

Tutorial de Git en Español

27 Nov 2011

Este tutorial es una traducción con algunos agregados de [Git Tutorial](#), de [Lars Vogel](#).

Para la traducción se han mejorado algunas cosas para adaptarlas al habla hispana. De todas maneras, el vocabulario usado puede resultarte raro. No estoy de acuerdo con traducir todas las palabras desde el inglés al español, porque muchas veces esto es imposible. Me he encontrado en libros de bases de Datos que me hablan de "comprometer" una transacción, cuando podrían directamente haber usado la palabra "commit". Por eso, tales palabras quedan en inglés, como corresponde. También puede resultarte raro que transforme esas palabras en verbos y las conjuge, como por ejemplo, la acción de hacer un "commit" la llamo "comitear". Si nunca lo escuchaste, no te preocupes, porque esta es la jerga que se utiliza en la industria.

- 1. [Git](#)
 - 1.1. [¿Qué es Git?](#)
 - 1.2. [Terminología importante](#)
 - 1.3. [Staging index](#)
- 2. [Instalación](#)
- 3. [Configuración](#)
 - 3.1. [Configuración del usuario](#)
 - 3.2. [Resaltado de colores](#)
 - 3.3. [Ignorar ciertos archivos](#)
- 4. [Empezando con Git](#)
 - 4.1. [Creando contenido](#)
 - 4.2. [Crear el repositorio y hacer un commit](#)
 - 4.3. [Ver las diferencias via diff y commitear los cambios](#)
 - 4.4. [Status, Diff y el log de Commit](#)
 - 4.5. [Corrección de mensajes de commit - git amend](#)
 - 4.6. [Eliminar archivos](#)
- 5. [Trabajando con repositorios externos](#)
 - 5.1. [Creando un repositorio Git externo](#)
 - 5.2. [Subir los cambios a otro repositorio - Push](#)
 - 5.3. [Add remote](#)




- 5.4. Mostrar los repositorios externos existentes
 - 5.5. Clonar tu repositorio
 - 5.6. Trayendo los cambios - Pull
- 6. Revertir cambios
- 7. Etiquetando en Git - Tagging
- 8. Branches y Merging
 - 8.1. Branches
 - 8.2. Merging
 - 8.3. Borrar un branch
- 9. Resolviendo conflictos después de un merge
- 10. Rebase
 - 10.1. Utilizando rebase para varios commits en un mismo branch
 - 10.2. rebase entre branches
 - 10.3. Buenas prácticas para el uso de rebase
- 11. Creando y aplicando parches
- 12. Definiendo un alias
- 13. Ignorando un archivo / directorio
- 14. Otros comandos útiles
- 15. Instalando un servidor Git
- 16. Repositorios remotos online
 - 16.1. Clonando repositorios remotos
 - 16.2. Agregando más repositorios remotos
 - 16.3. Operaciones remotas via HTTP y un proxy
- 17. Proveedores de hosting Git
 - 17.1. GitHub
 - 17.2. Bitbucket
- 18. Herramientas gráficas para Git
- 19. Obtené la versión original para Kindle
- 20. Preguntas y comentarios
- 21. Links y recursos útiles

1. Git

1.1. ¿qué es Git?

Git es un sistema de control de versiones distribuído (scvd) escrito en C. Un sistema de control de versiones permite la creación de una historia para una colección de archivos e incluye la funcionalidad para revertir la colección de archivos a otro estado. Otro estado



puede significar a otra colección diferente de archivos o contenido diferente de los archivos.

Podés, por ejemplo, cambiar la colección de archivos a un estado de 2 días atras, o podés cambiar entre los estados para características experimentales y problemas de producción.

Esta colección de archivos generalmente es llamada "código fuente". En un sistema de control de versiones distribuído todos tienen una copia completa del código fuente (incluyendo la historia completa del código fuente) y puede realizar operaciones referidas al control de versiones mediante esa copia local. El uso de un scvd no requiere un repositorio central.

Si haces cambios al código fuente, haces esos cambios relevantes para el control de versiones (los agregas al staging index) y después los agregas al repositorio (commit).

Git mantiene todas las versiones. Por lo tanto podés revertir a cualquier punto en la historia de tu código fuente usando Git.

Git realiza los commits a tu repositorio local y es posible sincronizar ese repositorio con otros (tal vez remotos) repositorios. Git te permite clonar los repositorio, por ejemplo, crear una copia exacta de un repositorio incluyendo la historia completa del código fuente. Los dueños de los repositorios pueden sincronizar los cambios via `push` (transfiere los cambios al repositorio remoto) o `pull` (obtiene los cambios desde un repositorio remoto).

Git soporta el concepto de `branching` , es decir, podés tener varias versiones diferentes de tu código fuente. Si querés desarrollar una nueva característica, podés abrir un `branch` en tu código fuente y hacer los cambios en este `branch` sin afectar el principal desarrollo de tu código.

Git puede ser usado desde la linea de comandos; esta manera será la descrita en este tutorial. También existen herramientas gráficas, por ejemplo Egit para el IDE Eclipse, pero estas herramientas no van a ser descritas en este tutorial.

1.2. Terminología importante

Término	Definición
Repositorio	Un repositorio contiene la historia, las diferentes versiones en el tiempo y todas los distintos branch y etiquetas. En Git, cada copia del repositorio es un repositorio completo. El repositorio te permite obtener revisiones en tu copia actual.
Branch	Un branch es una linea separada de código con su propia historia. Te es posible crear un nuevo branch de a partir de uno existente y cambiar el código independientemente de otros branches. Uno de los branches es el original (generalmente



	llamado master). El usuario selecciona un branch y trabaja en ese branch seleccionado, el cual es llamado copia actual (working copy). Seleccionar un branch es llamado "obtener un branch" (checkout a branch)
Etiquetas (Tags)	Un tag (una etiqueta) apunta a un cierto punto en el tiempo en un branch específico. Con un tag, es posible tener un punto con un tiempo al cual siempre sea posible revertir el código, por ejemplo, ponerle el tag "testing" al código 25.01.2009
Commit	Vos hacés un commit de los cambios a un repositorio. Esto crea una revisión nueva la cual puede ser obtenida después, por ejemplo si querés ver el código fuente de una versión anterior. Cada commit contiene el autor y el que realizó el commit (comiteador), siendo así posible identificar el origen del cambio. El autor y el comiteador pueden ser diferentes personas.
URL	Una URL en Git determina la ubicación de un repositorio.
Revisión	Representa una versión del código fuente. Git identifica revisiones con un id SHA1. Los id son de 160 bits de largo y son representados en hexadecimal. La última versión puede ser direccionada a través de "HEAD", la versión anterior mediante "HEAD-1" y así siguiendo.

1.3. Staging index (el índice)

Git requiere que los cambios sean marcados explícitamente para el próximo commit. Por ejemplo, si haces un cambio en un archivo y querés que y querés que ese cambio sea relevante para el próximo commit, es necesario que agregues el archivo a lo que se llama comunmnete "staging index" mediante el comando `git add` . Dicho índice será una imagen completa de los cambios.

Los nuevos archivos siempre deben ser explícitamente añadidos al índice. Para archivos que ya han sido comiteados podés usar la opción -a durante el commit.

2. Instalación

En Ubuntu podés instalar la herramienta de consola de Git con el siguiente comando:

```
sudo apt-get install git-core
```

Para otras distribuciones de Linux deberías consultar la documentación correspondiente a la distro.



Una versión de Git para Windows podés encontrarla en el sitio del proyecto msysgit. La URL es: <http://code.google.com/p/msysgit/> .

3. Configuración

Git te permite almacenar configuraciones globales en el archivo `.gitconfig`. Ese archivo debe ser ubicado en tu directorio de usuario (el home). Como mencioné anteriormente Git almacena el comiter y el autor para cada commit. Esta y otra información adicional puede ser almacenada en la configuración global.

A continuación veremos cómo configurar Git para utilizar un usuario y dirección de correo electrónico en particular, habilitar los colores y cómo decirle a Git que ignore ciertos archivos.

3.1. Configuración del usuario

Configurá tu usuario e email para Git mediante los siguientes comandos:

```
# Configura el usuario que será usado por git
# Obviamente deberías usar tu nombre
git config --global user.name "Santiago Basulto"
# Lo mismo para el correo electrónico
git config --global user.email "santiago.basulto-nospa
m@example.com"
# Configurar para que todos los cambios generen un pus
h por defecto
git config --global push.default "matching"
```

3.2. Resaltado de colores

Esto activará el resaltado de colores en la consola:

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

3.3. Ignorar ciertos archivos

Git puede ser configurado para que ignore ciertos archivos y directorios. Esta configuración es realizada mediante el archivo `.gitignore` . Este archivo puede estar en cualquier directorio y puede contener patrones para archivos. Por ejemplo, es posible decirle a git que ignore el directorio `bin` y todos los archivos que finalicen con la extensión `pyc` (archivos de python compilados) mediante el siguiente patrón en el archivo `.gitignore` en el directorio principal.

```
bin
*.pyc
```

Git también ofrece la configuración global `core.excludefile` para especificar de manera global qué patrones ignorar.

4. Empezando con Git



A continuación trataré de guiarte a través de una utilización típica de Git. Vas a crear algunos archivos, crear un repositorio Git local y commitear los archivos en este repositorio. Después clonaremos el repositorio y haremos push y pull entre los repositorios. Los comentarios (marcados con #) antes de cada comando explican las acciones específicas.

Abrí una consola para comenzar.

4.1. Creando contenido

Los comandos a continuación crean algunos archivos con algo de contenido y después son puestos bajo el control de versiones.

```
# Navegá hasta el home
cd ~/
# Crea un directorio
mkdir ~/repo01
# Ingresá a ese directorio
cd repo01
# Crea un nuevo directorio
mkdir datafiles
# Crea algunos archivos
touch test01
touch test02
touch test03
touch datafiles/data.txt
# Poné algo de texto adentro en el primer archivo
ls > test01
```

4.2. Crear el repositorio y hacer un commit

Cada repositorio Git es almacenado en la carpeta `.git` del directorio en el cual el repositorio ha sido creado. Este directorio contiene la historia completa del repositorio. El archivo `.git/config` contiene la configuración local del repositorio.

A continuación crearemos un repositorio Git, agregamos los archivos al índice del repositorio y commiteamos los cambios.

```
# Inicializamos el repositorio local Git
git init
# Agregamos todo (archivos y directorios) al repositorio
git add .
# Hacemos un commit al repositorio
git commit -m "Initial commit"
# Muestra el log (un historial)
git log
```

4.3. Ver las diferencias via diff y commitear los cambios

El comando `diff` de Git permite al usuario ver los cambios hechos. Para probar esto, hacé algunos cambios a un archivo y fijate qué te muestra `diff`

Después comitea los cambios al repositorio.



```
# Hací algunos cambios al archivo
echo "Este es un cambio" > test01
echo "y este es otro cambio" > test02

# Check the changes via the diff command
# Mirá los cambios con el comando diff
git diff

# Comitea los cambios, -a comitea los cambios de los a
rchivos modificados
# pero no agrega automaticamente nuevos archivos
git commit -a -m "Hay nuevos cambios"
```

4.3. Status, Diff y el log de Commit

A continuación utilizaremos comandos que te ayudarán a ver el estado actual de tu repositorio y un log de todos los commits.

```
# Hací algunos cambios al archivo
echo "Nuevo cambio" > test01
echo "y otro nuevo cambio" > test02

# Mirá el estado actual de tu repositorio
# (qué archivos han cambiado, son nuevos o eliminados)
git status

# Muestra las diferencias entre los archivos no comite
ados
# y el último commit en el branch actual
git diff

# Agrega los cambios al índice y comitea
git add . && git commit -m "Más cambioos - con un erro
r en el mensaje de commit"

# Muestra el histórico de commits en el branch actual
git log

# Muestra una linda vista gráfica de los cambios
gitk --all
```

4.5. Corrección de mensajes de commit - git amend

El comando `amend` hace posible cambiar el último mensaje de commit.

En el ejemplo anterior el mensaje de commit era incorrecto y tenía un error. A continuación lo corregimos mediante el parámetro `--amend`.

```
git commit --amend -m "Más cambios, ahora correctos"
```

4.6. Eliminar archivos

Si eliminas un archivo que está bajo el control de versiones, el comando `git add .` no tendrá en cuenta que has eliminado el archivo. Es neceasrio que uses el comando `git commit` con la bandera `-a` o el comando `git add` con la bandera `-A`.

```
# Crea un archivo y agregalo al control de versiones
touch nonsense.txt
```



```
git add . && git commit -m "un nuevo archivo ha sido c
reado"
# Elimina el archivo
rm nonsense.txt
# Proba hacer un commit de la forma normal -> no va a
andar
git add . && git commit -m "elimine el archivo, pero n
o anda"
# Ahora hace un commit con la bandera -a
git commit -a -m "El archivo nonsense.txt ahora si fue
eliminado del índice"
# Alternativamente podrías agregar los archivos borrados al índice mediante
git add -A .
git commit -m "El archivo nonsense.txt ahora si fue el
iminado del índice"
```

5. Trabajando con repositorios externos

5.1. Creando un repositorio Git externo

Ahora crearemos un repositorio Git remoto. Git te permite almacenar este repositorio remoto tanto en la red como localmente.

Un repositorio Git estandar es diferente de un repositorio remoto. Un repo estandar contiene el código fuente y el repositorio Git. Podés trabajar directamente en este directorio ya que este contiene una copia de todos los archivos.

Los repositorios remotos no contienen las copias de todos los archivos. Estos solo tienen los archivos del repositorio, la información perteneciente a Git. Para crear tal repositorio utilizá la bandera --bare.

Para simplificar los siguientes ejemplos, el repo será creado localmente en el filesystem.

```
# Dirigite al repo
cd ~/repo01
# hace un clone con la bandera --bare
git clone --bare . ../remote-repository.git

# Comprobá que el contenido es idéntico al directorio
.git en repo01
ls ~/remote-repository.git
```

5.2. Subir los cambios a otro repositorio - Push

Hacé algunos cambios y subilos desde tu primer repositorio al repositorio remoto mediante estos comandos

```
# Navegá hasta el primer repo
cd ~/repo01

# Hacé algunos cambios en los archivos
echo "Hola hola, prendé tu radio" > test01
echo "Chau chau, apagá la radio" > test02

# Comitea los cambios, -a comiteara los cambios para a
rchivos modificados
# pero no agrega automaticamente nuevos archivos
```




```
git commit -a -m "Algunos cambios"
```

```
# Subí los cambios
git push ../remote-repository.git
```

5.3. Add remote

Siempre es posible subir cambios a un repositorio Git mediante su URL completa. Pero también es posible agregar un "apodo" a un repositorio mediante el comando `git remote add . origin` es un nombre especial que es normalmente usado de forma automática, si clonas un repositorio Git. `origin` indica el repositorio original desde el cual has empezado. Ya que nosotros comenzamos desde el principio, este nombre está aun disponible.

```
# agregá ../remote-repository.git con el nombre origin
git remote add origin ../remote-repository.git

# Otros cambios
echo "Agregué un repo remoto" > test02
# Commit
git commit -a -m "Test para el nuevo repo remoto origin"

# sube automaticamente a origin
git push origin
```

5.4. Mostrar los repositorios externos existentes

Para ver las definiciones existentes de los repositorios remotos, utilizá el siguiente comando:

```
# Muestra los repositorios externos existentes
git remote
```

5.5. Clonar tu repositorio

Crea un nuevo repositorio en un nuevo directorio con los siguientes comandos:

```
# Navegá al home
cd ~
# Crea un nuevo directorio
mkdir repo02

# Navegá al nuevo directorio
cd ~/repo02
# clonalo
git clone ../remote-repository.git .
```

5.6. Trayendo los cambios - Pull

Pull te permite obtener los últimos cambios de otro repositorio. En tu segundo repositorio, hacé unos cambios, subilos (push) a tu repositorio remoto y después traelos (con un pull) desde el primer repo.

```
# Navegá hacia el home
cd ~
```



```
# Navegá al segundo repo
cd ~/repo02
# Hacé algunos cambios
echo "Un cambio" > test01
# Commit
git commit -a -m "Un cambio"

# Subí los cambios al repo remoto
# Origin es automaticamente mantenido ya que clonamos desde este repositorio
git push origin

# Navegá al primer repositorio y traete los cambios
cd ~/repo01
git pull ../remote-repository.git/
# Reivés los cambios
less test01
```

6. Revertir cambios

Si creas archivos en tu copia actual que no quisieses comitear, es posible desecharlos.

```
# Crea un nuevo archivo con algo de contenido
touch test04
echo "esto no sirve" > test04

# Hacé un dry-run para ver que ocurriría desde el último cambio
# -n es lo mismo que --dry-run
git clean -n

# Ahora lo borra
git clean -f
```

You can check out older revisions of your source code via the commit ID. The commit ID is shown if you enter the git log command. It is displayed behind the commit word.

Es posible obtener (mediante un check out) revisiones anteriores de tu código mediante el ID de commit. El ID de commit se muestra si ingresás el comando `git log`. Es mostrado después de la palabra commit.

```
# Navegá al home
cd ~/repo01
# obtene un log
git log

# copiá uno commit viejo y obtené las revisiones anteriores
git checkout nombre_del_commit
```

Si no agregaste los cambios al índice, también es posible revertir los cambios directamente.

```
# algún cambio sin sentido
echo "sin sentido" > test01
# no ha sido agregado al índice, por lo tanto
# simplemente podemos obtener la versión anterior
git checkout test01
# Compró el contenido
cat test01
```



```
# Otro cambio sin sentido
echo "otro cambio sin sentido" > test01
# Agregamos el archivo al índice
git add test01
# Recuperamos el archivo en el índice
git reset HEAD test01
# Obtenemos la versión vieja desde el índice
git checkout test01
```

También es posible revertir los commits mediante el siguiente comando:

```
# Revertir un commit
git revert commit_name
```

Si borraste un archivo pero no has hecho el cambio en el índice, o comiteado el cambio, es posible volver a obtener ese archivo.

```
# borramos el archivo
rm test01
# revertimos la eliminación
git checkout test01
```

Si agregaste el archivo al índice pero no querés comitearlo, es posible eliminarlo del índice mediante el comando `git reset .`

```
# creamos un archivo
touch incorrecto.txt
# lo agregamos al índice accidentalmente
git add .
# lo borramos del índice
git reset incorrecto.txt
# borramos el archivo
rm incorrect.txt
```

Si borraste un directorio y no has comiteado los cambios es posible recuperarlo mediante el siguiente comando:

```
git checkout HEAD -- directorio\_a\_recuperar
```

7. Etiquetando en Git - Tagging

Git tiene la opción de etiquetar (taguear) ciertas versiones para encontrarlas de forma más fácil en el futuro. Lo más común es utilizarlo para etiquetar una cierta versión que ha sido largada en producción.

Es posible listar los tags disponibles mediante este comando:

```
git tag
```

Podés crear un nuevo tag de la siguiente manera. Con el parámetro `-m` especificás la descripción del tag.

```
git tag version1.6 -m 'version 1.6'
```

Si querés acceder al código asociado con ese tag:

```
git checkout tag\name
```

8. Branches y Merging

8.1. Branches y Merging

Git te permite crear branches. Branches, el plural de branch, son copias independientes del código fuente que pueden ser cambiadas independientemente de las otras. El branch por defecto es llamado *master*. Git te permite crear branches de forma muy rápida y barato en cuanto a recursos. Es recomendable utilizar branches frecuentemente.

El siguiente comando lista todos los branches disponibles localmente. El branch actualmente activo es marcado con un asterisco *

```
git branch
```

Si querés ver todos los branches (incluyendo aquellos remotos), utilizá el siguiente comando.

```
git branch -a
```

Es posible crear un nuevo branch de la siguiente manera.

```
# Sintaxis: git branch <nombre> <hash>
# <hash> es opcional
# si no es especificado se utiliza el del último commit
# si es especificado se utiliza el del commit correspondiente
git branch testing
# cambiá al nuevo branch
git checkout testing
# algunos cambios
echo "Algo nuevo en este branch" > test01
git commit -a -m "algo nuevo"
# Volvé al branch master
git checkout master
# comprobá que el contenido de test01 es el anterior
cat test01
```

8.2. Merging

Merge te permite combinar los cambios de dos branches. Merge realiza el comunmente llamado *merge de tres versiones* entre la última imagen de dos branches, basado en el más reciente ancestro en común de ambos.

Como resultado se obtiene una nueva imagen. Es posible hacer un merge entre los cambios de un branch y el actualmente activo mediante el siguiente comando.



```
# sintaxis: git merge <nombre-del-branch>
git merge testing
```

Si un conflicto ocurre, Git marcará el conflicto en el archivo y el programador tiene que resolver el conflicto manualmente. Después de resolverlo, puede agregar el archivo al índice y comitear el cambio.

8.3. Borrar un branch

Para eliminar un branch que ya no es necesario, podés utilizar el siguiente comando.

```
# borra el branch 'testing'
git branch -d testing
# comprobá que el branch fue borrado
git branch
```

9. Resolviendo conflictos después de un merge

Un conflicto ocurre si dos personas han modificado el mismo contenido y Git no puede determinar automaticamente como deberían ser aplicados ambos cambios.

Git requiere que los conflictos sean resueltos manualmente. En esta sección; primero crearemos un conflicto y después lo resolveremos y aplicaremos el cambio al repositorio.

De la siguiente manera creamos un conflicto.

```
# navegá al primer directorio
cd ~/repo01
# algunos cambios
touch conflictivo.txt
echo "Cambio en el primer repo" > conflictivo.txt
# agregamos al índice y comiteamos
git add . && git commit -a -m "Creamos el conflicto 1"

# Navegamos al segundo directorio
cd ~/repo02
# más cambios
touch conflictivo.txt
echo "Cambio en el segundo repo" > conflictivo.txt
# agregamos al índice y comiteamos
git add . && git commit -a -m "Creamos el conflicto 2"
# Hacemos un push al master
git push

# ahora tratá de hacer un push desde el primer directorio
# navegá al primer directorio
cd ~/repo01
# Tratá de hacer un push -> ocurrirá un error
git push
# obtené los cambios
git pull origin master
```

Git marca el conflicto en el archivo afectado (conflictivo.txt). El archivo debería tener el siguiente contenido.



```
<<<<<<< HEAD
Cambio en el primer repo
=====
Cambio en el segundo repo
>>>>>>> b29196692f5ebfd10d8a9ca1911c8b08127c85f8
```

La parte superior (sobre la linea ====) contiene el contenido de tu repositorio, y la parte inferior es la que tiene el contenido del repositorio remoto. Ahora podés editar el archivo manualmente y después comitear los cambios. Alternativamente, podrías usar el comando `git mergetool`. `git mergetool` lanza una herramienta configurable que muestra los cambios en una pantalla dividida.

```
# podes editar el archivo manualmente o usar
git mergetool
# Te pedirá que selecciones qué herramienta para
# resolver el conflicto deseas utilizar
# por ejemplo en ubuntu podés usar la herramienta "meld"
# después de resolver los conflictos manualmente, los comi
teamos
git commit -m "conflictos resueltos"
```

10. Rebase

10.1 Utilizando rebase para varios commits en un mismo branch

El comando `rebase` te permite combinar varios commits en uno único. Esto es útil ya que permite al usuario reescribir un poco de la historia de los commits (haciendo algo de limpieza) antes de hacer un push de los cambios a un repositorio remoto.

A continuación crearemos varios commits que serán combinados más adelante.

```
# Crea un nuevo archivo
touch rebase.txt

# agregamos al índice y comiteamos
git add . && git commit -m "rebase.txt añadido al índice"

# Algunos cambios tontos
echo "contenido" >> rebase.txt
git add . && git commit -m "agregué contenido"
echo " más contenido" >> rebase.txt
git add . && git commit -m "agregué más contenido"
echo " más contenido" >> rebase.txt
git add . && git commit -m "agregué más contenido"
echo " más contenido" >> rebase.txt
git add . && git commit -m "agregué más contenido"
echo " más contenido" >> rebase.txt
git add . && git commit -m "agregué más contenido"

# mirá el log (largo, no?)
git log
```




Ahora combinaremos los últimos siete commits. Podés hacer esto interactivamente mediante el siguiente comando.

```
git rebase -i HEAD~7
```

Esto abre tu editor preferido y te deja editar el mensaje de commit o compactar (mediante `squash - s`) / arreglar (mediante `fixup - f`) el commit con el último.

La opción de compactar (squash) combina los mensajes de commit mientras que arreglar (fixup) ignora los mensajes.

10.2 rebase entre branches

También podés usar Git para hacer un rebase entre dos branches. Como describimos anteriormente, el comando `merge` combina los cambios de dos branches. Rebase toma los cambios de un branch, crea un patch y lo aplica al otro branch.

El resultado final del código fuente es el mismo que con `merge` pero la historia de commit es más limpia. La historia parece ser lineal.

```
# Crea un nuevo branch
git branch testing
# checkout del branch
git checkout testing
# Hacemos algunos cambios
echo "Haremos un rebase con el master" > test01
# hacemos un commit al branch testing
git commit -a -m "Algo nuevo en el branch"
# Hacemos un rebase con el master
git rebase master
```

10.1 Buenas prácticas para el uso de rebase

Siempre deberías chequear tu historial local antes de hacer un push con tus nuevos cambios a otro repositorio Git.

Git te permite hacer commits locales. Esta característica es frecuentemente utilizada para tener puntos a los cuales volver si algo sale mal más adelante durante el desarrollo de nuevas funcionalidades. Si haces esto, antes de hacer un push deberías mirar tu historial local y validar si estos commits son relevantes para otros desarrolladores.

Si todos esos commits son parte de la implementación de una funcionalidad en particular, lo más probable es que quieras resumirlos en un único commit antes de hacer el push.

El rebase interactivo consiste básicamente en reescribir la historia. Es seguro hacer esto siempre y cuando los commits no han sido subidos (mediante un puhs) a otro repositorio. Esto significa que los commits solo deben ser reescritos mientras no se haya hecho un push.



Si reescribis algo y después haces un push de un commit que ya está presente en otros repositorios Git, en realidad parecerá como si hubieses implementado algo que alguien ya implementó en el pasado.

11. Creando y aplicando parches

Un parche (patch) es un archivo de texto que contiene cambios para el código fuente. Este archivo puede ser enviado a alguna persona y esta persona puede usar dicho archivo para aplicar los cambios a su repositorio.

Los siguientes comandos crean un branch, hacen algunos cambios, crean un parche y aplican el parche al master.

```
# creamos un nuevo branch
git branch mybranch
# usamos este branch
git checkout mybranch
# hacemos algunos cambios
touch test05
# cambiamos el contenido de un archivo
echo "New content for test01" >test01
# comiteamos esto al branch
git add .
git commit -a -m "primer commit en el branch"

# creamos un parche -> sintaxis: git format-patch mast
er
git format-patch origin/master
# se crea el archivo 0001-primer-commit-en-el-branch.p
atch

# obtenemos el master
git checkout master

# aplicamos el parche
git apply 0001-primer-commit-en-el-branch.patch
# hacemos un commit normal en el master
git add .
git commit -a -m "parche aplicado"

# borramos el archivo
rm 0001-primer-commit-en-el-branch.patch
```

12. Definiendo un alias

Un alias en Git te permite crear tus propios comandos Git. Por ejemplo, podés definir un alias que sea una forma más corta para tu comando favorito o podés combinar varios comandos con un alias.

Por ejemplo, a continuación definimos el comando `git add-commit` que combina `git add . -A` y `git commit -m .` Después de definir este comando, podés usarlo de la siguiente manera: `git add-commit -m "mensaje"`

```
git config --global alias.add-commit '!git add . -A && git
commit'
```

Unfortunately, defining an alias is at the time of writing this not completely supported in msysGit. You can do single aliases, e.g. ca

for ca = commit -a) but you can't do ones beginning with ! .

Desafortunadamente, la funcionalidad de los alias al momento de finalizar estas lineas no está completamente soportado en *msysGit*. Podés hacer alias simples (para un solo comando), por ejemplo utilizar el alias `ca` para `commit -a` pero no podes hacer los que comienzan con un `!`.

13. Ignorando un archivo / directorio

Algunas veces deseamos que algunos archivos o directorios no sean incluídos en tu repositorio Git. Si los agregas al archivo *.gitignore*, Git comenzará a ignorarlose desde ese momento. Esto quiere decir que no los eliminará del repo. Por lo tanto, la última versión seguirá estando en el índice. Para que sean eliminados del índice podés usar:

```
# eliminamos el directorio .metadata del repo
git rm -r --cached .metadata
# eliminamos el archivo test.txt del repo
git rm --cached test.txt
```

Esto no quiere decir que sean eliminados del historial de commits. Si realmente desees que sean borrados del historial, consultá el comando `git filter-branch` que te permite reescribir el historial de commits.

14. Otros comandos útiles

La siguiente lista contiene algunos comandos de Git que son útiles para el trabajo diario.

Comando	Descripción
<code>git blame "nombre-archivo"</code>	Muestra quién creó o modificó el archivo
<code>git checkout -b mi-branch master~1</code>	Crea un branch basándose en el master sin incluir el último commit

15. Instalando un servidor Git

Como describí anteriormente no es necesario un servidor. Podés utilizar simplemente el filesystem o un proveedor público de Git, como Github o Bitbucket. De todas fromas, algunas veces es conveiente tener nuestro propio servidor. Instalarlo en Ubuntu es relativamente sencillo.

Primero comprobá que tenés instalado *ssh*.

```
apt-get install ssh
```

Si no instalaste Git, es necesario que lo hagas:

```
sudo apt-get install git-core
```

Crea un nuevo usuario para Git.

```
sudo adduser git
```

Ahora logueate con el usuario de Git y crea un repositorio.

```
# logueate en el servidor
# para probar podes usar localhost
ssh git@DIRECCION_IP_DEL_SERVER

# Crea el repositorio
git init --bare example.git
```

Ahora podés comitear al repositorio remoto.

```
mkdir gitexample
cd gitexample
git init
touch README
git add README
git commit -m 'primer commit'
git remote add origin git@DIRECCION_IP_DEL_SERVER:exam
ple.git
git push origin master
```

16. Repositorios remotos online

16.1 Clonando repositorios remotos

Git también soporta operaciones remotas. Git soporta varios tipos de protocolos de transporte; el protocolo nativo de Git es llamado *git*.

A continuación clonaremos un repositorio existente mediante el protocolo git.

```
git clone git@github.com:vogella/gitbook.git
```

Alternativamente podrías clonar el mismo repositorio mediante el protocolo http.

```
# clonamos a través de HTTP
git clone http://vogella@github.com/vogella/gitbook.git
```

16.2 Agregando más repositorios remotos

Si clonas un repositorio remoto, el repositorio original será llamado automáticamente *origin*.

Podés hacer push con los cambios a este repositorio *origin* mediante *git push origin*. Por supuesto, para hacer un push al repo reomoto tenés que contar con los permisos de escritura para tal repo.



También podés agregar más repositorios remotos a tu repositorio mediante el comando `git remote add [nombre] [url_repo]` . Por ejemplo si clonaste el repositorio de arriba con el protocolo git, podés agregar tambien el protocolo `https` de la siguiente manera:

```
// agregamos el protocolo https
git remote add githttp https://vogella@github.com/vogella/gitbook.git
```

16.3 Operaciones remotas via HTTP y un proxy

Es posible utilizar el protocolo HTTP para clonar repositorios Git. Es especialmente útil si tu firewall bloquea todo excepto HTTP.

Git también provee soporte para acceso http mediante un servidor proxy. El siguiente comando Git podría por ejemplo, clonar un repositorio mediante http y un proxy. Podés incluir la variable proxy en general para todas las aplicaciones o solamente para Git.

Este ejemplo usa variables de entorno (enviroment variables).

```
# Linux
export http_proxy=http://proxy:8080
# Windows
# Set http_proxy=http://proxy:8080
git clone http://dev.eclipse.org/git/org.eclipse.jface/org.eclipse.jface.snippets.git
# Push al origin usando http
git push origin
```

Este ejemplo utiliza la configuracion de Git

```
# Set proxy for git globally
# proxy para git global
git config --global http.proxy http://proxy:8080
# para comprobar las configuraciones del proxy
git config --get http.proxy
# por si necesitas borrar la configuración del proxy
git config --global --unset http.proxy
```

17. Proveedores de hosting Git

En lugar de crear tu propio servidor, también es posible utilizar un servicio de hosting. Los proveedores de hosting de Git más populares son [Github](#) y [Bitbucket](#). Ambos ofrecen almacenamiento gratuito con ciertas limitaciones.

17.1 GitHub

Git Hub es gratuito para todos los repositorios públicos, o sea que si querés tener repositorios privados que solo son visibles por las personas que vos elegis, tenés que pagarle a GitHub una tarifa mensual.

GitHub requiere que crees lo que se llama una *key ssh* (llave SSH). Una descripción para crear dicha llave en Ubuntu puede ser encontrada en la sección de *ssh key creation* en el sitio de Ubuntu. Para windows remitite a *ssh key generation* de *msysgit*.