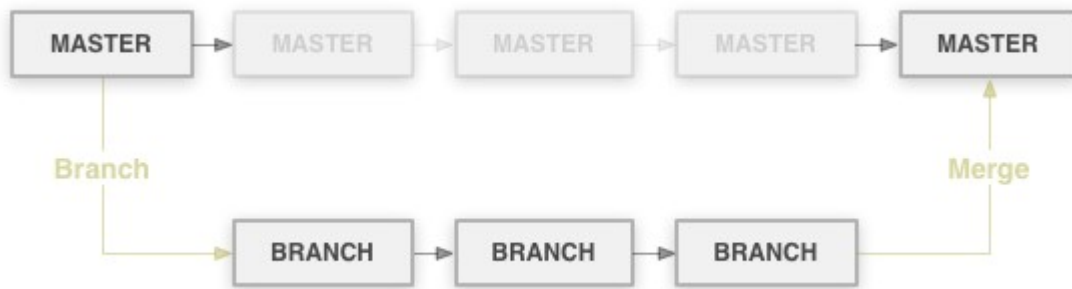


Figure 3: Branching y mezclado (merging)

El branching es genial para que dos personas trabajen juntas en cosas que requieren aislamiento del código base principal. Esto podría incluir cualquier cosa, desde código que tendrá resultados permanentes, como por ejemplo una refactorización de código realmente grande o el rediseño de un sitio, hasta cosas temporales, como es el caso de testing de performance.

### Creando una rama (Branch)

Para crear un Branch en Git, debes ejecutar **git checkout -b <nombre de la rama>**. Cualquier archivo modificado será listado.

```
$ git checkout -b redesign
M public/index.html
Switched to a new branch "redesign"
```

Ahora has chequeado la rama de rediseño. Para switchear al master,

```
$ git checkout master
M public/index.html
Switched to a new branch "master"
```

Encontrarás muy útil crear la rama en el repositorio remoto, de modo que los otros puedan tirar de tus cambios.

```
$ git push origin redesign
```

Puedes también empujar tu rama a otra rama diferente remotamente

```
$ git push origin redesign:master
```

Esto establece la copia de trabajo actual para cometer y empujar todos los cambios a la rama **redesign** en los repositorios locales y remotos. Ahora cualquier cambio que agregues y cometas vivirán en esta rama en vez de la **master**.

**Aside:** ¿En qué rama estoy? Para ver tu rama actual, y para listar todas las ramas locales, ejecuta **git branch**

Si necesitas tirar de los cambios desde **master** a tu rama (por ejemplo, cambios importantes de código, actualizaciones de seguridad, etc.) lo puedes hacer usando **git pull**:

```
git pull origin
git merge master
```

Este comando le dice a Git que tire todos los cambios desde el repositorio **origin** ( el nombre Git para el repositorio canónico remoto) incluyendo todas las ramas. Entonces, combinas la rama master a la tuya. Cuando estas listo para combinarla con la **master**, necesitas verificar a **master** y entonces combinar las ramas:

```
git checkout master
git merge redesign
```

Ahora tus cambios serán combinados a la rama **master** desde la rama **redesign**. Si terminaste con la rama que creaste, puedes borrarla usando el parámetro **-d**.

```
git branch -d redesign
```

Para borrar la rama en el repositorio remoto, debes capturar el comando push (recuerda que puedes empujar una rama local a una rama remota con **git push <remote> <local branch>:<remote branch>** ) y enviar una rama local vacía a la rama remota.

```
git push origin :redesign
```

## Más herramientas útiles

### Deshaciendo tus cambios

Puedes remover un archivo del staging con **git reset HEAD <filename>**.

Si quieres revertir un archivo a la copia en el repositorio, solo chequealo de nuevo con **git checkout <filename>**

Para revertir un archivo a una versión mas vieja, usa **git checkout**. Necesitarás saber el ID de revisión, el que puedes encontrar con **git log**:

```
$ git log index.html
commit 86429cd28708e22b643593b7081229017b7f0f8d
Author: joe <joe>
Date: Sun Feb 17 22:19:21 2008 -0800
    build new html files

commit 3607253d20c7a295965f798109f9d4af0fbedd8
Author: fred <fred>
Date: Sun Feb 17 21:32:00 2008 -0500
    Oops.
```

Para revertir el archivo a la versión mas vieja (360725...) ejecutas checkout. Git establecerá la versión anterior por ti, lista para ser revisar y cometer.

```
$ git checkout 3607253d20c7a295965f798109f9d4af0fbedd8 index.html
```

Si ya no quieres restaurar esta versión vieja, puedes volver atrás de nuevo.

```
$ git reset HEAD index.html
```

```
$ git checkout index.html
```

O en un comando

```
$ git checkout HEAD index.html
```

¿Has notado que HEAD es intercambiable con el numero de revisión? Esto es así porque con git, las revisiones y ramas son efectivamente la misma cosa.

## ¿Quién escribió esa línea?

Corre **git blame** <file> para ver quien cambió el último archivo y cuando.

## Ver el árbol completo

Puedes ver una historia detallada de tu copia de trabajo con **gitk**.

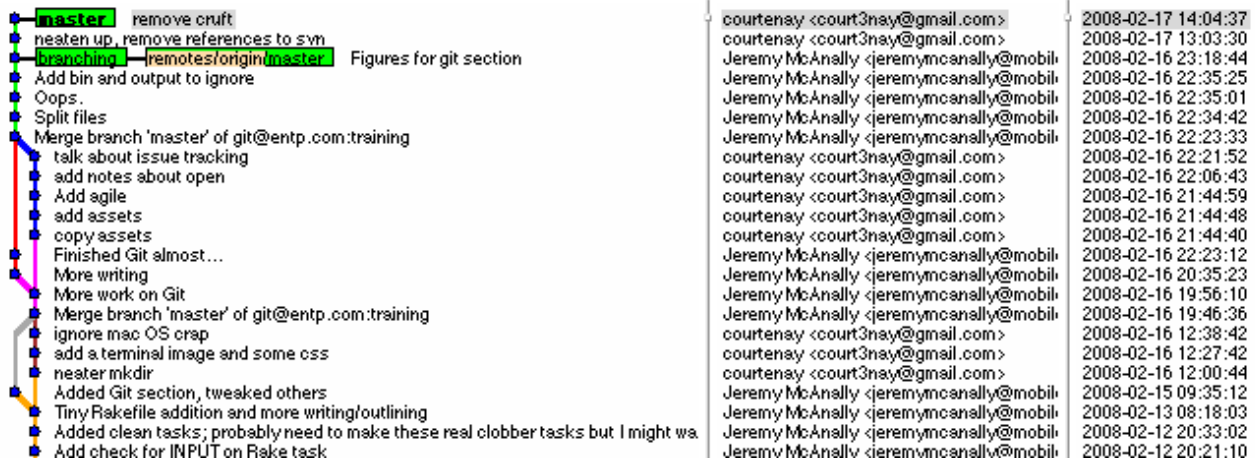


Figure 4: Sample gitk screenshot

La aplicación gitk te permite navegar a través del árbol de cambios, ver diferencias, buscar revisiones viejas y más.

## Buenas prácticas

Pensamos que podemos pasar esta sección con unas pocas hints y tips que pueden ayudarte cuando estas trabajando con sistemas de control de versiones.

## Comete(commit) con frecuencia

Como cuando la gente siempre dice “graba con frecuencia o te arrepentirás” cuando trabajas con procesadores de texto, deberías cometer a tu repositorio local tan frecuentemente como sea posible.

No solo te resguarda de la posibilidad de perder tu trabajo (que no debería ocurrir si sigues la primera pieza de este aviso), sino que te dará la seguridad que puedes volver atrás en cualquier momento que lo necesites. Por supuesto, committing *commit* after *commit* every *commit* *wocommitrd* or *lcommitecommitcommittcommitecommitrcommit* podría ser un tanto excesivo. En cada paso que realices en tu trabajo, deberías cometer. *you take in your work, you should commit.*

## Tira con frecuencia

A la inversa, deberías tirar con frecuencia. El tirar frecuentemente mantiene tu código al día y, con suerte, elimina la posibilidad de duplicar trabajo. Siempre es frustrante cuando dedicas horas trabajando en una característica y tu colega ya la implementado y empujado al repositorio, pero no lo sabías debido a que tiras cada tres semanas.

## Usa *checkout* y *reset* con cuidado

Para revertir cualquier cambio local que hayas hecho a un archivo específico desde tu último commit, puedes usar **git checkout <filename>**, o también usar **git reset** para matar todos los cambios desde tu último commit. Tener la habilidad de volver pasos atrás es una gran herramienta (especialmente si te das cuenta que estás siguiendo un camino *totalmente* errado), pero definitivamente es una espada de doble filo. Una vez que cambios fueron, ellos fueron, por lo que ten cuidado!. Es terrible cuando te das cuenta que tiraste por la borda algunas horas de trabajo por un reset sin retorno.

## Crear tu propio repositorio donde sea.

Si quieres tener algún control de versiones en un siempre proyecto local (por ej., no tiene un gran repositorio remoto o similar), entonces simplemente puedes usar **git init** para crear tu propio repositorio local. Por ejemplo, si estás trabajando en algunos conceptos de diseño de una aplicación, entonces podrías hacer algo como esto:

```
mkdir design_concepts
git init
```

Ahora puedes agregar archivos, commit, branch, y así, como en un repositorio Git remoto “real”. Si quieres empujar y tirar, necesitarás configurar un repositorio remoto.

```
git remote add <alias> <url>
git pull <alias> master
```