Control de versiones flexible, potente y útil.

# PRESTIDI-GIT-ACIÓN

Todo aquel que haya editado parte de un código o de un texto y se haya dado cuenta más tarde de que prefería una versión anterior del mismo, habrá notado que necesita un sistema de control de versiones. Mostraremos cómo nos puede ayudar Git en este menester. **POR JULIET KEMP**

Con el software de control de versiones podemos seguir la pista al historial del proyecto y revertir cualquier edición cuando cambiemos de idea. Además, podemos crear y guardar evidencia de múltiples versiones y ramas de nuestro trabajo. Todos los proyectos de software usan invariablemente algún tipo de control de versiones y cada vez se emplea más Git [1]. Git fue originalmente diseñado por Linus Torvalds para ser usado por el equipo de desarrollo del kernel Linux, pero su flexibilidad, su velocidad y una estructura altamente distribuida lo convierten en el candidato ideal para cualquier tipo de proyecto.

Git está pensado para estar altamente distribuido y ser muy rápido y flexible. "Distribuido" significa que, a diferencia de otros muchos mecanismos de control de versiones, como *CVS* o *Subversion*, Git no necesita un repositorio central definitivo. En lugar de eso, todos los repositorios tienen el mismo status y cualquiera de ellos puede actualizarse con el resto. Es algo que funciona muy bien en proyectos altamente cooperativos [2].

Una de las funcionalidades principales de Git es que fue diseñado para soportar un desarrollo no lineal: Espera cambios que podrán combinarse repetidamente conforme pasan por diferentes revisores y desarrolladores. Como consecuencia, es fácil combinar ramas, o incluso árboles enteros, independientemente de si se comparten o no ancestros. Otra característica que lo diferencia del resto de sistemas de control de versiones es la facilidad para combinar archivos no versionados en un árbol existente. Se trata de algo genial en proyectos cuyo desarrollo está extremadamente repartido, pero también aporta una flexibilidad sin parangón cuando se usa de modo individual.

## Nuestro Primer Repositorio Git

Para comenzar, instalamos el paquete *git-core* (así se llama en *Debian*, *Ubuntu* y *Fedora*) a fin de obtener las piezas básicas; conviene también disponer del paquete *git-doc* (la documentación). Es posible obtener también otras extensiones; como *git-svn*, que permite interoperar con Subverion; o *gitweb*, que proporciona una interfaz web. Alternativamente, se puede instalar desde los fuentes [3].

Una vez instalado Git, el primer repositorio es muy fácil de crear. La naturaleza distribuida de Git implica que cada copia de trabajo lleve consigo su propio repositorio (en el subdirectorio *.git*), en vez de residir éste en una ubicación centralizada, como ocurre en sistemas del estilo de CVS o SVN. Por tanto, para poner bajo control de versiones un directorio dado, el proceso es increíblemente sencillo:

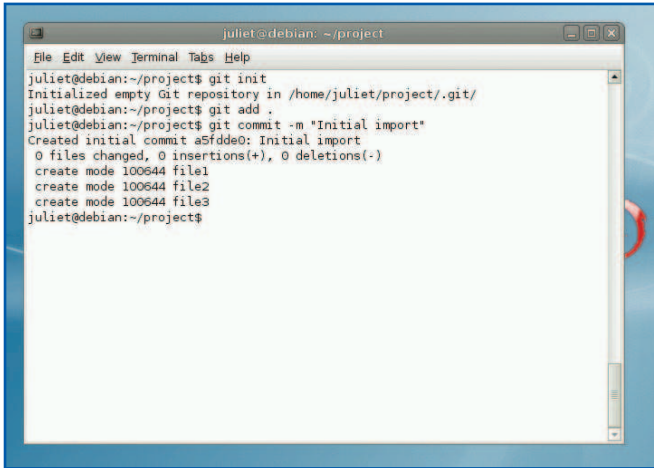


**Figura 1: Configurar un directorio para usarlo como repositorio de Git no tiene ninguna complicación, como se puede apreciar por el código del texto.**

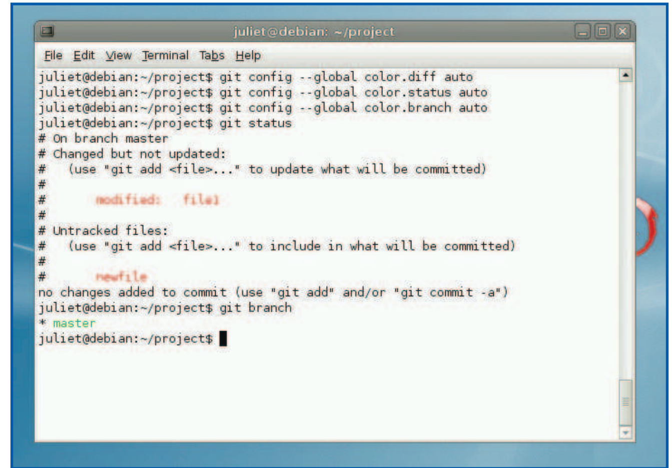


**Figura 2: Salida coloreada de *git status* y *git branch*. Las líneas con # del centro son la salida relativa al estado.**

```
cd mi_directorio
git init
git add .
git commit
```

Una vez encomendado a Git el directorio (Figura 1), podemos probar a añadir otro archivo.

```
touch nuevo_archivo.txt
```

y a ejecutar

```
git status
```

Un mensaje nos advierte de que tenemos un archivo cuya actividad no está siendo registrada. Para añadir este archivo e incluir los cambios en el repositorio introducimos

```
git add nuevo_archivo.txt
git commit
```

Se abrirá un editor de textos con el que editar el mensaje del commit; si se quiere evitar esto, pueden enviarse los cambios junto con el mensaje directamente con

```
git commit -m "Mensaje"
```

## Listado 1: Configuración Commit por Repositorio

```
01 cd /home/juliet/mi_reposito-
   rio_anonimo
02 git config user.name Cualquier
   Nombre
03 git config user.email cual-
   quier.nombre@gmail.com
```

El mensaje se guardará en el registro. Conviene siempre escribir mensajes descriptivos, por si hubiera que volver atrás en algún momento.

Para añadir todos los archivos que no estén añadidos ya en un directorio, basta con hacer:

```
git add .
git commit -m "Message"
```

Luego podemos probar a editar uno de los archivos y a introducir *git status* de nuevo. Se puede ver que el archivo en cuestión aparece como *Changed but not updated* (*Modificado, pero no actualizado*). A diferencia de otros sistemas, Git no añade ni edita a menos que se le indique explícitamente. Para añadir cualquier archivo que ya esté siendo controlado y haya sido modificado, se puede hacer

```
git add nombre_de_archivo
```

o bien

```
git commit
```

o incluso

```
git commit -a
```

Así no se añadirán, sin embargo, los archivos que no estén siendo controlados (hay que hacerlo explícitamente con *git add*).

Alternativamente, podemos eliminar el archivo de prueba y enviar los cambios con

```
git rm nuevo_archivo.txt
git commit
```

En el supuesto de que el archivo haya sido editado después del último commit, se mostrará un aviso advirtiendo de tal circunstancia; en caso de estar seguros, se puede forzar la eliminación con

```
git rm -f nuevo_archivo.txt
```

Al ejecutar *git status*, se muestra el estado de todos los archivos del directorio, pudiéndose añadir cualquier archivo. Sin embargo, podría ocurrir que existan archivos cuya adición no queramos que tenga lugar jamás (archivos de respaldo o archivos temporales, por ejemplo), porque la salida de *git status* estaría siempre recargada con archivos que no tendríamos por qué ver. Estos archivos supondrían también un obstáculo a la hora de hacer *git add .*, ya que ralentizarían considerablemente la adición. La solución a este problema pasa por crear un archivo de texto llamado *.gitignore* en el directorio de trabajo actual que incluya un listado con todos los archivos que no se han de controlar o monitorizar.

Un ejemplo de archivo *.gitignore* podría tener el siguiente aspecto:

```
.*.sw*
tmp
images
```

Con esta configuración, los archivos con nombres como *.mi_archivo.swp* (los archivos temporales de Vim tienen este tipo de nombres) serán ignorados, al

igual que los contenidos en los directorios *tmp/* e *images/*.

Si en algún momento se quiere dejar de controlar un directorio (es decir, se desea que deje de ser un repositorio de Git), basta con eliminar el directorio *.git*:

```
rm -rf .git
```

Entonces, al ejecutar

```
git status
```

el programa nos responde diciendo que el directorio no es un repositorio de Git:

```
fatal: Not a git repository (or any
 of the parent directories): .git
```

No hace falta decir que el hecho de que sea tan fácil eliminarlo de este modo implica que haya que guardar una copia de seguridad del repositorio en todo momento, ¿verdad?

## Configuración de Git

Llegados a este punto, puede aparecer un aviso relativo a nuestro nombre y nuestra dirección de correo electrónico; en Git, todos los commits tienen asociado el nombre de quien envía los cambios y su dirección de correo electrónico.

Git trata de obtener dicha información directamente desde la máquina, pero a veces puede que no lo consiga. Para corregir la información y desactivar el aviso, se pueden definir ambos campos manualmente:

```
git config --global user.name ➥
Juliet Kemp
git config --global user.email ➥
juliet@earth.li
```

Obviamente, si el repositorio es privado, estos valores no tendrán mucha trascendencia. Sin embargo, sigue habiendo dos buenas razones para definirlos. En primer lugar, desaparece el molesto aviso; en segundo lugar, estas propiedades se configuran globalmente; por tanto, una vez configuradas, se usan en todos los repositorios Git en los que actúe el mismo usuario. De este modo, cuando contribuimos a otro proyecto que use Git, este nombre de usuario y este correo electrónico serán los que identifiquen nuestros commits. De todas formas, se puede establecer una configuración por repositorio, como se ilustra en el Listado 1.

Además, se pueden configurar otras muchas cosas. Por ejemplo, se le puede decir a Git que coloree las diferencias mostradas en las salidas producidas por sus comandos *diff*, *status* y *branch* (Figura 2):

```
git config --global color.diff ➥
auto
git config --global ➥
color.status auto
git config --global ➥
color.branch auto
```

En el manual de usuario [4] y el tutorial de Git [5] existe más información sobre todas estas opciones de configuración y otras muchas.

## Usando Git con un Repositorio Externo

Hasta ahora sólo hemos visto cómo usar Git con proyectos privados, ya sean nuevos o existentes. Sin embargo, para poder implicarnos en un proyecto controlado con Git en el que ya exista código, necesitaremos importar antes el código base desde un repositorio externo. Dada la naturaleza distribuida de Git, se puede importar desde la copia de cualquier otra persona, siempre y cuando nos permita el acceso a su repositorio. Sin embargo, la mayoría de los proyectos, por conveniencia y también para facilitar la publicación del código, tendrán un repositorio central desde el cual importar, que hace las veces de versión maestra del proyecto.

La importación de un proyecto es tan sencilla como la creación de un repositorio propio. Para descargar el árbol del kernel Linux, por ejemplo, usamos el siguiente comando:

```
git clone git://git.kernel.org/➥
pub/scm/linux/kernel/git/➥
torvalds/linux-2.6.git ➥
linux-2.6
```

Ojo: ¡Este árbol es bastante grande y tardaremos un buen rato en bajárnoslo!

Otros proyectos de Linux que usan Git son *GIMP*, *Debian*, *One Laptop Per Child* y *Perl* (ver la página de Wikipedia sobre Git para conocer más proyectos); sus sitios web informan sobre cuál es el repositorio principal que utilizan.

Una vez en posesión de la copia del repositorio principal del proyecto, ya podemos empezar a trabajar con el código. Crear una nueva rama para albergar nuestros cambios siempre es una buena idea. Luego, podemos utilizar *git pull* para actualizar desde el repositorio desde el cual clonamos inicialmente el código. Más tarde, puede que lo que queramos sea generar un parche con nuestros cambios y enviarlos al proyecto para su revisión e inclusión. La utilización de ramas, la actualización y el parcheado serán asuntos que abordaremos más adelante en este artículo.

## Análisis del Historial de Revisiones

Al igual que cualquier otro sistema de control de versiones, Git guarda un historial completo con todas las revisiones del proyecto, almacenando los cambios hechos con cada commit. Por tanto, se pueden comparar diferentes versiones, comprobar acciones pasadas y restaurar versiones anteriores.

Hay varios comandos disponibles con los que analizar el historial de un proyecto. El comando *git show* muestra los detalles del último commit; dará el ID de revisión, el autor, la fecha, el mensaje y un diff con los cambios (Listado 2). El comando *git log* muestra una línea breve por cada uno de los commits del historial; en la página de manual se puede encontrar información sobre cómo controlar lo que se muestra y lo que no. El comando *git log nombre_de_archivo* puede resultar particularmente útil, ya que sólo muestra los commits que afectan a *nombre_de_archivo*. Además, se puede tener una salida monolínea mucho más agradable con

```
git log pretty=oneline.
# Otras opciones, aparte de
# oneline, son short, medium,
# full, fuller, email y raw
```

Los commits tienen IDs alfanuméricos largos (ver línea 1 del Listado 2) que son la consecuencia de una interesante característica de seguridad de Git: El nombre de cada commit se obtiene calculando un hash a partir de los contenidos del commit, de modo que se garantiza el historial (no se pueden cambiar los contenidos del commit sin cambiarle el nombre), siendo dicho nombre único globalmente. Por tanto, a la hora de trabajar con otras personas, todas saben que se trata de ese commit, independientemente de en qué

repositorio se encuentren. Obviamente, trabajar con estos números de revisión es más tedioso, pero Git autocompleta el nombre de revisión al indicarle los primeros caracteres (además de que siempre se puede copiar y pegar).

El comando *git diff* muestra las diferencias entre la última versión enviada y la copia local. Para ver las diferencias entre dos versiones anteriores se usa

```
git diff commitID1 commitID2
```

Para revertir (es decir, cancelar) un commit, se utiliza

```
git revert commitID
```

O, para restaurar el árbol completo a la última revisión:

```
git checkout
```

¡Ojo! ¡Con este último comando se sobrescriben todos nuestros cambios locales!

## Ramas y Combinaciones

Git es tremendamente flexible, y sus posibilidades en cuanto a creación y combinación de ramas son casi infinitas. Las ramas (múltiples copias de un mismo repositorio) nos permiten separar una serie de cambios mientras experimentamos con ellos o crear diferentes versiones de un proyecto, sin alterar el árbol principal del mismo. Las ramas en Git se pueden hacer rápida y fácilmente.

```
juliet@debian: ~/project
File Edit View Terminal Tabs Help

juliet@debian:~/project$ git branch V1.5
juliet@debian:~/project$ git branch
  V1.5
* master
juliet@debian:~/project$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   file1
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       newfile
no changes added to commit (use "git add" and/or "git commit -a")
juliet@debian:~/project$ git checkout V1.5
M       file1
Switched to branch "V1.5"
juliet@debian:~/project$ git status
# On branch V1.5
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   file1
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       newfile
no changes added to commit (use "git add" and/or "git commit -a")
juliet@debian:~/project$
```

**Figura 3: Creación y comprobación de una nueva rama. Nótese el nombre de la rama en la parte superior de la salida de estado.**

Combinarlas con el árbol principal es igual de fácil, por lo que merece la pena usarlas siempre que se trabaje en algo nuevo.

Los comandos básicos son:

- *git branch* para listar las ramas actuales del proyecto.
- *git branch NombreDeLaRama* para crear una nueva rama.
- *git branch -d NombreDeLaRama* para borrar una rama.
- *git checkout NombreDeLaRama* para actualizar el directorio de trabajo actual de modo que refleje la rama *NombreDeLaRama*. Para crear una rama al tiempo que hacemos un checkout utilizamos *git checkout -b NombreDeLaRama*.

Imaginemos que queremos crear una nueva rama a partir de nuestro proyecto y llamarla *V1.5*. Lo primero que hay que hacer es crear la nueva rama:

```
git branch V1.5
```

El repositorio cuenta ahora con dos ramas, pero el directorio de trabajo actual aún se corresponde con la rama principal (cuyo nombre por defecto es *master*). Si ejecutamos *git branch*, veremos un listado con todas las ramas, estando marcada con un asterisco la que se encuentra en uso en ese momento (Figura 3).

Para cambiar a la nueva rama y comenzar a trabajar en ella, introducimos

```
git checkout V1.5
```

El comando

```
git status
```

mostrará entonces el nombre de esta rama con un asterisco junto a él. Efectuaremos cambios y los enviaremos normalmente, y luego usaremos el comando

```
git checkout ➘
master
```

para volver a la rama principal. Nuestros cambios dejarán de ser visibles entonces. Al volver a la rama V1.5, volverán a estar ahí.

Para combinar las ramas de manera que formen una sola de nuevo, usamos

```
git merge NombreDeLaRama
```

De este modo, combinamos *NombreDeLaRama* con la rama en la que nos encontramos (en el directorio de trabajo actual). Por lo tanto, en este ejemplo, lo que hacemos es volver a la rama principal (*git checkout master*) e introducir *git merge V1.5* para combinar los cambios que hay en ella. A partir de entonces, la rama principal y la rama *V1.5* serán idénticas, aunque *git status* nos dice que la rama V1.5 aún existe. No hay ningún problema en volver a la rama V1.5 y seguir trabajando en ella.

Nótese que Git se negará a combinar las ramas si tenemos en ellas cambios sin enviar, ya que al combinarlas se sobrescribirá el directorio de trabajo actual. Si por algún motivo no se pudiera hacer un commit, se puede usar *git stash* (ver "Funcionalidad Stash").

Al hacer combinaciones, de ser posible, Git completará el commit automáticamente por nosotros. Pero a veces hay conflictos en los archivos (por ejemplo, cambios que se afectan unos a otros). En esos casos, la combinación falla, proporcionando información sobre el archivo o los archivos problemáticos. El comando *git status* presentará estos archivos como *unmerged*. Para resolver el conflicto, se han de abrir los archivos problemáticos en un editor de textos. Las secciones relevantes pueden verse en el Listado 3.

Se muestran ambas versiones del archivo para que elijamos cómo queremos resolver el problema. Una vez edi-

### Listado 2: Salida de git show

```
01 commit
   351e8cf452b92ed591f19fdbd6302
   3a68475a364
02 Author: Juliet
   <juliet@earth.li>
03 Date: Tue Mar 2 11:47:20+0000
04 Testing delete
05 diff --git a/nuevo_archivo.txt
   b/nuevo_archivo.txt
06 deleted file mode 100644
07 index e69de29..0000000
```

tado el archivo, hacemos un *git add archivo.txt* por cada archivo en conflicto, para terminar con un *git commit* que envíe los cambios de la combinación. Como alternativa, podemos usar *git commit -a* para marcar automáticamente todos los conflictos como resueltos. Lógicamente, habremos de estar completamente seguros de que realmente están resueltos antes de hacerlo.

## Trabajar con Parches

Al colaborar varias personas, hay un par de formas de compartir los cambios. Una opción es que los demás importen nuestros cambios o que combinen directamente desde nuestro árbol. Sin embargo, el procedimiento habitual pasa por generar un parche (un archivo de texto que describe nuestras variaciones con respecto del árbol del proyecto) y enviarlo, por ejemplo, por email. Afortunadamente, el sistema Git facilita mucho esta labor.

Antes de comenzar a realizar cambios, conviene crear una nueva rama que los albergue. Por ejemplo, con

```
git checkout -b MiSolucion
```

Desde ese momento, trabajaremos en la rama *MiSolucion* (consultar el comando *git stash* en la siguiente sección si ya se ha comenzado a trabajar en la rama principal). Una vez satisfechos con los cambios realizados y enviados éstos, debemos asegurarnos de que nuestro repositorio local está totalmente actualizado con respecto al resto del proyecto. Para ello, volvemos a la rama principal mediante *git checkout master*, y usamos *git pull* para traer los cambios desde el repositorio desde el cual obtuvimos originalmente nuestra copia del código base. Entonces volvemos a nuestra rama con *git checkout MiSolucion* y ejecutamos *git rebase master*. Con este último comando se aplican a la rama *MiSolucion* todos los cambios que hayan tenido lugar en la rama principal desde que actualizamos nuestra copia por última vez. También actualiza el historial de modo que nuestra rama tenga su origen en la versión más reciente de la rama principal. Evidentemente, tendríamos que resolver los conflictos que se hubieran podido generar.

Y ya estamos preparados para generar el parche:

```
git format-patch master ⮐
--stdout > ⮐
misolucion-patch.diff
```

El comando compara la rama actual (en este caso, *MiSolucion*) con la rama principal, buscando cualquier commit que no se dé en esta última. Produce un parche por commit, los envía todos a la salida estándar y los redirige al archivo *misolucion-patch.diff*. También podemos usar

```
git format-patch master
```

para generar en el directorio actual un archivo por cada commit, o

```
git format-patch master -o ⮐
DIRECTORIO
```

para guardarlos en un directorio de nuestra elección. Los parches tendrán todos formato de email.

Para aplicar un parche ajeno se usa *git am*. Una vez más, creamos una nueva rama con

```
git checkout -b Parche_Sara
```

para no confundir el parche con nuestros propios cambios; las ramas se pueden crear y combinar muy fácilmente, así que ¿por qué no usarlas? Luego, aplicamos el parche de Sara con

```
git am parchedesara.diff
```

con lo que se aplican los cambios especificados en él. En proyectos gestionados externamente, probablemente no se desee combinar los cambios con la rama principal hasta que no se hayan aceptado centralmente (momento en el que actualizaremos al hacer *git pull*). Pero en proyectos de menor envergadura, si el parche nos parece correcto, podemos simplemente aplicarlo a la rama principal con

```
git checkout master
git merge Parche_Sara
```

Cabe destacar que el comando *git am* es capaz de manejar también los parches que llegan por email. Sólo hay que guardarlos todos en un buzón de correo (en formato mailbox estándar de Linux), y ejecutar

```
git am mailbox
```

para aplicar a la rama actual todos los parches contenidos en el buzón. El comando utiliza el campo *From:* de cada mensaje como autor del commit; el campo *Date:* como fecha del commit; y el campo *Subject:* como título del commit. El mensaje del commit lo formarán el campo *Subject:* más el cuerpo del correo (hasta el inicio del parche). Este es el formato utilizado por *git format-patch*.

## Funcionalidad Stash

Otra herramienta útil a la hora de generar parches o de trabajar en proyectos complejos es *git stash*. Este comando fue diseñado para aquellos momentos en los que se quieren mezclar los cambios fuera del árbol existente sin perderlos (por lo que no basta con revertir al commit anterior) o para cuando se cambia a otra rama.

Para guardar todos los cambios y devolver el proyecto al estado del último commit usamos:

```
git stash save "trabajando en ⮐
el proyecto foo"
```

Hacemos un pequeño arreglo y lo enviamos como de costumbre. Luego hacemos *git stash pop* para aplicar los cambios guardados.

El comando *git stash* puede resultar útil cuando a mitad de un cambio importante nos encontramos con algún bug pequeño. Podemos usar el siguiente comando para guardar nuestros cambios, arreglar el problema menor, hacer el commit y generar un parche:

```
git stash save "Trabajando ⮐
en un $asunto mayor"
```

Una vez hecho esto, con

```
git stash pop
```

**Listado 3: Conflictos con Merge Marcados en Archivos**

```
01 test
02 <<<<<<< HEAD
03 this is the master branch
04 =======
05 this is the V1.5 branch
06 >>>>>>> V1.5
```

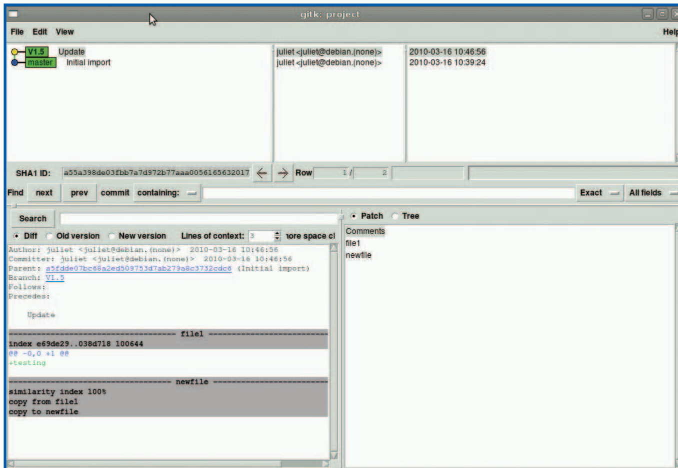


**Figura 4: Ventana de *gitk* mostrando el historial de revisiones, ramas y diffs.**

devolvemos al árbol los cambios que habíamos guardado. Como se puede apreciar, *git stash* y *git branch* parecen solaparse. Podemos pensar en *stash* como una especie de *branch* liviano.

Si en algún momento descubrimos que hemos estado trabajando sobre la rama equivocada, podemos echar mano de *git stash*. En este caso, guardamos los cambios al *stash*, nos cambiamos de rama y volcamos los cambios del *stash* a la nueva rama.

## Etiquetar con Git

El comando *git tag* nos permite adjuntar etiquetas a los commits. Las etiquetas se suelen usar para marcar los puntos de publicación, pero pueden usarse también para otras muchas cosas. Para ver un listado, ordenado alfabéticamente, con todas las etiquetas del directorio de trabajo actual, basta con ejecutar *git tag*. Se pueden buscar también patrones específicos. Con *git tag v1.** encontraríamos todas las etiquetas que comiencen por *v1*.

Para añadir una etiqueta propia al punto actual, usamos *git tag -a*:

```
git tag -a "Abril"
-m "Comienzo de Abril"
```

Con este comando se etiqueta el último commit con el nombre *Abril*, especificando como mensaje *Comienzo de Abril*. Si se omite el parámetro *-m*, git arranca un editor para introducir desde ahí el mensaje. Al ejecutar de nuevo *git tag*, se muestra la nueva etiqueta, mientras que ejecutando *git show Abril*, veremos información sobre la etiqueta así como los detalles del último commit.

En caso de disponer de una clave GPG privada, podemos usarla para firmar la etiqueta. Basta con usar *-s Etiqueta* en lugar de *-a Etiqueta*, para que se nos inste a introducir la frase de paso para la clave. Para verificar la firma de una etiqueta se usa *git tag -v Etiqueta*.

Además de todo esto, se puede crear una etiqueta ligera con *git tag Abril* que no guarda ningún mensaje ni ninguna firma GPG. El comando *git show Abril* mostraría únicamente la información del commit, sin información relativa a la etiqueta. Puede resultar útil como referencia rápida para un commit en particular, en vez de usar un marcador más detallado.

Por último, se puede etiquetar un commit anterior (distinto del último) especificando el inicio del ID del commit. Para averiguar el ID de un commit usamos *git log*. Una vez sepamos cuál es, ya le podemos aplicar la etiqueta. Por ejemplo, con

```
git tag -a Marzo
-m "Comienzo de Marzo" ad829ce
```

se etiquetarían todos los commits que comiencen por *ad829ce*.

## Más Cosas Útiles

En este artículo se ha cubierto lo básico para trabajar con Git de modo que nos ayude con nuestros propios proyectos o en proyectos de mayor envergadura. Pero Git es capaz de mucho más. Los comandos anteriormente mencionados son muy flexibles y aceptan una gran variedad de opciones. Las páginas de manual se pueden consultar con man *git-[herramienta]* (por ejemplo, *man git-add*). Todos los ejemplos mostrados aquí actúan sobre el último commit, pero son muchos los comandos que aceptan rangos de IDs de commits.

Otras funcionalidades estupendas que merece la pena investigar son:

- Instalación del paquete *gitk*, que muestra de forma gráfica el historial de revisiones del proyecto (Figura 4).
- Git, al igual que otros sistemas de control de versiones, soporta disparadores de scripts (*hooks*). De este modo, se pueden ejecutar scripts antes y después de determinados eventos. Por ejemplo, podríamos ejecutar una comprobación antes de permitir un commit, buscar espacios en blanco, o enviar un email después de cualquier evento. El directorio *.git/hooks/*, que se encuentra dentro de la copia de trabajo, contiene varios archivos con extensión *.sample* a los que se les puede quitar la extensión *.sample* para utilizarlos como ejemplo.
- Herramientas para interactuar con otros sistemas de control de versiones. Por ejemplo, *git-svn* nos permite usar un repositorio de Subversion [6] existente.

Ojo, a diferencia de los sistemas centralizados, Git no proporciona ningún tipo de mecanismo de backups: Si se elimina un directorio accidentalmente, se borra también el historial de revisiones. Habrá que mantener en todo momento una adecuada política de copias de respaldo. En un proyecto distribuido, sin embargo, existe la ventaja de que otras personas disponen también del historial de revisiones.

## Conclusión

Git es rápido, usable e increíblemente flexible. Es genial para trabajar en proyectos altamente distribuidos en los que no se dispone de un repositorio central definitivo, sino de versiones publicadas estables que se pueden tratar como ramas o etiquetas particulares, aunque tampoco viene nada mal para uso propio. ■

### RECURSOS


[1] Sitio web oficial de Git: *http://git-scm.com/*

[2] Email de Linus Torvalds comentando las ventajas de los sistemas distribuidos: *http://lwn.net/Articles/246381*

[3] Descarga de Git: *http://www.kernel.org/pub/software/scm/git*

[4] Manual del usuario de Git: *http://www.kernel.org/pub/software/scm/git/docs/user-manual.html*

[5] Tutorial oficial de Git: *http://www.kernel.org/pub/software/scm/git/docs/gittutorial.html*

[6] Curso de Subversion-a-Git: *http://git-scm.org/course/svn.html*