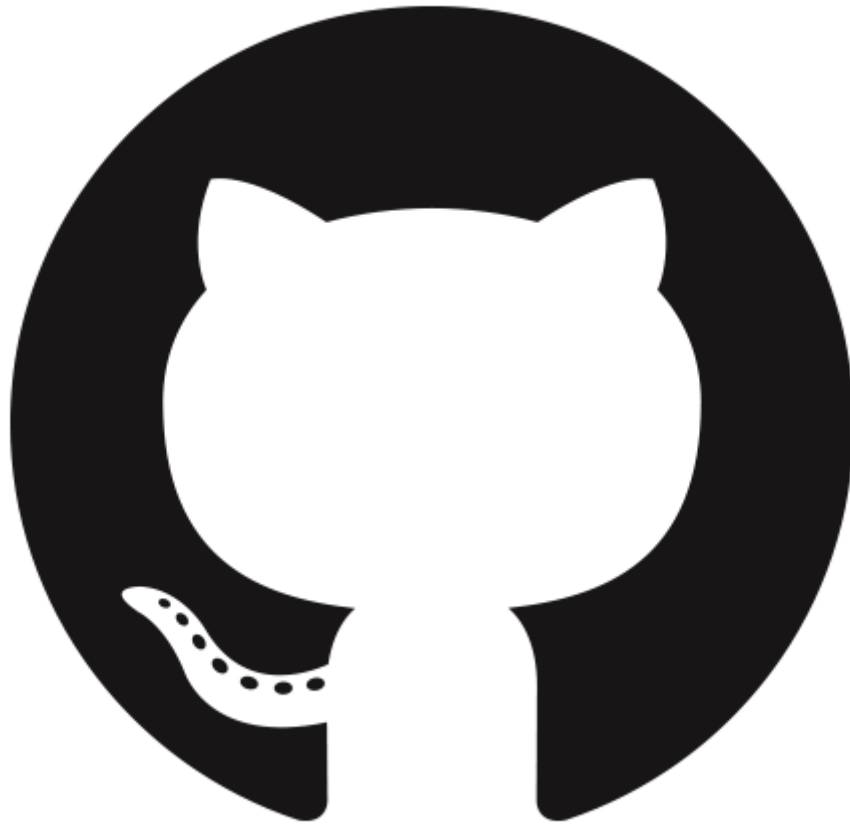


# TUTORIAL GIT



**github**  
SOCIAL CODING

## Contenido

- Aspectos básicos de Git
- Uso básico
- Uso avanzado
- Ramas
- Administración de repositorios
- Flujo de trabajo con Git (git flow)
- Github
- Referencias

# Aspectos básicos de Git

## Instalación

### Instalando en Linux

Si quieres instalar Git en Linux a través de un instalador binario, en general puedes hacerlo a través de la herramienta básica de gestión de paquetes que trae tu distribución. Si estás en Fedora, puedes usar yum:

```
1$ yum install git-core
```

O si estás en una distribución basada en Debian como Ubuntu, prueba con apt-get:

```
1$ apt-get install git
```

### Instalando en Windows

Instalar Git en Windows es muy fácil. El proyecto msysGit tiene uno de los procesos de instalación más sencillos. Simplemente descarga el archivo exe del instalador desde la página de GitHub, y ejecútalo:

<http://msysgit.github.com/>

Una vez instalado, tendrás tanto la versión de línea de comandos (incluido un cliente SSH que nos será útil más adelante) como la interfaz gráfica de usuario estándar. Se recomienda no modificar las opciones que trae por defecto el instalador.

### Instalando en MacOS

En MacOS se recomienda tener instalada la herramienta [homebrew](#). Después, es tan fácil como ejecutar:

```
1$ brew install git
```

## Configuración

### Tu identidad

Lo primero que deberías hacer cuando instalas Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios (commits) en Git usan esta información, y es introducida de manera inmutable en los commits que envías:

```
1$ git config --global user.name "John Doe"
2$ git config --global user.email johndoe@example.com
```

También se recomienda configurar el siguiente parámetro:

```
1$ git config --global push.default simple
```

## Bash Completion

Bash completion es una utilidad que permite a bash completar órdenes y parámetros. Por defecto suele venir desactivada en Ubuntu y es necesario modificar el archivo `$HOME/.bashrc` para poder activarla. Simplemente hay que descomentar las líneas que lo activan.

# Uso básico de Git

## Crear un proyecto

Crear un programa "Hola Mundo"

Creemos un directorio donde colocar el código

```
1$ mkdir curso-de-git
2$ cd curso-de-git
```

Creemos un fichero `hola.php` que muestre Hola Mundo.

```
1<?php
2echo "Hola Mundo\n";
```

## Crear el repositorio

Para crear un nuevo repositorio se usa la orden `git init`

```
1$ git init
2Initialized empty Git repository in /home/cc0gobas/git/curso-de-git/.git/
```

## Añadir la aplicación

Vamos a almacenar el archivo que hemos creado en el repositorio para poder trabajar, después explicaremos para qué sirve cada orden.

```
1$ git add hola.php
2$ git commit -m "Creación del proyecto"
3[master (root-commit) e19f2c1] Creación del proyecto
4 1 file changed, 2 insertions(+)
5 create mode 100644 hola.php
```

## Comprobar el estado del repositorio

Con la orden `git status` podemos ver en qué estado se encuentran los archivos de nuestro repositorio.

```
1$ git status
2# On branch master
```

```
3nothing to commit (working directory clean)
```

**Si modificamos el archivo `hola.php`:**

```
1<?php
2@print "Hola {$argv[1]}\n";
```

**Y volvemos a comprobar el estado del repositorio:**

```
1$ git status
2# On branch master
3# Changes not staged for commit:
4#   (use "git add <file>..." to update what will be committed)
5#   (use "git checkout -- <file>..." to discard changes in working directory)
6#
7#   modified:   hola.php
8#
9no changes added to commit (use "git add" and/or "git commit -a")
```

## Añadir cambios

**Con la orden `git add` indicamos a git que prepare los cambios para que sean almacenados.**

```
1$ git add hola.php
2$ git status
3# On branch master
4# Changes to be committed:
5#   (use "git reset HEAD <file>..." to unstage)
6#
7#   modified:   hola.php
8#
```

## Confirmar los cambios

**Con la orden `git commit` confirmamos los cambios definitivamente, lo que hace que se guarden permanentemente en nuestro repositorio.**

```
1$ git commit -m "Parametrización del programa"
2[master efc252e] Parametrización del programa
3 1 file changed, 1 insertion(+), 1 deletion(-)
4$ git status
5# On branch master
6nothing to commit (working directory clean)
```

## Diferencias entre workdir y staging.

**Modificamos nuestra aplicación para que soporte un parámetro por defecto y añadimos los cambios.**

```
1<?php
2$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
3@print "Hola, {$nombre}\n";
```

Este vez añadimos los cambios a la fase de staging pero sin confirmarlos (commit).

```
1git add hola.php
```

Volvemos a modificar el programa para indicar con un comentario lo que hemos hecho.

```
1<?php
2// El nombre por defecto es Mundo
3$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
4@print "Hola, {$nombre}\n";
```

Y vemos el estado en el que está el repositorio

```
1$ git status
2# On branch master
3# Changes to be committed:
4#   (use "git reset HEAD <file>..." to unstage)
5#
6#   modified:   hola.php
7#
8# Changes not staged for commit:
9#   (use "git add <file>..." to update what will be committed)
10#   (use "git checkout -- <file>..." to discard changes in working directory)
11#
12#   modified:   hola.php
13#
```

Podemos ver como aparecen el archivo hola.php dos veces. El primero está preparado para ser confirmado y está almacenado en la zona de staging. El segundo indica que el directorio hola.php está modificado otra vez en la zona de trabajo (workdir).

## Warning

Si volvieramos a hacer un `git add hola.php` sobreescribiríamos los cambios previos que había en la zona de staging.

Almacenamos los cambios por separado:

```
1$ git commit -m "Se añade un parámetro por defecto"
2[master 3283e0d] Se añade un parámetro por defecto
3 1 file changed, 2 insertions(+), 1 deletion(-)
4$ git status
5# On branch master
6# Changes not staged for commit:
7#   (use "git add <file>..." to update what will be committed)
8#   (use "git checkout -- <file>..." to discard changes in working directory)
```

```
9#
10#   modified:   hola.php
11#
12no changes added to commit (use "git add" and/or "git commit -a")
13$ git add .
14$ git status
15# On branch master
16# Changes to be committed:
17#   (use "git reset HEAD <file>..." to unstage)
18#
19#   modified:   hola.php
20#
21$ git commit -m "Se añade un comentario al cambio del valor por defecto"
22[master fd4da94] Se añade un comentario al cambio del valor por defecto
23 1 file changed, 1 insertion(+)
```

---

## Info

El valor "." despues de `git add` indica que se añadan todos los archivos de forma recursiva.

---

## Warning

Cuidado cuando uses `git add .` asegúrate de que no estás añadiendo archivos que no quieres añadir.

## Ignorando archivos

La orden `git add .` O `git add nombre_directorio` es muy cómoda, ya que nos permite añadir todos los archivos del proyecto o todos los contenidos en un directorio y sus subdirectorios. Es mucho más rápido que tener que ir añadiéndolos uno por uno. El problema es que, si no se tiene cuidado, se puede terminar por añadir archivos innecesarios o con información sensible.

Por lo general se debe evitar añadir archivos que se hayan generado como producto de la compilación del proyecto, los que generen los entornos de desarrollo (archivos de configuración y temporales) y aquellos que contentan información sensible, como contraseñas o tokens de autenticación. Por ejemplo, en un proyecto de C/C++, los archivos objeto no deben incluirse, solo los que contengan código fuente y los make que los generen.

Para indicarle a git que debe ignorar un archivo, se puede crear un fichero llamado `.gitignore`, bien en la raíz del proyecto o en los subdirectorios que queramos. Dicho fichero puede contener patrones, uno en cada línea, que especiquen qué archivos deben ignorarse. El formato es el siguiente:

```
1# .gitignore
```

```

1 dir1/                # ignora todo lo que contenga el directorio dir1
2 !dir1/info.txt      # El operador ! excluye del ignore a dir1/info.txt (sí se guardaría)
3 dir2/*.txt          # ignora todos los archivos txt que hay en el directorio dir2
4 dir3/**/*.txt       # ignora todos los archivos txt que hay en el dir3 y sus
5 subdirectorios
6 *.o                 # ignora todos los archivos con extensión .o en todos los directorios

```

Cada tipo de proyecto genera sus ficheros temporales, así que para cada proyecto hay un `.gitignore` apropiado. Existen repositorios que ya tienen creadas plantillas. Podéis encontrar uno en <https://github.com/github/gitignore>

## Ignorando archivos globalmente

Si bien, los archivos que hemos metido en `.gitignore`, deben ser aquellos ficheros temporales o de configuración que se pueden crear durante las fases de compilación o ejecución del programa, en ocasiones habrá otros ficheros que tampoco debemos introducir en el repositorio y que son recurrentes en todos los proyectos. En dicho caso, es más útil tener un `gitignore` que sea global a todos nuestros proyectos. Esta configuración sería complementaria a la que ya tenemos. Ejemplos de lo que se puede ignorar de forma global son los ficheros temporales del sistema operativo (\*~, .nfs\*) y los que generan los entornos de desarrollo.

Para indicar a git que queremos tener un fichero de `gitignore` global, tenemos que configurarlo con la siguiente orden:

```
1 git config --global core.excludesfile $HOME/.gitignore_global
```

Ahora podemos crear un archivo llamado `.gitignore_global` en la raíz de nuestra cuenta con este contenido:

```

1 # Compiled source #
2 #####
3 *.com
4 *.class
5 *.dll
6 *.exe
7 *.o
8 *.so
9
10 # Packages #
11 #####
12 # it's better to unpack these files and commit the raw source
13 # git has its own built in compression methods
14 *.7z
15 *.dmg
16 *.gz
17 *.iso
18 *.jar

```



```
19*.rar
20*.tar
21*.zip
22
23# Logs and databases #
24#####
25*.log
26*.sql
27*.sqlite
28
29# OS generated files #
30#####
31.DS_Store
32.DS_Store?
33._*
34.Spotlight-V100
35.Trashes
36ehthumbs.db
37Thumbs.db
38*~
39*.swp
40
41# IDEs #
42#####
43.idea
44.settings/
45.classpath
46.project
```

## Trabajando con el historial

### Observando los cambios

Con la orden `git log` podemos ver todos los cambios que hemos hecho:

```
1$ git log
2commit fd4da946326f8e8b24e89282ad25a71721bf40f6
3Author: Sergio Gómez <sergio@uco.es>
4Date:    Sun Jun 16 12:51:01 2013 +0200
5
6    Se añade un comentario al cambio del valor por defecto
7
8commit 3283e0d306c8d42d55ffcb64e456f10510df8177
9Author: Sergio Gómez <sergio@uco.es>
10Date:    Sun Jun 16 12:50:00 2013 +0200
11
```

```

12    Se añade un parámetro por defecto
13
14commit efc252e11939351505a426a6e1aa5bb7dc1dd7c0
15Author: Sergio Gómez <sergio@uco.es>
16Date:   Sun Jun 16 12:13:26 2013 +0200
17
18    Parametrización del programa
19
20commit e19f2c1701069d9d1159e9ee21acaalbbc47d264
21Author: Sergio Gómez <sergio@uco.es>
22Date:   Sun Jun 16 11:55:23 2013 +0200
23
24    Creación del proyecto

```

También es posible ver versiones abreviadas o limitadas, dependiendo de los parámetros:

```

1$ git log --oneline
2fd4da94 Se añade un comentario al cambio del valor por defecto
33283e0d Se añade un parámetro por defecto
4efc252e Parametrización del programa
5e19f2c1 Creación del proyecto
6git log --oneline --max-count=2
7git log --oneline --since='5 minutes ago'
8git log --oneline --until='5 minutes ago'
9git log --oneline --author=sergio
10git log --oneline --all

```

Una versión muy útil de `git log` es la siguiente, pues nos permite ver en que lugares está master y HEAD, entre otras cosas:

```

$ git log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
1* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (HEAD,
2Gómez]
3* 3283e0d 2013-06-16 | Se añade un parámetro por defecto [Sergio Gómez]
4* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
5* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]

```

## Crear alias

Como estas órdenes son demasiado largas, Git nos permite crear alias para crear nuevas órdenes parametrizadas. Para ello podemos configurar nuestro entorno con la orden `git config` de la siguiente manera:

```

1git config --global alias.hist "log --pretty=format:'%h %ad | %s%d [%an]' --graph --
date=short"

```

---

## Example

Puedes configurar incluso alias para abreviar comandos. Algunos ejemplos de alias útiles:

```
1git config --global alias.br branch
2git config --global alias.co checkout
3git config --global alias.ci commit
4git config --global alias.st "status -u"
5git config --global alias.cane "commit --amend --no-edit"
```

## Recuperando versiones anteriores

Cada cambio es etiquetado por un hash, para poder regresar a ese momento del estado del proyecto se usa la orden `git checkout`.

```
1$ git checkout e19f2c1
2Note: checking out 'e19f2c1'.
3
4You are in 'detached HEAD' state. You can look around, make experimental
5changes and commit them, and you can discard any commits you make in this
6state without impacting any branches by performing another checkout.
7
8If you want to create a new branch to retain commits you create, you may
9do so (now or later) by using -b with the checkout command again.
10Example:
11
12  git checkout -b new_branch_name
13
14HEAD is now at e19f2c1... Creación del proyecto
15$ cat hola.php
16<?php
17echo "Hello, World\n";
```

El aviso que nos sale nos indica que estamos en un estado donde no trabajamos en ninguna rama concreta. Eso significa que los cambios que hagamos podrían "perdersse" porque si no son guardados en una nueva rama, en principio no podríamos volver a recuperarlos. Hay que pensar que Git es como un árbol donde un nodo tiene información de su nodo padre, no de sus nodos hijos, con lo que siempre necesitaríamos información de dónde se encuentran los nodos finales o de otra manera no podríamos acceder a ellos.

Volver a la última versión de la rama master.

Usamos `git checkout` indicando el nombre de la rama:

```
1$ git checkout master
2Previous HEAD position was e19f2c1... Creación del proyecto
```

## Etiquetando versiones

Para poder recuperar versiones concretas en la historia del repositorio, podemos etiquetarlas, lo cual es más facil que usar un hash. Para eso usaremos la orden `git tag`.

```
1$ git tag v1
```

Ahora vamos a etiquetar la versión inmediatamente anterior como v1-beta. Para ello podemos usar los modificadores `^` o `~` que nos llevarán a un ancestro determinado. Las siguientes dos órdenes son equivalentes:

```
1$ git checkout v1^
2$ git checkout v1~1
3$ git tag v1-beta
```

Si ejecutamos la orden sin parámetros nos mostrará todas las etiquetas existentes.

```
1$ git tag
2v1
3v1-beta
```

Y para verlas en el historial:

```
$ git hist master --all
1 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag:
2 Gómez]
3 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (HEAD, tag: v1-beta) [Sergi
4 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
5 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

## Borrar etiquetas

Para borrar etiquetas:

```
1git tag -d nombre_etiqueta
```

## Visualizar cambios

Para ver los cambios que se han realizado en el código usamos la orden `git diff`. La orden sin especificar nada más, mostrará los cambios que no han sido añadidos aún, es decir, todos los cambios que se han hecho antes de usar la orden `git add`. Después se puede indicar un parámetro y dará los cambios entre la versión indicada y el estado actual. O para comparar dos versiones entre sí, se indica la más antigua y la más nueva. Ejemplo:

```
1$ git diff v1-beta v1
2diff --git a/hola.php b/hola.php
3index a31e01f..25a35c0 100644
4--- a/hola.php
```

```
5+++ b/hola.php
6@@ -1,3 +1,4 @@
7 <?php
8+// El nombre por defecto es Mundo
9 $nombre = isset($argv[1]) ? $argv[1] : "Mundo";
10 @print "Hola, {$nombre}\n";
```

# Uso avanzado de Git

## Deshacer cambios

Deshaciendo cambios antes de la fase de staging.

Volvemos a la rama máster y vamos a modificar el comentario que pusimos:

```
1$ git checkout master
2Previous HEAD position was 3283e0d... Se añade un parámetro por defecto
3Switched to branch 'master'
```

Modificamos hola.php de la siguiente manera:

```
1<?php
2// Este comentario está mal y hay que borrarlo
3$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
4@print "Hola, {$nombre}\n";
```

Y comprobamos:

```
1$ git status
2# On branch master
3# Changes not staged for commit:
4#   (use "git add <file>..." to update what will be committed)
5#   (use "git checkout -- <file>..." to discard changes in working directory)
6#
7#    modified:   hola.php
8#
9no changes added to commit (use "git add" and/or "git commit -a")
```

El mismo Git nos indica que debemos hacer para añadir los cambios o para deshacerlos:

```
1$ git checkout hola.php
2$ git status
3# On branch master
4nothing to commit, working directory clean
5$ cat hola.php
6<?php
7// El nombre por defecto es Mundo
8$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
```

```
9@print "Hola, {$nombre}\n";
```

## Deshaciendo cambios antes del commit

Vamos a hacer lo mismo que la vez anterior, pero esta vez sí añadiremos el cambio al staging (sin hacer commit). Así que volvemos a modificar hola.php igual que la anterior ocasión:

```
1<?php
2// Este comentario está mal y hay que borrarlo
3$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
4@print "Hola, {$nombre}\n";
```

## Y lo añadimos al staging

```
1$ git add hola.php
2$ git status
3# On branch master
4# Changes to be committed:
5#   (use "git reset HEAD <file>..." to unstage)
6#
7#   modified:   hola.php
8#
```

De nuevo, Git nos indica qué debemos hacer para deshacer el cambio:

```
1$ git reset HEAD hola.php
2Unstaged changes after reset:
3M   hola.php
4$ git status
5# On branch master
6# Changes not staged for commit:
7#   (use "git add <file>..." to update what will be committed)
8#   (use "git checkout -- <file>..." to discard changes in working directory)
9#
10#   modified:   hola.php
11#
12no changes added to commit (use "git add" and/or "git commit -a")
13$ git checkout hola.php
```

Y ya tenemos nuestro repositorio limpio otra vez. Como vemos hay que hacerlo en dos pasos: uno para borrar los datos del staging y otro para restaurar la copia de trabajo.

## Deshaciendo commits no deseados.

Si a pesar de todo hemos hecho un commit y nos hemos equivocado, podemos deshacerlo con la orden `git revert`. Modificamos otra vez el archivo como antes:

```
1<?php
```

```
2// Este comentario está mal y hay que borrarlo
3$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
4@print "Hola, {$nombre}\n";
```

Pero ahora sí hacemos commit:

```
1$ git add hola.php
2$ git commit -m "Ups... este commit está mal."
3master 5a5d067] Ups... este commit está mal
4 1 file changed, 1 insertion(+), 1 deletion(-)
```

Bien, una vez confirmado el cambio, vamos a deshacer el cambio con la orden `git revert`:

```
$ git revert HEAD --no-edit
1 [master 817407b] Revert "Ups... este commit está mal"
2 1 file changed, 1 insertion(+), 1 deletion(-)
3 $ git hist
4 * 817407b 2013-06-16 | Revert "Ups... este commit está mal" (HEAD, master) [Sergio Gómez]
5 * 5a5d067 2013-06-16 | Ups... este commit está mal [Sergio Gómez]
6 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1-beta) [Sergio Gómez]
7 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
8 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
9 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

## Borrar commits de una rama

El anterior apartado revierte un commit, pero deja huella en el historial de cambios.

Para hacer que no aparezca hay que usar la orden `git reset`.

```
$ git reset --hard v1
1 HEAD is now at fd4da94 Se añade un comentario al cambio del valor por defecto
2 $ git hist
3 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (HEAD, master) [Sergio Gómez]
4 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
5 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
6 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

El resto de cambios no se han borrado (aún), simplemente no están accesibles porque git no sabe como referenciarlos. Si sabemos su hash podemos acceder aún a ellos. Pasado un tiempo, eventualmente Git tiene un recolector de basura que los borrará. Se puede evitar etiquetando el estado final.

### **Danger**

La orden `reset` es una operación delicada. Debe evitarse si no se sabe bien lo que se está haciendo, sobre todo cuando se trabaja en repositorios compartidos, porque podríamos alterar la historia de cambios lo cual puede provocar problemas de sincronización.

## Modificar un commit

Esto se usa cuando hemos olvidado añadir un cambio a un commit que acabamos de realizar. Tenemos nuestro archivo `hola.php` de la siguiente manera:

```
1<?php
2// Autor: Sergio Gómez
3// El nombre por defecto es Mundo
4$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
5@print "Hola, {$nombre}\n";
```

Y lo confirmamos:

```
1$ git commit -a -m "Añadido el autor del programa"
2[master cf405c1] Añadido el autor del programa
3 1 file changed, 1 insertion(+)
```

---

### Tip

El parámetro `-a` hace un `git add` antes de hacer commit de todos los archivos modificados o borrados (de los nuevos no), con lo que nos ahorramos un paso. Ahora nos percatamos que se nos ha olvidado poner el correo electrónico. Así que volvemos a modificar nuestro archivo:

```
1<?php
2// Autor: Sergio Gómez <sergio@uco.es>
3// El nombre por defecto es Mundo
4$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
5@print "Hola, {$nombre}\n";
```

Y en esta ocasión usamos `commit -amend` que nos permite modificar el último estado confirmado, sustituyéndolo por el estado actual:

```
$ git add hola.php
1$ git commit --amend -m "Añadido el autor del programa y su email"
2[master 96a39df] Añadido el autor del programa y su email
3 1 file changed, 1 insertion(+)
4$ git hist
5* 96a39df 2013-06-16 | Añadido el autor del programa y su email (HEAD, master) [Sergio Gómez]
6* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1-beta) [Sergio Gómez]
7* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
8* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
9* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

---

### Danger

Nunca modifiques un commit que ya hayas sincronizado con otro repositorio o que hayas recibido de él. Estarías alterando la historia de cambios y provocarías problemas de sincronización.



# Moviendo y borrando archivos

## Mover un archivo a otro directorio con git

Para mover archivos usaremos la orden `git mv`:

```
1$ mkdir lib
2$ git mv hola.php lib
3$ git status
4# On branch master
5# Changes to be committed:
6#   (use "git reset HEAD <file>..." to unstage)
7#
8#   renamed:    hola.php -> lib/hola.php
9#
```

## Mover y borrar archivos.

Podíamos haber hecho el paso anterior con la orden del sistema `mv` y el resultado hubiera sido el mismo. Lo siguiente es a modo de ejemplo y no es necesario que lo ejecutes:

```
1$ mkdir lib
2$ mv hola.php lib
3$ git add lib/hola.php
4$ git rm hola.php
```

Y, ahora sí, ya podemos guardar los cambios:

```
1$ git commit -m "Movido hola.php a lib."
2[master 8c2a509] Movido hola.php a lib.
3 1 file changed, 0 insertions(+), 0 deletions(-)
4 rename hola.php => lib/hola.php (100%)
```

# Ramas

## Administración de ramas

### Crear una nueva rama

Cuando vamos a trabajar en una nueva funcionalidad, es conveniente hacerlo en una nueva rama, para no modificar la rama principal y dejarla inestable. Aunque la orden para manejar ramas es `git branch` podemos usar también `git checkout`. Vamos a crear una nueva rama:

```
1git branch hola
```

### Info

---

Si usamos `git branch` sin ningún argumento, nos devolverá la lista de ramas disponibles.

La orden anterior no devuelve ningún resultado y tampoco nos cambia de rama, para eso debemos usar checkout:

```
1$ git checkout hola
2Switched to branch 'hola'
```

### Tip

Hay una forma más rápida de hacer ambas acciones en un solo paso. Con el parámetro `-b` de `git checkout` podemos cambiarnos a una rama que, si no existe, se crea instantáneamente.

```
1$ git checkout -b hola
2Switched to a new branch 'hola'
```

## Modificaciones en la rama secundaria

Añadimos un nuevo archivo en el directorio `lib` llamado `HolaMundo.php`:

```
1<?php
2
3class HolaMundo
4{
5    private $nombre;
6
7    function __construct($nombre)
8    {
9        $this->nombre = $nombre;
10    }
11
12    function __toString()
13    {
14        return sprintf ("Hola, %s.\n", $this->nombre);
15    }
16}
```

Y modificamos `hola.php`:

```
1<?php
2// Autor: Sergio Gómez <sergio@uco.es>
3// El nombre por defecto es Mundo
4require('HolaMundo.php');
5
6$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
7print new HolaMundo($nombre);
```

Podríamos confirmar los cambios todos de golpe, pero lo haremos de uno en uno, con su comentario.

```
1$ git add lib/HolaMundo.php
2$ git commit -m "Añadida la clase HolaMundo"
3[hola 6932156] Añadida la clase HolaMundo
4 1 file changed, 16 insertions(+)
```

```
5 create mode 100644 lib/HolaMundo.php
6$ git add lib/hola.php
7$ git commit -m "hola usa la clase HolaMundo"
8[hola 9862f33] hola usa la clase HolaMundo
9 1 file changed, 3 insertions(+), 1 deletion(-)
```

**Y ahora con la orden `git checkout` podemos movernos entre ramas:**

```
1$ git checkout master
2Switched to branch 'master'
3$ git checkout hola
4Switched to branch 'hola'
```

## Modificaciones en la rama master

**Podemos volver y añadir un nuevo archivo a la rama principal:**

```
1$ git checkout master
2Switched to branch 'master'
```

**Creamos un archivo llamado `README.md` en la raíz de nuestro proyecto con el siguiente contenido:**

```
1# Curso de GIT
2
3Este proyecto contiene el curso de introducción a GIT
```

**Y lo añadimos a nuestro repositorio en la rama en la que estamos:**

```
1$ git add README.md
2$ git commit -m "Añadido README.md"
3[master c3e65d0] Añadido README.md
4 1 file changed, 3 insertions(+)
5 create mode 100644 README.md
6$ git hist --all
7* c3e65d0 2013-06-16 | Añadido README.md (HEAD, master) [Sergio Gómez]
8| * 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Gómez]
9| * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
10|/
11* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
12* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
13* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag:
14Gómez]
15* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
16* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
   * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

**Y vemos como `git hist` muestra la bifurcación en nuestro código.**

## Fusión de ramas y resolución de conflictos

### Mezclar ramas

**Podemos incorporar los cambios de una rama a otra con la orden `git merge`**

```

1 $ git checkout hola
2 Switched to branch 'hola'
3 $ git merge master
4 Merge made by the 'recursive' strategy.
5  README.md | 3 +++
6  1 file changed, 3 insertions(+)
7  create mode 100644 README.md
8  $ git hist --all
9  *    9c6ac06 2013-06-16 | Merge commit 'c3e65d0' into hola (HEAD, hola) [Sergio Gómez]
10 | \
11 * | 9862f33 2013-06-16 | hola usa la clase HolaMundo [Sergio Gómez]
12 * | 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
13 | |
14 | * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
15 | /
16 * 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
17 * 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
18 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: vl-beta) [Sergio Gómez]
19 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: vl-beta) [Sergio Gómez]
20 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
21 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]

```

De esa forma se puede trabajar en una rama secundaria incorporando los cambios de la rama principal o de otra rama.

## Resolver conflictos

Un conflicto es cuando se produce una fusión que Git no es capaz de resolver. Vamos a modificar la rama master para crear uno con la rama hola.

```

1$ git checkout master
2Switched to branch 'master'

```

Modificamos nuestro archivo hola.php de nuevo:

```

1<?php
2// Autor: Sergio Gómez <sergio@uco.es>
3print "Introduce tu nombre:";
4$nombre = trim(fgets(STDIN));
5@print "Hola, {$nombre}\n";

```

Y guardamos los cambios:

```

1$ git add lib/hola.php
2$ git commit -m "Programa interactivo"
3[master 9c85275] Programa interactivo
4 1 file changed, 2 insertions(+), 2 deletions(-)
5$ git hist --all

```

```

6 * 9c6ac06 2013-06-16 | Merge commit 'c3e65d0' into hola (hola) [Sergio Gómez]
7 | \
8 * | 9862f33 2013-06-16 | hola usa la clase HolaMundo [Sergio Gómez]
9 * | 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
10 | | * 9c85275 2013-06-16 | Programa interactivo (HEAD, master) [Sergio Gómez]
11 | | /
12 | * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
13 | /
14 * 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
15 * 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
16 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: Gómez]
17 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
18 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
19 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]

```

**Volvemos a la rama hola y fusionamos:**

```

1$ git checkout hola
2Switched to branch 'hola'
3$ git merge master
4Auto-merging lib/hola.php
5CONFLICT (content): Merge conflict in lib/hola.php
6Automatic merge failed; fix conflicts and then commit the result.

```

**Si editamos nuestro archivo lib/hola.php obtendremos algo similar a esto:**

```

1<?php
2// Autor: Sergio Gómez <sergio@uco.es>
3<<<<<<< HEAD
4// El nombre por defecto es Mundo
5require('HolaMundo.php');
6
7$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
8print new HolaMundo($nombre);
9=====
10print "Introduce tu nombre:";
11$nombre = trim(fgets(STDIN));
12@print "Hola, {$nombre}\n";
13>>>>>>> master

```

La primera parte marca el código que estaba en la rama donde trabajábamos (HEAD) y la parte final el código de donde fusionábamos. Resolvemos el conflicto, dejando el archivo como sigue:

```

1<?php
2// Autor: Sergio Gómez <sergio@uco.es>
3require('HolaMundo.php');
4

```

```
5print "Introduce tu nombre:";
6$nombre = trim(fgets(STDIN));
7print new HolaMundo($nombre);
```

**Y resolvemos el conflicto confirmando los cambios:**

```
1$ git add lib/hola.php
2$ git commit -m "Solucionado el conflicto al fusionar con la rama master"
3[hola a36af04] Solucionado el conflicto al fusionar con la rama master
```

## Rebasing vs Merging

Rebasing es otra técnica para fusionar distinta a merge y usa la orden `git rebase`.

Vamos a dejar nuestro proyecto como estaba antes del fusionado. Para ello necesitamos anotar el hash anterior al de la acción de merge. El que tiene la anotación "hola usa la clase HolaMundo".

Para ello podemos usar la orden `git reset` que nos permite mover HEAD donde queramos.

```
1$ git checkout hola
2Switched to branch 'hola'
3$ git hist
4*   a36af04 2013-06-16 | Solucionado el conflicto al fusionar con la rama master (H
5Gómez]
6|\
7| * 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
8* |   9c6ac06 2013-06-16 | Merge commit 'c3e65d0' into hola [Sergio Gómez]
9|\ \
10|  | /
11| * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
12* | 9862f33 2013-06-16 | hola usa la clase HolaMundo [Sergio Gómez]
13* | 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
14| /
15* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
16* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
17* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag:
18* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gón
19* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
20* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
21$ git reset --hard 9862f33
HEAD is now at 9862f33 hola usa la clase HolaMundo
```

**Y nuestro estado será:**

```
1$ git hist --all
2* 9862f33 2013-06-16 | hola usa la clase HolaMundo (HEAD, hola) [Sergio Gómez]
3* 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
4| * 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
5| * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
```

```

6 |/
7 * 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
8 * 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
9 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag:
10 Gómez]
11 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
12 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
13 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]

```

Hemos desecho todos los merge y nuestro árbol está "limpio". Vamos a probar ahora a hacer un rebase. Continuamos en la rama `hola` y ejecutamos lo siguiente:

```

$ git rebase master
1 First, rewinding head to replay your work on top of it...
2 Applying: Añadida la clase HolaMundo
3 Applying: hola usa la clase HolaMundo
4 Using index info to reconstruct a base tree...
5 M    lib/hola.php
6 Falling back to patching base and 3-way merge...
7 Auto-merging lib/hola.php
8 CONFLICT (content): Merge conflict in lib/hola.php
9 error: Failed to merge in the changes.
10 Patch failed at 0002 hola usa la clase HolaMundo
11 The copy of the patch that failed is found in: .git/rebase-apply/patch
12
13 When you have resolved this problem, run "git rebase --continue".
14 If you prefer to skip this patch, run "git rebase --skip" instead.
15 To check out the original branch and stop rebasing, run "git rebase --
16 abort".

```

El conflicto, por supuesto, se sigue dando. Resolvemos guardando el archivo `hola.php` como en los casos anteriores:

```

1<?php
2// Autor: Sergio Gómez <sergio@uco.es>
3require('HolaMundo.php');
4
5print "Introduce tu nombre:";
6$nombre = trim(fgets(STDIN));
7print new HolaMundo($nombre);

```

Añadimos los cambios en staging y en esta ocasión, y tal como nos indicaba en el mensaje anterior, no tenemos que hacer `git commit` sino continuar con el rebase:

```

1$ git add lib/hola.php
2$ git status
3rebase in progress; onto 269eaca
4You are currently rebasing branch 'hola' on '269eaca'.
5 (all conflicts fixed: run "git rebase --continue")

```

```

6
7Changes to be committed:
8  (use "git reset HEAD <file>..." to unstage)
9
10   modified:   lib/hola.php
11$ git rebase --continue
12Applying: hola usa la clase HolaMundo

```

**Y ahora vemos que nuestro árbol tiene un aspecto distinto, mucho más limpio:**

```

$ git hist --all
1 * 9862f33 2013-06-16 | hola usa la clase HolaMundo (HEAD -> hola) [Sergio Gómez]
2 * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
3 * 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
4 * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
5 * 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
6 * 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
7 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag:
8 Gómez]
9 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Góm
10 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
11 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]

```

Lo que hace rebase es volver a aplicar todos los cambios a la rama máster, desde su nodo más reciente. Eso significa que se modifica el orden o la historia de creación de los cambios. Por eso rebase no debe usarse si el orden es importante o si la rama es compartida.

## Mezclando con la rama master

Ya hemos terminado de implementar los cambios en nuestra rama secundaria y es hora de llevar los cambios a la rama principal. Usamos `git merge` para hacer una fusión normal:

```

1$ git checkout master
2Switched to branch 'master'
3$ git merge hola
4Updating c3e65d0..491f1d2
5Fast-forward
6 lib/HolaMundo.php | 16 ++++++
7 lib/hola.php       |  4 +++-
8 2 files changed, 19 insertions(+), 1 deletion(-)
9 create mode 100644 lib/HolaMundo.php
10 $ git hist --all
11 * 9862f33 2013-06-16 | hola usa la clase HolaMundo (HEAD -> master, hola) [Sergio
12 * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
13 * 9c85275 2013-06-16 | Programa interactivo [Sergio Gómez]
14 * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]

```



```

15 * 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
16 * 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
17 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag:
18 Gómez]
19 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
20 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
21 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]

```

Vemos que indica que el tipo de fusión es fast-forward. Este tipo de fusión tiene el problema que no deja rastro de la fusión, por eso suele ser recomendable usar el parámetro `--no-ff` para que quede constancia siempre de que se ha fusionado una rama con otra.

Vamos a volver a probar ahora sin hacer fast-forward. Reseteamos master al estado "Programa interactivo".

```

$ git reset --hard 9c85275
1 $ git hist --all
2 * 9862f33 2013-06-16 | hola usa la clase HolaMundo (HEAD -> hola) [Sergio Gómez]
3 * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
4 * 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
5 * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
6 * 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
7 * 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
8 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag:
9 Gómez]
10 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
11 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
12 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]

```

Vemos que estamos como en el final de la sección anterior, así que ahora mezclamos:

```

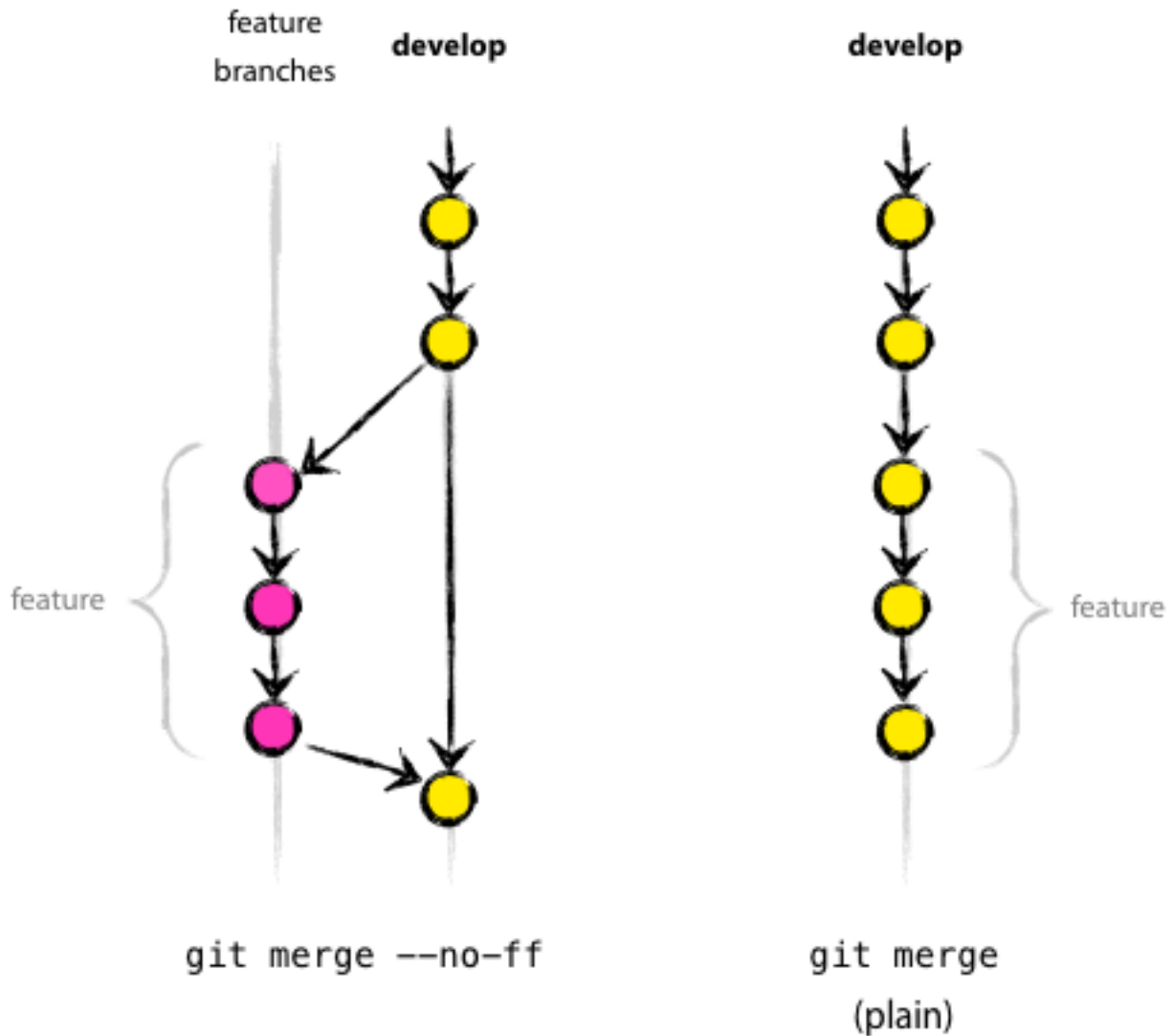
1$ git merge -m "Aplicando los cambios de la rama hola" --no-ff hola
2Merge made by the 'recursive' strategy.
3 lib/HolaMundo.php | 16 ++++++
4 lib/hola.php       |  4 +++-
5 2 files changed, 19 insertions(+), 1 deletion(-)
6 create mode 100644 lib/HolaMundo.php
7$ git hist --all
8* 2eab8ca 2013-06-16 | Aplicando los cambios de la rama hola (HEAD -> master) [Sergio Gómez]
9*\
10| * 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Gómez]
11| * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
12|/
13* 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
14* c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
15* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]

```

```

15 * 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
16 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag:
17 Gómez]
18 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gón
19 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
20 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

En la siguiente imagen se puede ver la diferencia:



# Github

Github es lo que se denomina una forja, un repositorio de proyectos que usan Git como sistema de control de versiones. Es la forja más popular, ya que alberga más de 10 millones de repositorios. Debe su popularidad a sus funcionalidades sociales, principalmente dos: la posibilidad de hacer forks de otros proyectos y la posibilidad de cooperar aportando código para arreglar errores o mejorar el código. Si bien, no es que fuera una novedad, sí lo es lo fácil que resulta hacerlo. A raíz de este proyecto han surgido otros como Gitorius o Gitlab, pero Github sigue siendo el más popular y el que tiene mejores y mayores características. algunas de estas son:

- Un wiki para documentar el proyecto, que usa Markdown como lenguaje de marca.
- Un portal web para cada proyecto.
- Funcionalidades de redes sociales como followers.
- Gráficos estadísticos.
- Revisión de código y comentarios.
- Sistemas de seguimiento de incidencias.

Lo primero es entrar en el portal (<https://github.com/>) para crearnos una cuenta si no la tenemos aún.

## Tu clave pública/privada

Muchos servidores Git utilizan la autenticación a través de claves públicas SSH. Y, para ello, cada usuario del sistema ha de generarse una, si es que no la tiene ya. El proceso para hacerlo es similar en casi cualquier sistema operativo. Ante todo, asegurarte que no tengas ya una clave. (comprueba que el directorio `$HOME/usuario/.ssh` no tiene un archivo `id_dsa.pub` o `id_rsa.pub`).

Para crear una nueva clave usamos la siguiente orden:

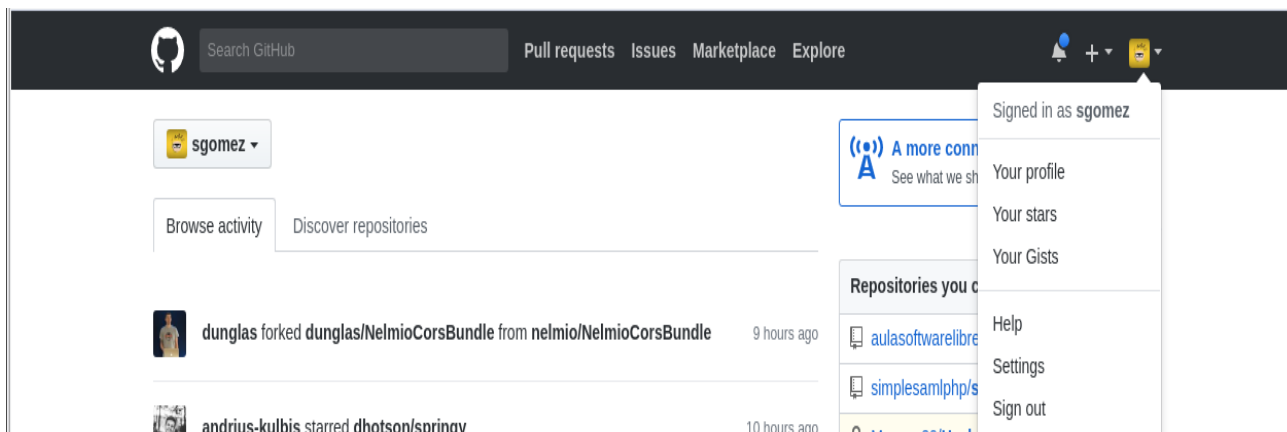
```
1$ ssh-keygen -t rsa -C "Cuenta Thinstation"
```

### Warning

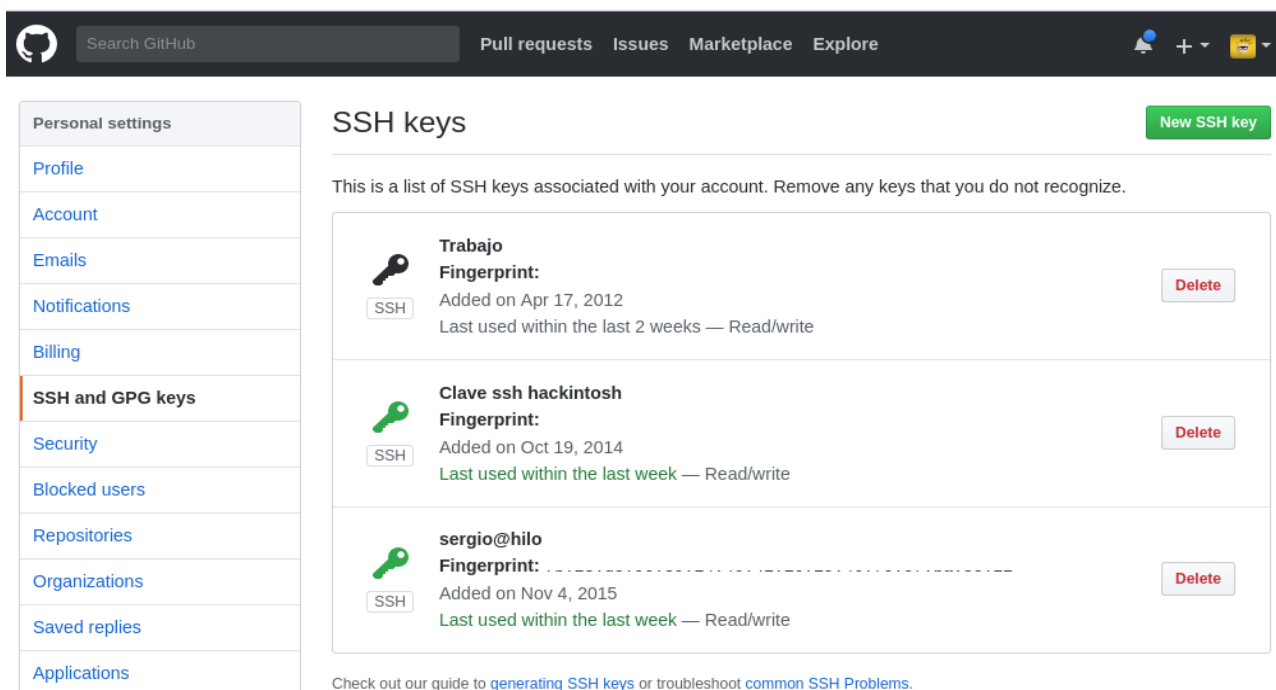
Tu clave RSA te identifica contra los repositorios remotos, asegúrate de no compartir la clave privada con nadie. Por defecto la clave se crea como solo lectura.

## Configuración

Vamos a aprovechar para añadir la clave RSA que generamos antes, para poder acceder desde git a los repositorios. Para ellos nos vamos al menú de configuración de usuario (Settings)



Nos vamos al menú 'SSH and GPG Keys' y añadimos una nueva clave. En Title indicamos una descripción que nos ayude a saber de dónde procede la clave y en key volcamos el contenido del archivo `~/.ssh/id_rsa.pub`. Y guardamos la clave.



Con esto ya tendríamos todo nuestro entorno para poder empezar a trabajar desde nuestro equipo.

## Cientes gráficos para GitHub

Además, para Github existe un cliente propio tanto para Windows como para MacOSX:

- Cliente Windows: <http://windows.github.com/>

- Cliente MacOSX: <http://mac.github.com/>

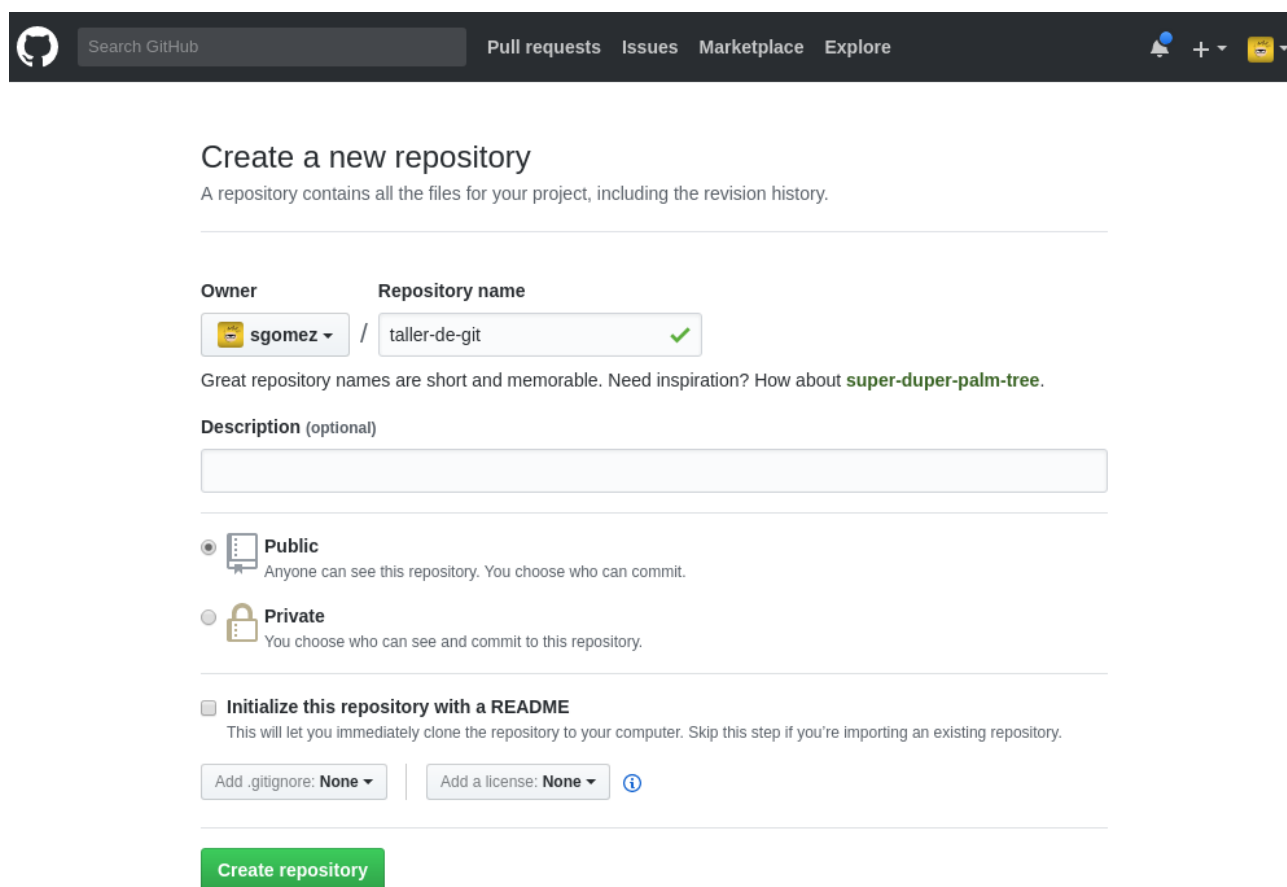
Para Linux no hay cliente propio, pero sí hay plugin para la mayoría de editores de texto como atom, netbeans, eclipse o los editores de jetbrains.

De todas maneras, estos clientes solo tienen el fin de facilitar el uso de Github, pero no son necesarios para usarlo. Es perfectamente válido usar el cliente de consola de Git o cualquier otro cliente genérico para Git. Uno de los más usados actualmente es [GitKraken](#).

## Crear un repositorio

Vamos a crear un repositorio donde guardar nuestro proyecto. Para ello pulsamos el signo + que hay en la barra superior y seleccionamos `New repository`.

Ahora tenemos que designar un nombre para nuestro repositorio, por ejemplo: 'taller-de-git'.



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: `sgomez` / Repository name: `taller-de-git` ✓

Great repository names are short and memorable. Need inspiration? How about `super-duper-palm-tree`.

Description (optional)

☒ **Public**  
Anyone can see this repository. You choose who can commit.

☐ **Private**  
You choose who can see and commit to this repository.

☐ **Initialize this repository with a README**  
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: `None` | Add a license: `None` ⓘ

**Create repository**

Nada más crear el repositorio nos saldrá una pantalla con instrucciones precisas de como proceder a continuación.

Básicamente podemos partir de tres situaciones:

1. Todavía no hemos creado ningún repositorio en nuestro equipo.
2. Ya tenemos un repositorio creado y queremos sincronizarlo con Github.
3. Queremos importar un repositorio de otro sistema de control de versiones distinto.

**Quick setup — if you've done this kind of thing before**

or **HTTPS** **SSH** `https://github.com/sgomez/taller-de-git.git`

We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

**...or create a new repository on the command line**

```
echo "# taller-de-git" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/sgomez/taller-de-git.git
git push -u origin master
```

**...or push an existing repository from the command line**

```
git remote add origin https://github.com/sgomez/taller-de-git.git
git push -u origin master
```

**...or import code from another repository**

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

[Import code](#)

**ProTip!** Use the URL for this page when adding GitHub as a remote.

Nuestra situación es la segunda, así que nos aseguramos de que hemos elegido SSH como protocolo. A continuación pulsamos el icono del portapapeles y ejecutamos las dos ordenes que nos indica la web en nuestro terminal.

```
1$ git remote add origin git@github.com:sgomez/taller-de-git.git
2$ git push -u origin master
3Counting objects: 33, done.
4Delta compression using up to 4 threads.
5Compressing objects: 100% (24/24), done.
6Writing objects: 100% (33/33), 3.35 KiB | 1.12 MiB/s, done.
7Total 33 (delta 2), reused 0 (delta 0)
8remote: Resolving deltas: 100% (2/2), done.
9To github.com:sgomez/taller-de-git.git
10 * [new branch]      master -> master
11Branch master set up to track remote branch master from origin by rebasing.
```

Si recargamos la página veremos que ya aparece nuestro proyecto.

The screenshot shows the GitHub interface for the repository 'sgomez / taller-de-git'. At the top, there's a navigation bar with links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. Below this, the repository name is displayed along with statistics: 1 commit, 0 stars, and 0 forks. A secondary navigation bar includes links for 'Code', 'Issues', 'Pull requests', 'Projects', 'Wiki', 'Insights', and 'Settings'. The main content area shows a message 'No description, website, or topics provided.' followed by an 'Add topics' link. Below this, a summary bar indicates '11 commits', '1 branch', '0 releases', and '1 contributor'. A row of buttons allows for creating new files, uploading files, finding files, or cloning/downloading the repository. The commit history shows a recent commit by 'sgomez' with the message 'Aplicando los cambios de la rama hola'. The file list includes 'lib' and 'README.md'. The 'README.md' file is expanded, showing the title 'Curso de GIT' and the text 'Este proyecto contiene el curso de introducción a GIT'.

## Clonar un repositorio

Una vez que ya tengamos sincronizado el repositorio contra Github, eventualmente vamos a querer descargarlo en otro de nuestros ordenadores para poder trabajar en él. Esta acción se denomina clonar y para ello usaremos la orden `git clone`.

En la página principal de nuestro proyecto podemos ver un botón que indica `Clone or download`. Si la pulsamos nos da, de nuevo, la opción de elegir entre clonar con `ssh` o `https`. Recordad que si estáis en otro equipo y queréis seguir utilizando `ssh` deberéis generar otra par de claves privada/pública como hicimos en la sección de [Aspectos básicos de Git](#) y instalarla en nuestro perfil de Github, como vimos anteriormente.

Para clonar nuestro repositorio y poder trabajar con él todo lo que debemos hacer es lo siguiente:

```
1$ git clone git@github.com:sgomez/taller-de-git.git
2$ cd taller-de-git
```

## Ramas remotas

Si ahora vemos el estado de nuestro proyecto veremos algo similar a esto:

```
1$ git hist --all
```

```

1 * 2eab8ca 2013-06-16 | Aplicando los cambios de la rama hola (HEAD -> master, origin/
2 Gomez]
3 * \
4 | * 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Gómez]
5 | * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
6 | /
7 * 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
8 * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
9 * 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
10 * 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
11 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag:
12 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
13 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
14 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]

```

Aparece que hay una nueva rama llamada `origin/master`. Esta rama indica el estado de sincronización de nuestro repositorio con un repositorio remoto llamado `origin`. En este caso el de Github.

### Info

---

Por norma se llama automáticamente `origin` al primer repositorio con el que sincronizamos nuestro repositorio.

Podemos ver la configuración de este repositorio remoto con la orden `git remote:`

```

1$ git remote show origin
2* remote origin
3  Fetch URL: git@github.com:sgomez/taller-de-git.git
4  Push URL: git@github.com:sgomez/taller-de-git.git
5  HEAD branch: master
6  Remote branch:
7    master tracked
8  Local ref configured for 'git push':
9    master pushes to master (up to date)

```

De la respuesta tenemos que fijarnos en las líneas que indican `fetch` y `push` puesto que son las acciones de sincronización de nuestro repositorio con el remoto. Mientras que `fetch` se encarga de traer los cambios desde el repositorio remoto al nuestro, `push` los envía.

## Enviando actualizaciones

Vamos a añadir una licencia a nuestra aplicación. Creamos un fichero `LICENSE` con el siguiente contenido:

```

1 MIT License
2
3 Copyright (c) [year] [fullname]
4
5 Permission is hereby granted, free of charge, to any person obtaining a copy

```



```
5 of this software and associated documentation files (the "Software"), to deal
6 in the Software without restriction, including without limitation the rights
7 to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
8 copies of the Software, and to permit persons to whom the Software is
9 furnished to do so, subject to the following conditions:
10
11 The above copyright notice and this permission notice shall be included in all
12 copies or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
15 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
16 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
17 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
18 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
19 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
20 SOFTWARE.
21
```

**Y añadidos y confirmamos los cambios:**

```
$ git add LICENSE
1 $ git commit -m "Añadida licencia"
2 [master 3f5cb1c] Añadida licencia
3 1 file changed, 21 insertions(+)
4 create mode 100644 LICENSE
5 $ git hist --all
6 * 3f5cb1c 2013-06-16 | Añadida licencia (HEAD -> master) [Sergio Gómez]
7 * 2eab8ca 2013-06-16 | Aplicando los cambios de la rama hola (origin/master) [Sergio Gómez]
8 * \
9 | * 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Gómez]
10 | * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
11 | /
12 * 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
13 * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
14 * 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
15 * 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
16 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1-beta) [Sergio Gómez]
17 Gómez]
18 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
19 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
20 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

**Viendo la historia podemos ver como nuestro master no está en el mismo punto que origin/master. Si vamos a la web de Github veremos que LICENSE no aparece aún. Así que vamos a enviar los cambios con la primera de las acciones que vimos git push:**

```
1$ git push -u origin master
2Counting objects: 3, done.
3Delta compression using up to 4 threads.
4Compressing objects: 100% (3/3), done.
5Writing objects: 100% (3/3), 941 bytes | 0 bytes/s, done.
6Total 3 (delta 0), reused 0 (delta 0)
7To git@github.com:sgomez/taller-de-git.git
8 2eab8ca..3f5cb1c master -> master
9Branch master set up to track remote branch master from origin.
```

## Info

La orden `git push` necesita dos parámetros para funcionar: el repositorio y la rama destino. Así que realmente lo que teníamos que haber escrito es:

```
1$ git push origin master
```

Para ahorrar tiempo escribiendo git nos deja vincular nuestra rama local con una rama remota, de tal manera que no tengamos que estar siempre indicándolo. Eso es posible con el parámetro `--set-upstream` o `-u` en forma abreviada.

```
1$ git push -u origin master
```

Si repasas las órdenes que te indicó Github que ejecutaras verás que el parámetro `-u` estaba presente y por eso no ha sido necesario indicar ningún parámetro al hacer push.

## Recibiendo actualizaciones

Si trabajamos con más personas, o trabajamos desde dos ordenadores distintos, nos encontraremos con que nuestro repositorio local es más antiguo que el remoto. Necesitamos descargar los cambios para poder incorporarlos a nuestro directorio de trabajo.

Para la prueba, Github nos permite editar archivos directamente desde la web. Pulsamos sobre el archivo `README.md`. En la vista del archivo, veremos que aparece el icono de un lápiz. Esto nos permite editar el archivo.



## Info

Los archivos con extensión `.md` están en un formato denominado Markdown. Se trata de un lenguaje de marca que nos permite escribir texto enriquecido de manera muy sencilla. Dispones de un tutorial aquí: <https://www.markdowntutorial.com/>

Modificamos el archivo como queramos, por ejemplo, añadiendo nuestro nombre:

```
1# Curso de GIT
2
3Este proyecto contiene el curso de introducción a GIT
4
5Desarrollado por Sergio Gómez.
```



### Commit changes

Actualizado README.md

Add an optional extended description...

- ☒ Commit directly to the `master` branch.
- ☐ Create a new branch for this commit and start a pull request. [Learn more about pull requests.](#)

Commit changes

Cancel

El cambio quedará incorporado al repositorio de Github, pero no al nuestro.

Necesitamos traer la información desde el servidor remoto. La orden asociada es `git fetch`:

```
$ git fetch
1
$ git hist --all
2 * cbaf831 2013-06-16 | Actualizado README.md (origin/master) [Sergio Gómez]
3 * 3f5cb1c 2013-06-16 | Añadida licencia (HEAD -> master) [Sergio Gómez]
4 * 2eab8ca 2013-06-16 | Aplicando los cambios de la rama hola [Sergio Gomez]
5 * \
6 | * 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Gómez]
7 | * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
8 | /
9 * 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
10 * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
11 * 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
12 * 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
13 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: Gómez]
14 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gón
15 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
16 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Ahora vemos el caso contrario, tenemos que `origin/master` está por delante que `HEAD` y que la rama `master` local.

Ahora necesitamos incorporar los cambios de la rama remota en la local. La forma de hacerlo lo vimos en el [capítulo anterior](#) usando `git merge` o `git rebase`.

Habitualmente se usa `git merge`:

```
$ git merge origin/master
1 Updating 3f5cb1c..cbaf831
2 Fast-forward
3 README.md | 2 ++
4 1 file changed, 2 insertions(+)
5
```

```

6 $ git hist --all
7 * cbaf831 2013-06-16 | Actualizado README.md (HEAD -> master, origin/master) [Sergio Gómez]
8 * 3f5cb1c 2013-06-16 | Añadida licencia [Sergio Gómez]
9 * 2eab8ca 2013-06-16 | Aplicando los cambios de la rama hola [Sergio Gomez]
10 * \
11 | * 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Gómez]
12 | * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
13 | /
14 * 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
15 * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
16 * 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
17 * 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
18 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: Gómez]
19 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
20 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
21 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]

```

Como las operaciones de traer cambios (`git fetch`) y de mezclar ramas (`git merge` o `git rebase`) están muy asociadas, git nos ofrece una posibilidad para ahorrar pasos que es la orden `git push` que realiza las dos acciones simultáneamente. Para probar, vamos a editar de nuevo el archivo README.md y añadimos algo más:

```

1 # Curso de GIT
2
3 Este proyecto contiene el curso de introducción a GIT del Aula de Software
4 Libre.
5
6 Desarrollado por Sergio Gómez.

```

Como mensaje del commit: 'Indicado que se realiza en el ASL'.

Y ahora probamos a actualizar con `git pull`:

```

1 $ git pull
2 remote: Counting objects: 3, done.
3 remote: Compressing objects: 100% (3/3), done.
4 remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
5 Unpacking objects: 100% (3/3), done.
6 From github.com:sgomez/taller-de-git
7   cbaf831..d8922e4  master      -> origin/master
8 First, rewinding head to replay your work on top of it...
9 Fast-forwarded master to d8922e4ffa4f87553b03e77df6196b7e496bfec4.
10 $ git hist --all
11 * d8922e4 2013-06-16 | Indicado que se realiza en el ASL (HEAD -> master, origin/master) [Sergio Gómez]
12
13 * cbaf831 2013-06-16 | Actualizado README.md [Sergio Gómez]
14 * 3f5cb1c 2013-06-16 | Añadida licencia [Sergio Gómez]

```

```

15 * 2eab8ca 2013-06-16 | Aplicando los cambios de la rama hola [Sergio Gomez]
16 * \
17 | * 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Gómez]
18 | * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
19 | /
20 * 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
21 * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
22 * 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
23 * 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
24 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag:
25 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gón
26 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
27 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]

```

Vemos que los cambios se han incorporado y que las ramas remota y local de master están sincronizadas.

## Problemas de sincronización

### No puedo hacer push

Al intentar subir cambios nos podemos encontrar un mensaje como este:

```

1$ git push
2git push
3To git@github.com:sgomez/taller-de-git.git
4 ! [rejected]          master -> master (fetch first)
5error: failed to push some refs to 'git@github.com:sgomez/taller-de-git.git'
6hint: Updates were rejected because the remote contains work that you do
7hint: not have locally. This is usually caused by another repository pushing
8hint: to the same ref. You may want to first integrate the remote changes
9hint: (e.g., 'git pull ...') before pushing again.
10hint: See the 'Note about fast-forwards' in 'git push --help' for details.

```

La causa es que el repositorio remoto también se ha actualizado y nosotros aún no hemos recibido esos cambios. Es decir, ambos repositorios se han actualizado y el remoto tiene preferencia. Hay un conflicto en ciernes y se debe resolver localmente antes de continuar.

Vamos a provocar una situación donde podamos ver esto en acción. Vamos a modificar el archivo `README.md` tanto en local como en remoto a través del interfaz web.

En el web vamos a cambiar el título para que aparezca de la siguiente manera.

```
1Curso de GIT, 2020
```

En local vamos a cambiar el título para que aparezca de la siguiente manera.

```
1Curso de GIT, febrero
```

---

### Question

Haz el commit para guardar el cambio en local.

Respuesta al ejercicio anterior

La forma de proceder en este caso es hacer un `git fetch` y un `git rebase`. Si hay conflictos deberán resolverse. Cuando esté todo solucionado ya podremos hacer `git push`.

## Info

---

Por defecto `git pull` lo que hace es un `git merge`, si queremos hacer `git rebase` deberemos especificarlos con el parámetro `-r`:

```
1$ git pull --rebase
```

Vamos a hacer el pull con rebase y ver qué sucede.

```
1$ git pull --rebase
2First, rewinding head to replay your work on top of it...
3Applying: Añadido el mes al README
4Using index info to reconstruct a base tree...
5M   README.md
6Falling back to patching base and 3-way merge...
7Auto-merging README.md
8CONFLICT (content): Merge conflict in README.md
9error: Failed to merge in the changes.
10Patch failed at 0001 Añadido el mes al README
11hint: Use 'git am --show-current-patch' to see the failed patch
12
13Resolve all conflicts manually, mark them as resolved with
14"git add/rm <conflicted_files>", then run "git rebase --continue".
15You can instead skip this commit: run "git rebase --skip".
16To abort and get back to the state before "git rebase", run "git rebase --
  abort".
```

Evidentemente hay un conflicto porque hemos tocado el mismo archivo. Se deja como ejercicio resolverlo.

Respuesta al ejercicio anterior

## Warning

---

¿Por qué hemos hecho rebase en master si a lo largo del curso hemos dicho que no se debe cambiar la línea principal?

Básicamente hemos dicho que lo que no debemos hacer es modificar la línea temporal compartida. En este caso nuestros cambios en master solo estaban en nuestro repositorio, porque al fallar el envío nadie más ha visto nuestras actualizaciones. Al hacer rebase estamos deshaciendo nuestros cambios, bajarnos la última actualización compartida de master y volviéndolos a aplicar. Con lo que realmente la historia compartida no se ha modificado.

Este es un problema que debemos evitar en la medida de lo posible. La menor cantidad de gente posible debe tener acceso de escritura en master y las actualizaciones de dicha rama deben hacerse a través de ramas secundarias y haciendo merge en master como hemos visto en el capítulo de ramas.

## No puedo hacer pull

Al intentar descargar cambios nos podemos encontrar un mensaje como este:

```
1$ git pull
2error: Cannot pull with rebase: You have unstaged changes.
```

O como este:

```
1$ git pull
2error: Cannot pull with rebase: Your index contains uncommitted changes.
```

Básicamente lo que ocurre es que tenemos cambios sin confirmar en nuestro espacio de trabajo. Una opción es confirmar (commit) y entonces proceder como el caso anterior.

Pero puede ocurrir que aún estemos trabajando todavía y no nos interese confirmar los cambios, solo queremos sincronizar y seguir trabajando. Para casos como estos git ofrece una pila para guardar cambios temporalmente. Esta pila se llama stash y nos permite restaurar el espacio de trabajo al último commit.

De nuevo vamos a modificar nuestro proyecto para ver esta situación en acción.

## Example

---

En remoto borra el año de la fecha y en local borra el mes. Pero esta vez no hagas commit en local. El archivo solo debe quedar modificado.

La forma de proceder es la siguiente:

```
$ git stash save # Guardamos los cambios en la pila
1$ git pull # Sincronizamos con el repositorio remoto, -r para hacer rebase puede ser
2requerido
3$ git stash pop # Sacamos los cambios de la pila
```

## Info

---

Como ocurre habitualmente, git nos proporciona una forma de hacer todos estos pasos de una sola vez. Para ello tenemos que ejecutar lo siguiente:

```
1$ git pull --autostash
```

En general no es mala idea ejecutar lo siguiente si somos conscientes, además, de que tenemos varios cambios sin sincronizar:

```
1$ git pull --autostash --rebase
```

Podría darse el caso de que al sacar los cambios de la pila hubiera algún conflicto. En ese caso actuamos como con el caso de merge o rebase.

De nuevo este tipo de problemas no deben suceder si nos acostumbramos a trabajar en ramas.

## Citar proyectos en GitHub

Extraído de la [guía oficial de GitHub](#).

A través de una aplicación de terceros (Zenodo, financiado por el CERN), es posible crear un DOI para uno de nuestros proyectos.

Estos son los pasos

### Paso 1. Elegir un repositorio

Este repositorio debe ser abierto (público), o de lo contrario Zenodo no podrá acceder al mismo. Hay que recordar escoger una licencia para el proyecto. Esta web puede ayudarnos <http://choosealicense.com/>.

### Paso 2. Entrar en Zenodo

Iremos a [Zenodo](#) y haremos login con GitHub. Lo único que tenemos que hacer en esta parte es autorizar a Zenodo a conectar con nuestra cuenta de GitHub.

#### Important

---

Si deseas archivar un repositorio que pertenece a una organización en GitHub, deberás asegurarte de que el administrador de la organización haya habilitado el acceso de terceros a la aplicación Zenodo.

### Paso 3. Seleccionar los repositorios

En este punto, hemos autorizado a Zenodo para configurar los permisos necesarios para permitir el archivado y la emisión del DOI. Para habilitar esta funcionalidad, simplemente haremos clic en el botón que está junto a cada uno de los repositorios que queremos archivar.

#### Important

---

Zenodo solo puede acceder a los repositorios públicos, así que debemos asegurarnos de que el repositorio que deseamos archivar sea público.

### Paso 4. Crear una nueva release

Por defecto, Zenodo realiza un archivo de nuestro repositorio de GitHub cada vez que crea una nueva versión. Como aún no tenemos ninguna, tenemos que volver a la vista del repositorio principal y hacer clic en el elemento del encabezado de versiones (releases).

### Paso 5. Acuñar un DOI

Antes de que Zenodo pueda emitir un DOI para nuestro repositorio, deberemos proporcionar cierta información sobre el repositorio de GitHub que acaba de archivar.



Una vez que estemos satisfechos con la descripción, heremos clic en el botón publicar.

## Paso 6. Publicar

De vuelta a nuestra página de Zenodo, ahora deberíamos ver el repositorio listado con una nueva insignia que muestra nuestro nuevo DOI.

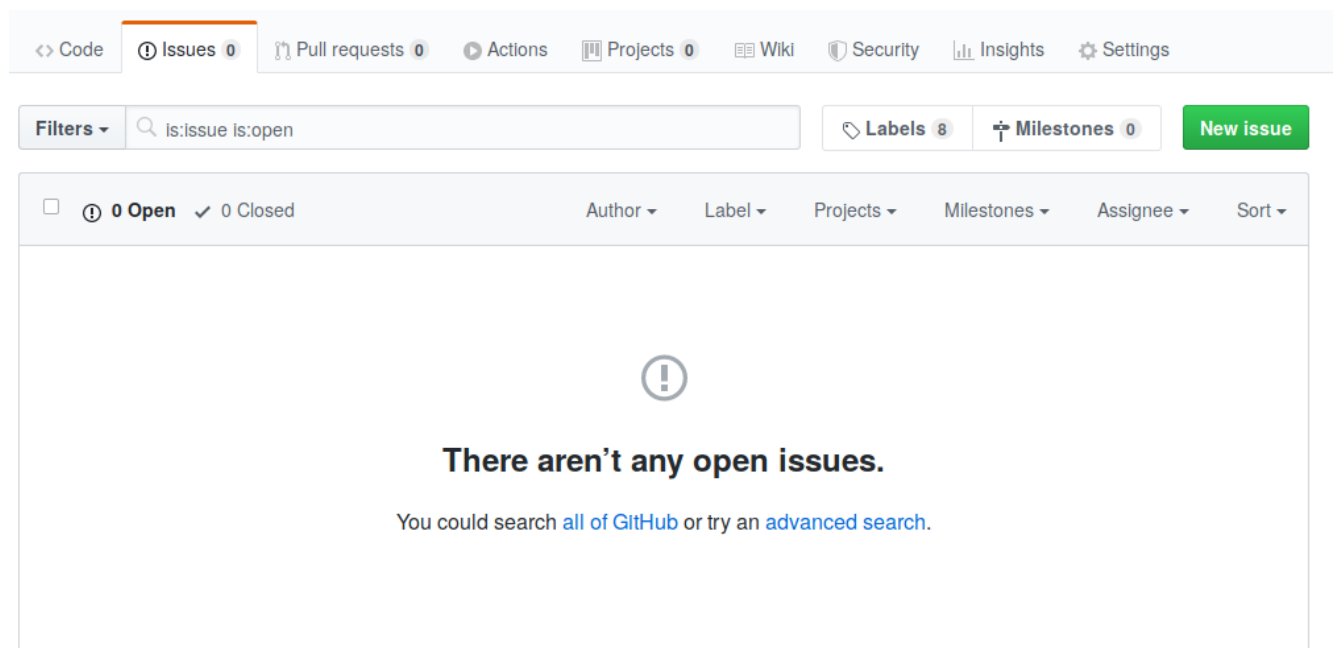
### Tip

Podemos colocar la insignia en nuestro proyecto. Para eso haremos clic en la imagen DOI gris y azul. Se abrirá una ventana emergente y el texto que aparece como Markdown es el que deberemos copiar en nuestro archivo README.md.

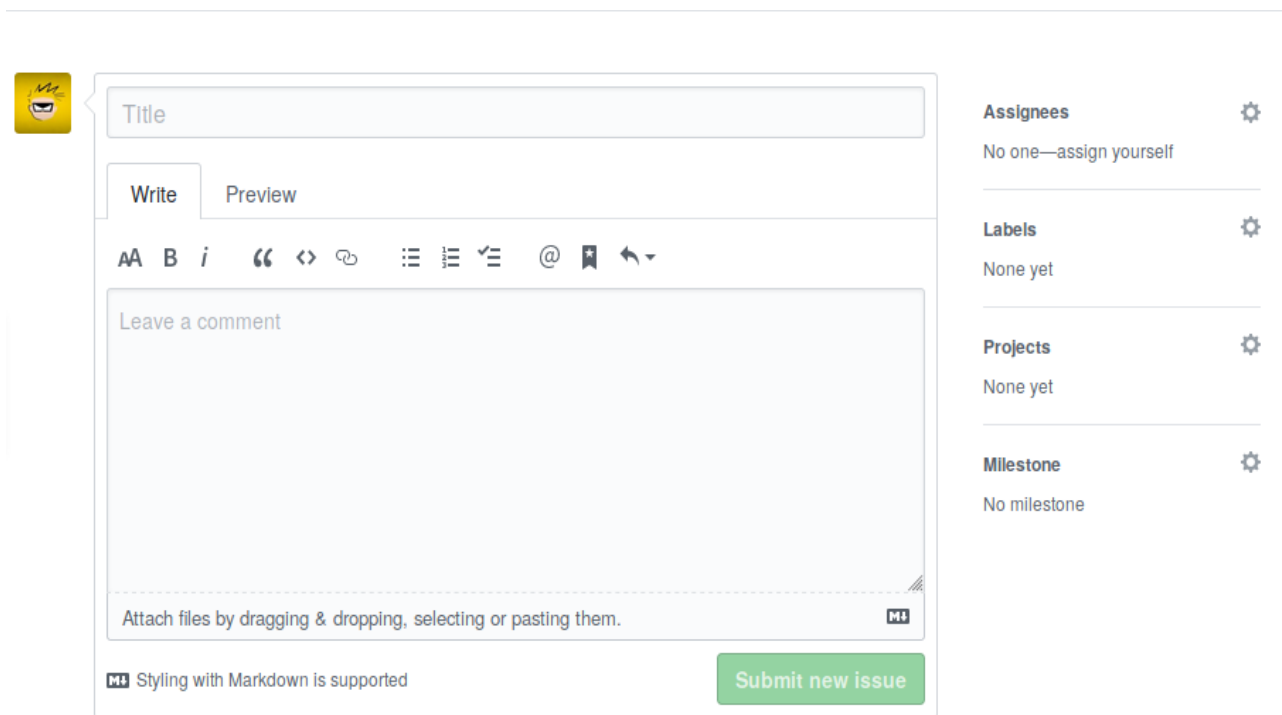
# Flujo de trabajo en GitHub

## Paso 0. Abrir una incidencia (issue)

Habitualmente el trabajo puede partir a raíz de una reporte por parte de un miembro del equipo o de una persona externa. Para eso tenemos la sección Issues.



Una issue cuando se crea se compone de un título y una descripción en Markdown. Si la persona es miembro del equipo, opcionalmente puede asignarle una serie de metadatos: etiquetas (labels), hitos (milestone), proyecto al que pertenece o responsables encargados de cerrar la incidencia.



Una vez creado, al mismo se le asignará un número.

### Example

Vamos a crear una incidencia llamada "Crear archivo de autores", donde indiquemos que vamos a crear un archivo `AUTHORS.md` con la lista de desarrolladores del proyecto.

## Paso 1. Crear una rama

Crearemos una rama cada vez que queramos implementar una nueva característica al proyecto que estamos realizando. La misma puede estar provocada por una incidencia o no.

### Tip

Es una buena costumbre crear en Issues el listado de casos de uso, requisitos, historias de usuario o tareas (como lo queramos llamar), para tener un registro del trabajo que llevamos y el que nos queda.

El nombre de la rama puede ser el que creamos conveniente, pero hay que intentar ser coherente y usar siempre el mismo método, sobre todo si trabajamos en equipo.

Un método puede ser el siguiente:

```
1$ # tipo-número/descripción
2$ git checkout -b feature-1/create-changelog
3$ git checkout -b hotfix-2/updated-database
```

En entornos de trabajo multiusuario se puede usar el siguiente:

```
1$ # usuario/tipo-número/descripción
2$ git checkout -b sgomez/feature-1/create-changelog
3$ git checkout -b sgomez/hotfix-2/updated-database
```

De esa manera, podemos seguir fácilmente quién abrió la rama, en qué consiste y a qué issues está conectada. Pero como decimos es más un convenio que una imposición, pudiéndole poner el nombre que queramos.

Vamos a crear la rama y los commits correspondientes y subir la rama con push al servidor.

```
1$ git checkout -b sgomez/feature-1/create-changelog
2$ git add AUTHORS.md
3$ git commit -m "Añadido fichero de autores"
```

El archivo puede contener, por ejemplo, lo siguiente:

```
1# AUTHORS
2
3* Sergio Gómez <sergio@uco.es>
```

Hacemos push y obtenemos algo como esto:

```
1$ git push
2fatal: The current branch sgomez/feature-1/create-changelog has no upstream branch.
3To push the current branch and set the remote as upstream, use
4
5    git push --set-upstream origin sgomez/feature-1/create-changelog
```

Como la rama es nueva, git no sabe dónde debe hacer push. Le indicamos que debe hacerla en origin y además que guarde la vinculación (equivalente al parámetro -u que vimos en el capítulo anterior). Probamos de nuevo:

```
1$ git push -u origin sgomez/feature-1/create-changelog
2Enumerating objects: 4, done.
3Counting objects: 100% (4/4), done.
4Delta compression using up to 4 threads
5Compressing objects: 100% (2/2), done.
6Writing objects: 100% (3/3), 1.03 KiB | 1.03 MiB/s, done.
7Total 3 (delta 0), reused 0 (delta 0)
8remote:
9remote: Create a pull request for 'sgomez/feature-1/create-changelog' on GitHub by
10remote:      https://github.com/sgomez/taller-de-git/pull/new/sgomez/feature-1/create-changelog
11remote:
12To github.com:sgomez/taller-de-git.git
13* [new branch]      sgomez/feature-1/create-changelog -> sgomez/feature-1/create-changelog
14Branch 'sgomez/feature-1/create-changelog' set up to track remote branch 'sgomez/feature-1/create-changelog' of 'origin'.
```

Ahora la rama ya se ha subido y nos informa, además, de que podemos crear un Pull Request (PR). Si vamos al enlace que nos aparece veremos lo siguiente:

# Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

base: master ← compare: sgomez/feature-1/create-cha...

✓ Able to merge. These branches can be automatically merged.

Añadido fichero de autores

Write

Preview

AA B i “ <> ↻ ☰ ☷ ☰ @ 📖 ↶

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

Create pull request

Reviewers

No reviews

Assignees

No one—assign yourself

Labels

None yet

Projects

None yet

Milestone

No milestone

Aquí podemos informar de en qué consiste la rama que estamos enviando. Si ya tenemos una issue abierta, no es necesario repetir la misma información. Podemos hacer referencia con el siguiente texto:

```
1Closes #1
```

Esto lo que le indica a GitHub que esta PR cierra el issues número 1. Cuando se haga el merge de la rama, automáticamente se cerrará la incidencia.

Lo hacemos y le damos a crear.

Añadido fichero de autores #2

Edit

Open

sgomez wants to merge 1 commit into master from sgomez/feature-1/create-changelog

Conversation 0

Commits 1

Checks 0

Files changed 1

+4 -0

sgomez commented now

Closes #1

Añadido fichero de autores

Verified

ad22e7e

Add more commits by pushing to the sgomez/feature-1/create-changelog branch on sgomez/taller-de-git.

Continuous integration has not been set up

GitHub Actions and several other apps can be used to automatically catch bugs and enforce style.

✓ This branch has no conflicts with the base branch

Merging can be performed automatically.

Merge pull request

or view command line instructions.

Reviewers

No reviews

Assignees

No one—assign yourself

Labels

None yet

Projects

None yet

Milestone

No milestone

Notifications

Customize

Unsubscribe

## Paso 2. Crear commits

A partir de ahora podemos seguir creando commits en local y enviarlos hasta que terminemos de trabajar.

Editamos el archivo AUTHORS.md .

```
1# AUTHORS
2
3* Sergio Gómez <sergio@uco.es>
4* John Doe
```

Y mandamos otro commit

```
1$ git commit -am "Actualizado AUTHORS.md"
2$ git push
```

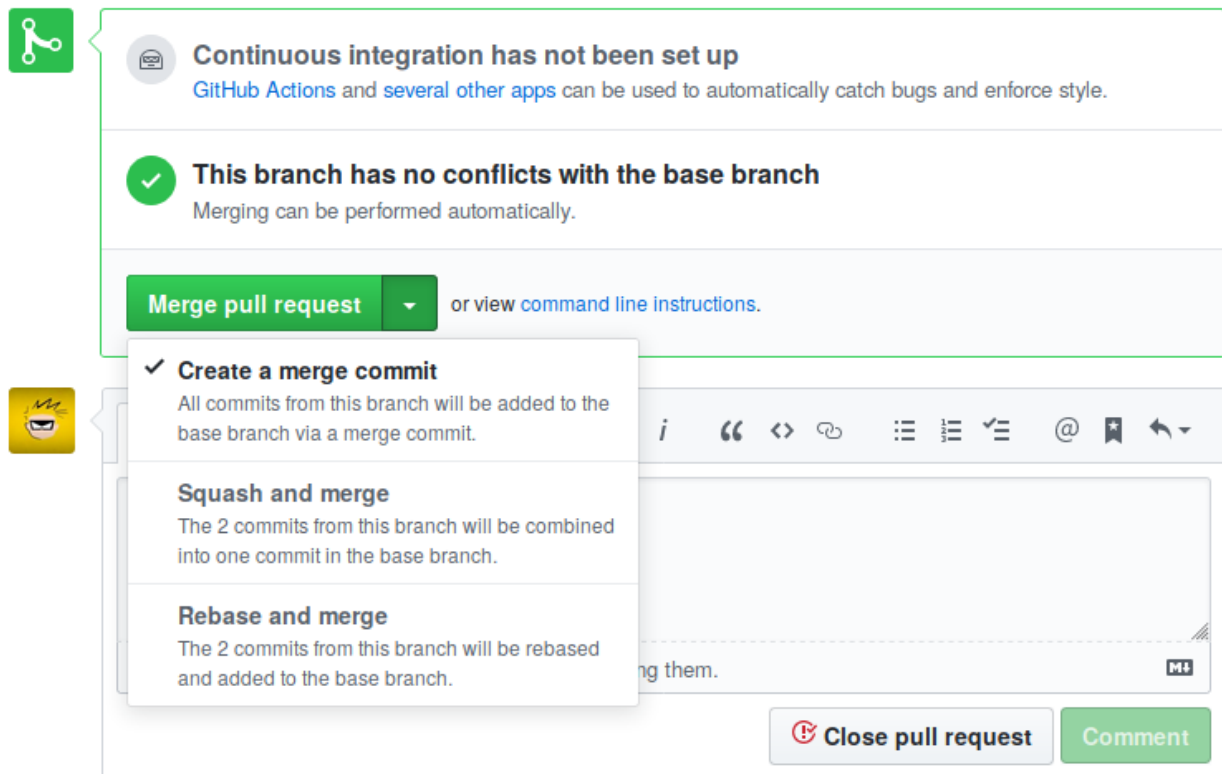
Si volvemos a la página de PR, veremos que aparece el nuevo commit que acabamos de enviar.

## Paso 3. Discutir

GitHub permite que entre los desarrolladores se pueda abrir una discusión sobre el código, de tal manera que el trabajo de crear la rama sea colaborativo. Se puede incluso pedir revisiones por parte de terceros y que esas revisiones sean obligatorias antes de aceptar los cambios.

## Paso 4. Desplegar

Una vez que hemos terminado de crear la función de la rama ya podemos incorporar los cambios a master. Este trabajo ya no es necesario hacerlo en local y GitHub nos proporciona 3 maneras de hacerlo:



### Crear un merge commit

Esta opción es el equivalente a hacer lo siguiente en nuestro repositorio:

```
1$ git checkout master
2$ git merge --no-ff sgomez/feature-1/create-changelog
3$ git push
```

Es decir, el equivalente a hacer un merge entre nuestra rama y master.

### Info

GitHub siempre desactiva el fast forward.

### Crear un rebase y merge

Esta opción es el equivalente a hacer lo siguiente en nuestro repositorio

```
1$ git rebase master
2$ git checkout master
3$ git merge --no-ff sgomez/feature-1/create-changelog
4$ git push
```

Es decir, nos aseguramos de que nuestra rama está al final de master haciendo rebase, como vimos en el capítulo de ramas, y posteriormente se hace el merge.

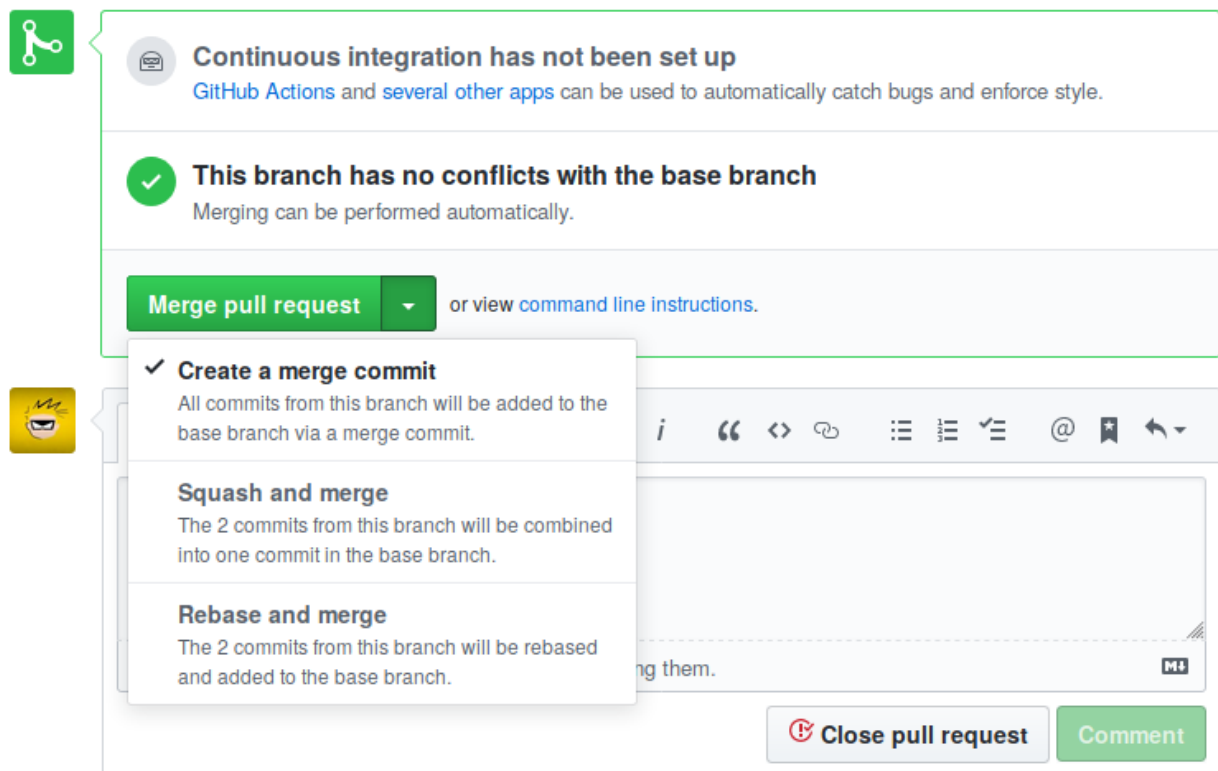
### Crear un squash commit y un merge

Esta opción es el equivalente a hacer lo siguiente en nuestro repositorio:

```
1$ git checkout master
2$ git merge --squash sgomez/feature-1/create-changelog
3$ git push
```

Esta opción es algo especial. En vez de aplicar cada uno de los commits en la rama master, ya sea directamente (fast forward) o no, lo que hace es crear un solo commit con los cambios de todos los commits de la rama. El efecto final es como si en la rama solo hubiera producido un solo commit.

Vamos a seleccionar este último (squash and merge) y le damos al botón para activarlo. Nos saldrá una caja para que podamos crear una descripción del commit y le damos a confirmar.



Ya hemos terminado y nos aparecerá una opción para borrar la rama, lo más recomendado para no tener ramas obsoletas.

Las consecuencias de esta acción son las siguientes:

- 1.El PR aparecerá como estado merged y en la lista de PR como cerrado.
- 2.El issue que abrimos se habrá cerrado automáticamente.
- 3.En el listado de commits aparecerá solo uno con un enlace al PR (en vez de los dos commits que hicimos).

## Paso 5. Sincronizar

Hemos cambiado el repositorio en GitHub, pero nuestra rama master no contiene los mismos cambios que el de origen. Así que nos toca sincronizar y borrar la rama obsoleta:

```
1$ git checkout master
2$ git pull --rebase --autostash
3$ git branch -D sgomez/feature-1/create-changelog
```

## Info

---

¿Por qué squash and merge y no un merge o rebase? De nuevo depende de los gustos de cada equipo de desarrollo. Las características de squash es que elimina (relativamente) rastros de errores intermedios mientras se implementaba la rama, deja menos commits en la rama master y nos enlace al PR donde se implementaron los cambios.

Para algunas personas estas características son unas ventajas, para otras no. Lo mejor es experimentar cada opción y cada uno decida como quiere trabajar.

# Github avanzado

Esta sección trata de cómo colaborar con proyectos de terceros.

## Clonar un repositorio

Nos vamos a la web del proyecto en el que queremos colaborar. En este caso el proyecto se encuentra en <https://github.com/sgomez/miniblog>. Pulsamos en el botón de fork y eso creará una copia en nuestro perfil.



Una vez se termine de clonar el repositorio, nos encontraremos con el espacio de trabajo del mismo:

- En la parte superior información sobre los commits, ramas, etiquetas, etc.
- Justo debajo un explorador de archivos.
- En la parte derecha un selector para cambiar de contexto entre: explorador de código, peticiones de colaboración (pull request), wiki, configuración, etc.
- Justo abajo a la derecha información sobre como clonar localmente o descargar un proyecto.



Trata de un blog muy simple hecho con silex, con un frontend y un backend sencillo para la creación de nuevas entradas — Edit

4 commits2 branches1 release1 contributor

branch: masterminiblog

This branch is 0 commits ahead and 0 commits behind master

Archivo de versión

sgomez	authored 8 months ago	latest commit 95e156d679
config	Primer commit	a year ago
src	Primer commit	a year ago
templates	Pequeño bug a la hora de mostrar la lista de entradas	a year ago
web	Primer commit	a year ago
.gitignore	Fix: Error al añadir composer.lock al repositorio	8 months ago
README.md	Primer commit	a year ago
RELEASE	Archivo de versión	8 months ago
composer.json	Primer commit	a year ago
console	Primer commit	a year ago

Code

Pull Requests0

Wiki

Pulse

Graphs

Network

Settings

SSH clone URL

git@github.com:ceer

You can clone with HTTPS, SSH, or Subversion.

Download ZIP

Github nos permite clonar localmente un proyecto por tres vías: HTTPS, SSH y Subversion. Seleccionamos SSH y copiamos el texto que después añadiremos a la orden `git clone` como en la primera línea del siguiente grupo de órdenes:

```
1$ git clone git@github.com:miusuario/miniblog.git
2$ cd miniblog
3$ composer.phar install
4$ php console create-schema
```

Lo que hace el código anterior es:

- 1.Clona el repositorio localmente
- 2.Entramos en la copia
- 3.Instalamos las dependencias que la aplicación tiene
- 4.Arrancamos un servidor web para pruebas

Y probamos que nuestra aplicación funciona:

```
1$ php -S localhost:9999 -t web/
```

Podemos usar dos direcciones para probarla:

- Frontend: [http://localhost:9999/index\\_dev.php](http://localhost:9999/index_dev.php)
- Backend: [http://localhost:9999/index\\_dev.php/admin/](http://localhost:9999/index_dev.php/admin/) con usuario admin y contraseña 1234.

## Sincronizar con el repositorio original

Cuando clonamos un repositorio de otro usuario hacemos una copia del original. Pero esa copia es igual al momento en el que hicimos la copia. Cuando el repositorio original cambie, que lo hará, nuestro repositorio no se actualizará solo. ¡Son dos repositorios diferentes! Necesitamos una manera de poder incorporar los cambios que vaya teniendo el repositorio original en el nuestro. Para eso crearemos una nueva rama remota. Por convenio, y como vimos anteriormente, ya existe una rama

remota llamada origin que apunta al repositorio de donde clonamos el proyecto, en este caso apunta a nuestro fork en github:

```
1$ git remote show origin
2* remote origin
3  Fetch URL: git@github.com:miusuario/miniblog.git
4  Push URL: git@github.com:miusuario/miniblog.git
5  HEAD branch (remote HEAD is ambiguous, may be one of the following):
6    develop
7    master
8  Remote branches:
9    develop tracked
10   master tracked
11  Local branch configured for 'git pull':
12    master merges with remote master
13  Local ref configured for 'git push':
14    master pushes to master (up to date)
```

También por convenio, la rama remota que hace referencia al repositorio original se llama upstream y se crea de la siguiente manera:

```
1$ git remote add upstream git@github.com:sgomez/miniblog.git
2$ git remote show upstream
3* remote upstream
4  Fetch URL: git@github.com:sgomez/miniblog.git
5  Push URL: git@github.com:sgomez/miniblog.git
6  HEAD branch: master
7  Remote branches:
8    develop new (next fetch will store in remotes/upstream)
9    master new (next fetch will store in remotes/upstream)
10  Local ref configured for 'git push':
11    master pushes to master (local out of date)
```

En este caso, la URI debe ser siempre la del proyecto original. Y ahora para incorporar actualizaciones, usaremos el merge en dos pasos:

```
1$ git fetch upstream
2$ git merge upstream/master
```

Recordemos que fetch solo trae los cambios que existan en el repositorio remoto sin hacer ningún cambio en nuestro repositorio. Es la orden merge la que se encarga de que todo esté sincronizado. En este caso decimos que queremos fusionar con la rama master que está en el repositorio upstream.

## Creando nuevas funcionalidades

Vamos a crear una nueva funcionalidad: vamos a añadir una licencia de uso. Para ello preferentemente crearemos una nueva rama.

```
1$ git checkout -b add-license
```

```
2$ echo "LICENCIA MIT" > LICESE
3# el error es intencionado
4$ git add LICESE
5$ git commit -m "Archivo de licencia de uso"
```

En principio habría que probar que todo funciona bien y entonces integraremos en la rama master de nuestro repositorio y enviamos los cambios a Github:

```
1$ git checkout master
2$ git merge add-license --no-ff
3$ git branch -d add-license
4# Borramos la rama que ya no nos sirve para nada
5$ git push --set-upstream origin add-license
6# Enviamos la rama a nuestro repositorio origin
```



Si volvemos a Github, veremos que nos avisa de que hemos subido una nueva rama y si queremos crear un pull request.

Your recently pushed branches:

 **add-license** (1 minute ago)

 **Compare & pull request**

Pulsamos y entramos en la petición de Pull Request. Este es el momento para revisar cualquier error antes de enviar al dueño del repositorio. Como vemos hemos cometido uno, nombrando el fichero, si lo corregimos debemos hacer otro push para ir actualizando la rama. Cuando esté lista volvemos aquí y continuamos. Hay que dejar una descripción del cambio que vamos a hacer.

 base fork: **sgomez/miniblog** ▾ base: **master** ▾ ... head fork: **ceepsuco/miniblog** ▾ compare: **add-license** ▾ 


Archivo de licencia de uso

Write


Preview

Comments are parsed with [GitHub Flavored Markdown](#)

Leave a comment



Attach images by dragging & dropping, [selecting them](#), or pasting from the clipboard.

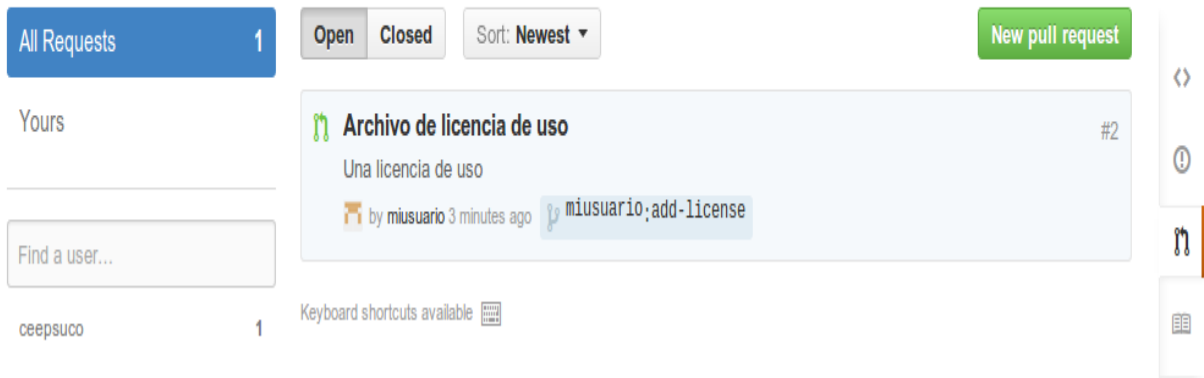


✓ **Able to merge**

These branches can be automatically merged

**Send pull request**

Una vez hemos terminado y nos aseguramos que todo está correcto, pulsamos Send pull request y le llegará nuestra petición al dueño del proyecto.



Sin embargo, para esta prueba, no vamos a cambiar el nombre del archivo y dejaremos el error como está. Así de esta manera al administrador del proyecto le llegará el Pull Request y la lista de cambios. Ahora en principio, cabría esperar que el administrador aprobara los cambios, pero podría pasar que nos indicara que cambiemos algo. En ese caso solo habría que modificar la rama y volverla a enviar.

```
1$ git mv LICESE LICENSE
2$ git commit -m "Fix: Nombre de archivo LICENSE"
3$ git push
```

Ahora sí, el administrador puede aprobar la fusión y borrar la rama del repositorio. El panel de Github permite aceptar los cambios directamente o informa de como hacer una copia de la rama ofrecida por el usuario para hacer cambios, como puede verse en la siguiente imagen.

miusuario commented 20 minutes ago

Una licencia de uso

Usuario de SLCS added some commits an hour ago

- Archivo de licencia de uso 6eb3d86
- Fix: Nombre de archivo LICENSE 8d8bd6f

**This pull request can be automatically merged.**  
You can also merge branches on the [command line](#).

[Merge pull request](#)

**Merging via command line**  
If you do not want to use the merge button or an automatic merge cannot be performed, you can perform a manual merge on the command line.

HTTP Git Patch <https://github.com/miusuario/miniblog.git>

**Step 1:** From your project repository, check out a new branch and test the changes.

```
git checkout -b miusuario-add-license master
git pull https://github.com/miusuario/miniblog.git add-license
```

**Step 2:** Merge the changes and update on GitHub.

```
git checkout master
git merge miusuario-add-license
git push origin master
```

Labels: None yet

Milestone: No milestone

Assignee: No one assigned

Notifications: Unsubscribe

1 participant

Una vez que se han aceptado los cambios, podemos borrar la rama y actualizar nuestro repositorio con los datos del remoto como hicimos antes. ¿Por qué actualizar desde el remoto y no desde nuestra rama add-license? Pues porque usualmente el administrador puede haber modificado los cambios que le hemos propuesto, o incluso una tercera persona. Recordemos el cariz colaborativo que tiene Github.

```
1$ git checkout master
2$ git branch -d add-license
3# Esto borra la rama local
4$ git push origin --delete add-license
5# Esto borra la rama remota. También puede hacerse desde la web.
```

Todo esto es algo complicado...

Sí, lo es, al menos al principio. Git tiene una parte muy sencilla que es el uso del repositorio local (órdenes tales como add, rm, mv y commit). El siguiente nivel de complejidad lo componen las órdenes para trabajar con ramas y fusionarlas (checkout, branch, merge, rebase) y por último, las que trabajan con repositorios remotos (pull, push, fetch, remote). Además hay otra serie de órdenes para tener información (diff, log, status) o hacer operaciones de mantenimiento (fsck, gc). Lo importante para no perderse en Git, es seguir la siguiente máxima:

No avanzar al siguiente nivel de complejidad, hasta no haber entendido completamente el anterior.

Muy poco sentido tiene ponernos a crear ramas en github si aún no entendemos cómo se crean localmente y para que deben usarse. En la parte de referencias hay varios manuales en línea, incluso tutoriales interactivos. También hay mucha documentación disponible en Github que suele venir muy bien explicada. En caso de que tengamos un problema que no sepamos resolver, una web muy buena es [StackOverflow](https://stackoverflow.com/). Es una web de preguntas y respuestas para profesionales; es muy difícil que se os plantee una duda que no haya sido ya preguntada y respondida en esa web. Eso sí, el inglés es imprescindible.

## Último paso, documentación.

Github permite crear documentación. En primer lugar, generando un archivo llamado `README.md`. También permite crear una web propia para el proyecto y, además, una wiki. Para marcar el texto, se utiliza un lenguaje de marcado de texto denominado Markdown. En la siguiente web hay un tutorial interactivo:

<http://www.markdowntutorial.com/>. Como en principio, no es necesario saber Markdown para poder trabajar con Git o con Github, no vamos a incidir más en este asunto.

En el propio GitHub podemos encontrar algunas plantillas que nos sirvan de referencia.

Algunos ejemplos:

- [Plantilla básica](#)
- [Plantilla avanzada](#)

### Documentación del curso

Esta documentación está hecha en Markdown y pasada a HTML gracia a la herramienta [mkdocs](#). La plantilla usada es [Material for MkDocs](#).

El material está publicado con licencia [Atribución-NoComercial 4.0 Internacional \(CC BY-NC 4.0\)](#)

## Comandos de git

Esta sección describe algunos de los comandos más interesantes de git

### Git stash (reserva)

La orden `git stash` nos permite salvar momentáneamente el espacio de trabajo cuando tenemos que cambiar de rama o preparar la rama actual para sincronizar cambios.

Las operaciones más importantes que podemos hacer con `git stash` son:  
`git stash save`

Es equivalente a poner solo `git stash` pero nos permite realizar más acciones como:

```
1git stash save "Tu mensaje"
```

```
2git stash save -u
```

El parámetro `-u` permite que se almacenen también los ficheros sin seguimiento previo (untracked en inglés, aquellos ficheros que no se han metido nunca en el repositorio).

## git stash list

Permite mostrar la pila del stash.

```
1$ git stash list
```

```
2stash@{0}: On master: Stash con mensaje
```

```
3stash@{1}: WIP on master: 4ab21df First commit
```

## git stash apply

Esta orden coge el stash que está arriba en la pila y lo aplica al espacio de trabajo actual. En este caso siempre `@{0}`. El stash permanece en la pila.

Se puede indicar como parámetro un stash en concreto.

## git stash pop

Funciona igual que `git apply` con la diferencia de que el stash sí se borra de la pila.

## git stash show

Muestra un resumen de los ficheros que se han modificado en ese stash.

```
1$ git stash show
```

```
2A.txt | 1 +
```

```
3B.txt | 3 +++
```

```
42 file changed, 4 insertions(+)
```

Para ver los cambios podemos usar el parámetro `-p`

```
1$ git stash show -p
```

```
2--- a/A.txt
```

```
3+++ b/A.txt
```

```
4@@ -45,6 +45,7 @@ nav:
```

```
5+ This is a change
```

Por defecto siempre muestra la cabeza de la pila. Igual que en casos anteriores podemos indicar un stash en concreto.

```
1$ git stash show stash@{1}
```

## git stash branch

Permite crear una nueva rama a partir del último stash. Además, el mismo es borrado de la pila. Se puede especificar uno en concreto si lo queremos, como en el resto de comandos.

```
1git stash branch nombre-de-nueva-rama stash@{1}
```

## git stash clear

Este comando borra todos los stash de la pila. Es destructiva y no se puede deshacer.

## git stash drop

Permite borrar un stash en concreto (o el último si no se indica ninguno). Como con clear, borrarlo implica que no se puede recuperar.

## Git worktree

Uno de los problemas más habituales es tener que tocar una rama distinta a la que tenemos actualmente. Eso implica que si estamos en medio de un trabajo tendríamos que hacer un commit o un stash, lo cual a veces es bastante molesto.

Con `git worktree` podemos crear un directorio de trabajo que contenga otra rama distinta, de forma temporal. No supone otro clon del repositorio porque ambos usan el mismo.

### git worktree add

Esta función es la que crea el espacio de trabajo temporal. Imaginemos que estamos en una rama llamada `develop`:

```
1$ git worktree add ../project-master master
2$ git worktree add -b fix ../project-fix master
```

La primera orden crea un directorio llamado `project-master` que contiene el estado de `master`. La segunda, que contiene el parámetro `-b` equivale a crear una nueva rama llamada `fix`, que se crea desde `master` (suponemos que no existe `fix`).

### git worktree list

Muestra el listado de directorios y espacios de trabajo.

```
1$ git worktree list
2/home/sergio/taller-de-git 3b63b4b [master]
3/home/sergio/fix           3b63b4b [fix]
```

### git worktree remove

Borrar un espacio de trabajo. Hay que indicar el nombre entre corchetes que aparece en el listado

```
1$ git worktree delete fix
```

### git worktree prune

Una cuestión importante, es que las ramas que estén desplegadas en otro espacio de trabajo, se encuentran bloqueadas y no se pueden desbloquear en otro distinto.

Esto significa que si estamos trabajando en la rama `developer`, creamos otro `worktree` en otro directorio de la rama `master`, no podemos hacer pasar a `master`. No es posible tener la misma rama en varios espacios de trabajo.

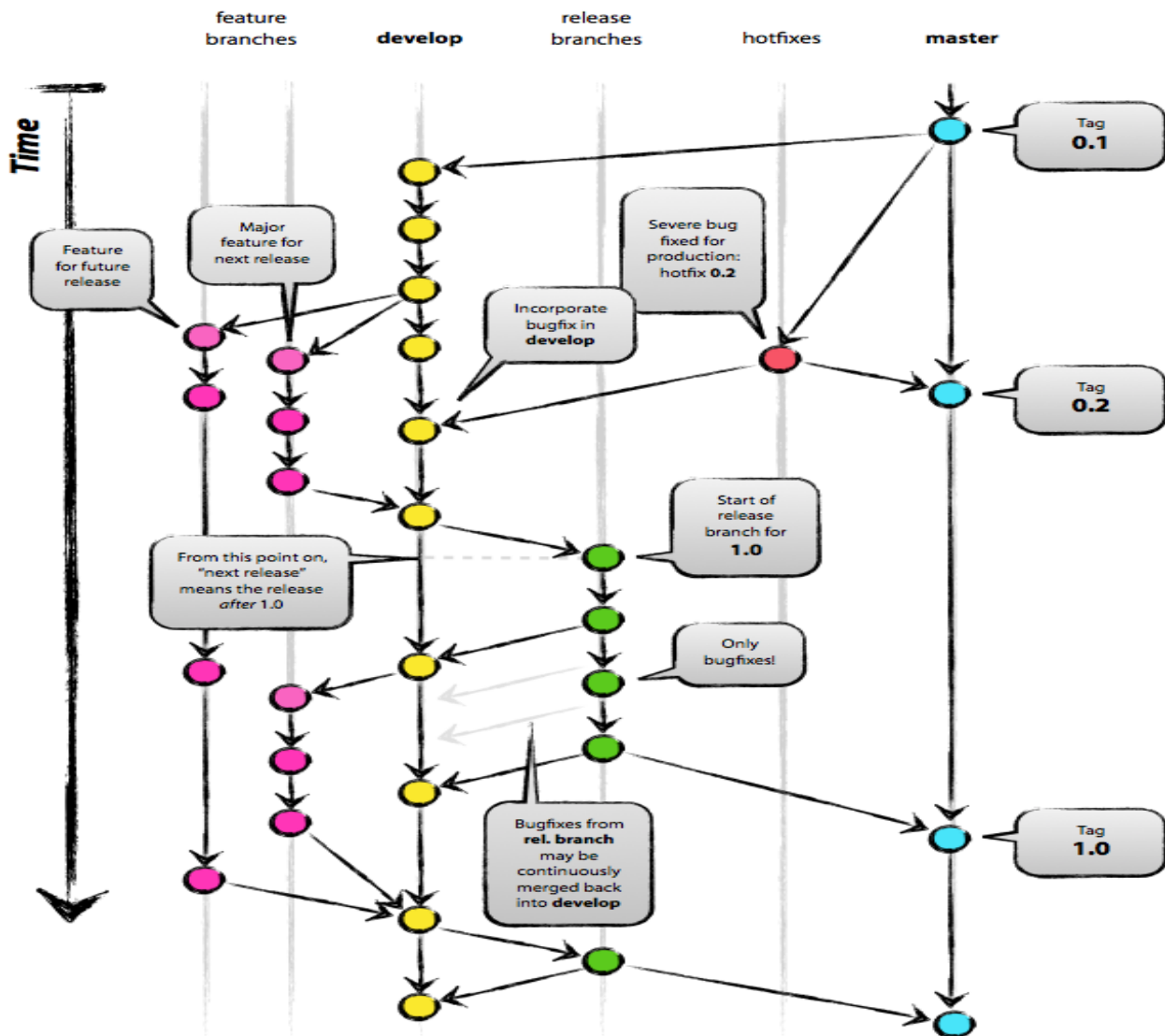
Si se ha borrado el directorio a mano (en vez de usando `remove`), eso no implica que el bloqueo desaparezca. Con esta orden podemos hacer que git compruebe que los espacios de trabajo secundario se comprueben de nuevo para ver si siguen existiendo y se elimine el bloqueo.



# Flujo de trabajo con Git (git flow)

## La importancia de la organización del flujo de trabajo

En la introducción vimos los diferentes esquemas de organización externa de los repositorios (es decir, en lo relativo a los usuarios que componen el equipo de trabajo). Pero el repositorio en sí también tiene su esquema de organización. En los ejemplos hemos visto que usábamos una rama máster y creábamos ramas para añadir funcionalidades que luego integrábamos. Es una forma de trabajar de las muchas que hay propuestas, posiblemente la más simple, pero tiene el inconveniente de dejar la rama máster a expensas de una mala actualización y quedarnos sin una rama estable. Por eso, hay otras propuestas mejores que permiten separar el trabajo de desarrollo con el mantenimiento de las versiones estables. Una de las más conocidas es la propuesta por [Vincent Driessen](#) y que podemos ver en la figura siguiente.



## Las ramas principales

En este esquema hay dos ramas principales con un tiempo de vida indefinido:

- master (origin/master): el código apuntado por HEAD siempre contiene un estado listo para producción.
- develop (origin/develop): el código apuntado por HEAD siempre contiene los últimos cambios desarrollados para la próxima versión del software. También se le puede llamar rama de integración. No es necesariamente estable.

Cuando el código de la rama de desarrollo es lo suficientemente estable, se integra con la rama master y una nueva versión es lanzada.

## Las ramas auxiliares

Para labores concretas, pueden usarse otro tipo de ramas, las cuales tienen un tiempo de vida definido. Es decir, cuando ya no son necesarias se eliminan:

- Ramas de funcionalidad (feature branches)
- Ramas de versión (release branches)
- Ramas de parches (hotfix branches)

### Feature branches

- Pueden partir de: develop
- Deben fusionarse con: develop
- Convenición de nombres: feature-NUMissue-\*

### Release branches

- Pueden partir de: develop
- Deben fusionarse con: develop y master
- Convenición de nombres: release-\*

### Hotfix branches

- Pueden partir de: master
- Deben fusionarse con: develop y master
- Convenición de nombres: hotfix-\*

## La extensión flow de Git

Una de las ventajas de Git es que, además, es extensible. Es decir, se pueden crear nuevas órdenes como si de plugins se tratara. Una de las más usadas es [gitflow](#), que está basada en el artículo que hablamos al principio de este capítulo.

### Instalación

Aunque la fuente original de la extensión es del mismo autor del artículo, el código no se encuentra ya muy actualizado y hay un fork bastante más activo en [petervanderdoes/gitflow](#). En el wiki del repositorio están las [instrucciones de instalación](#) para distintos sistemas. Una vez instalados tendremos una nueva orden:

```
git flow.
```

## Uso

Para cambiar a las ramas master y develop, seguiremos usando `git checkout`, pero para trabajar con las ramas antes indicadas gitflow nos facilita las siguientes órdenes:

- `git flow init`: Permite inicializar el espacio de trabajo.
- `git flow feature`: Permite crear y trabajar con ramas de funcionalidades.
- `git flow release`: Permite crear y trabajar con ramas de versiones.
- `git flow hotfix`: Permite crear y trabajar con ramas de parches.

En el siguiente capítulo veremos como usarlas para trabajar en un proyecto ya creado.

## Referencias

- [Documentación oficial en inglés](#).
- [Documentación oficial en español \(quizás incompleta\)](#).
- [Curso de Git \(inglés\)](#). La mayoría de la documentación de este manual está basada en este curso.
- [Curso interactivo de Git \(inglés\)](#).
- [Página de referencia de todas las órdenes de Git \(inglés\)](#).
- [Chuleta con las órdenes más usuales de Git](#).
- [Gitmagic \(ingles y español\)](#). Otro manual de Git
- [Artículo técnico: Un modelo exitoso de ramificación en Git](#).
- [Curso detallado y gratuito sobre Git y github](#)
- [Otra guia rápida de git](#)
- [Guía de estilos según Udacity](#)