

Inicio

Atención

Los contenidos del **Tema 5 - Control de versiones** están basados íntegramente en el curso que se menciona a continuación.

Su licencia original permite la reutilización y difusión del mismo.



Este taller forma parte de las actividades del [Aula de Software Libre de la Universidad de Córdoba](#).

El contenido del mismo es en parte de producción propia, en parte de otros manuales libres que pueden encontrarse en la sección de [Referencias](#).

Contenido

- Inicio
- Sistemas de control de versiones
- Introducción a Git
- Aspectos básicos de Git
- Uso básico
- Uso avanzado
- Ramas
- Administración de repositorios
- Flujo de trabajo con Git (git flow)
- Github
- Referencias

Agradecimientos

Este curso ha sido impartido por las siguientes personas:

- [Adrián López](#)
- [Héctor Romero](#)
- [Javier de Santiago](#)
- [José Márquez](#)
- [Sergio Gómez](#)

Licencia

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

Sistemas de control de versiones

Definición, clasificación y funcionamiento

Se llama control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo. Una versión, revisión o edición de un producto, es el estado en el que se encuentra dicho producto en un momento dado de su desarrollo o modificación. Aunque un sistema de control de versiones puede realizarse de forma manual, es muy aconsejable disponer de herramientas que faciliten esta gestión dando lugar a los llamados sistemas de control de versiones o SVC (del inglés System Version Control).

Estos sistemas facilitan la administración de las distintas versiones de cada producto desarrollado, así como las posibles especializaciones realizadas (por ejemplo, para algún cliente específico). Ejemplos de este tipo de herramientas son entre otros: CVS, Subversion, SourceSafe, ClearCase, Darcs, Bazaar , Plastic SCM, Git, Mercurial, Perforce.

Terminología

Repositorio ("repository") : El repositorio es el lugar en el que se almacenan los datos actualizados e históricos de cambios.

Revisión ("revision") : Una revisión es una versión determinada de la información que se gestiona. Hay sistemas que identifican las revisiones con un contador (Ej. subversion). Hay otros sistemas que identifican las revisiones mediante un código de detección de modificaciones (Ej. git usa SHA1).

Etiqueta ("tag") : Los tags permiten identificar de forma fácil revisiones importantes en el proyecto. Por ejemplo se suelen usar tags para identificar el contenido de las versiones publicadas del proyecto.

Rama ("branch") : Un conjunto de archivos puede ser ramificado o bifurcado en un punto en el tiempo de manera que, a partir de ese momento, dos copias de esos archivos se pueden desarrollar a velocidades diferentes o en formas diferentes de forma independiente el uno del otro.

Cambio ("change") : Un cambio (o diff, o delta) representa una modificación específica de un documento bajo el control de versiones. La granularidad de la modificación que es considerada como un cambio varía entre los sistemas de control de versiones.

Desplegar ("checkout") : Es crear una copia de trabajo local desde el repositorio. Un usuario puede especificar una revisión en concreto u obtener la última. El término 'checkout' también se puede utilizar como un sustantivo para describir la copia de trabajo.

Confirmar ("commit") : Confirmar es escribir o mezclar los cambios realizados en la copia de trabajo del repositorio. Los términos 'commit' y 'checkin' también se pueden utilizar como sustantivos para describir la nueva revisión que se crea como resultado de confirmar.

Conflicto ("conflict") : Un conflicto se produce cuando diferentes partes realizan cambios en el mismo documento, y el sistema es incapaz de conciliar los cambios. Un usuario debe resolver el conflicto mediante la integración de los cambios, o mediante la selección de un cambio en favor del otro.

Cabeza ("head") : También a veces se llama tip (punta) y se refiere a la última confirmación, ya sea en el tronco ('trunk') o en una rama ('branch'). El tronco y cada rama tienen su propia cabeza, aunque HEAD se utiliza a veces libremente para referirse al tronco.

Tronco ("trunk") : La única línea de desarrollo que no es una rama (a veces también llamada línea base, línea principal o máster).

Fusionar, integrar, mezclar ("merge") : Una fusión o integración es una operación en la que se aplican dos tipos de cambios en un archivo o conjunto de archivos. Algunos escenarios de ejemplo son los siguientes:

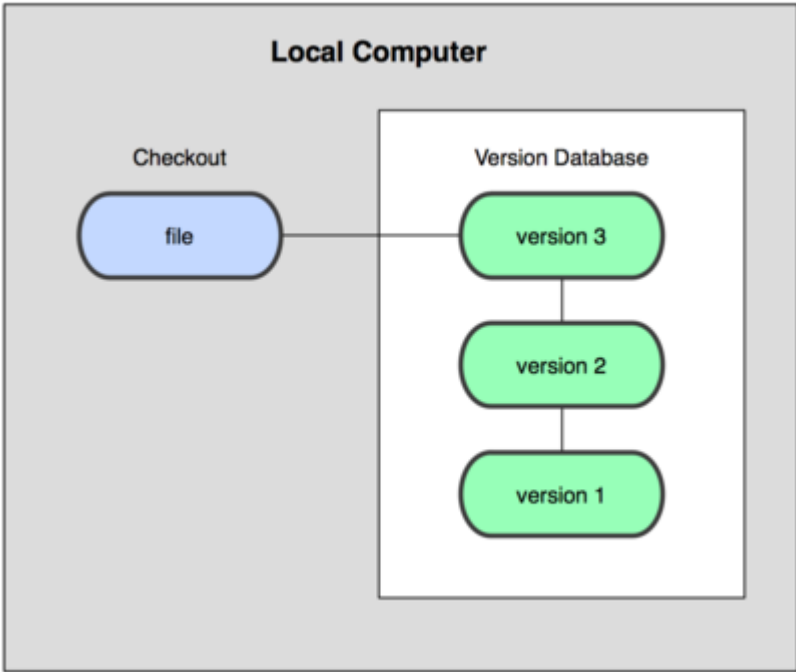
- Un usuario, trabajando en un conjunto de archivos, actualiza o sincroniza su copia de trabajo con los cambios realizados y confirmados, por otros usuarios, en el repositorio.
- Un usuario intenta confirmar archivos que han sido actualizado por otros usuarios desde el último despliegue ('checkout'), y el software de control de versiones integra automáticamente los archivos (por lo general, después de preguntarle al usuario si se debe proceder con la integración automática, y en algunos casos sólo se hace si la fusión puede ser clara y razonablemente resuelta).
- Un conjunto de archivos se bifurca, un problema que existía antes de la ramificación se trabaja en una nueva rama, y la solución se combina luego en la otra rama.
- Se crea una rama, el código de los archivos es independiente editado, y la rama actualizada se incorpora más tarde en un único tronco unificado.

Clasificación

Podemos clasificar los sistemas de control de versiones atendiendo a la arquitectura utilizada para el almacenamiento del código: locales, centralizados y distribuidos.

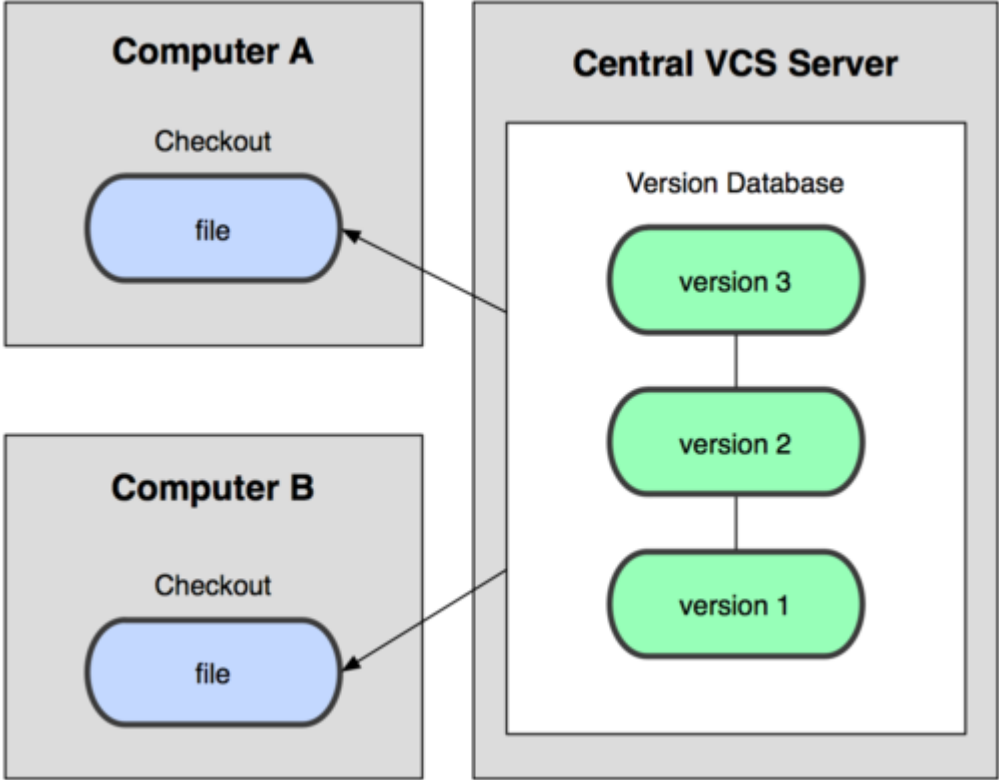
Locales

Los cambios son guardados localmente y no se comparten con nadie. Esta arquitectura es la antecesora de las dos siguientes.



Centralizados

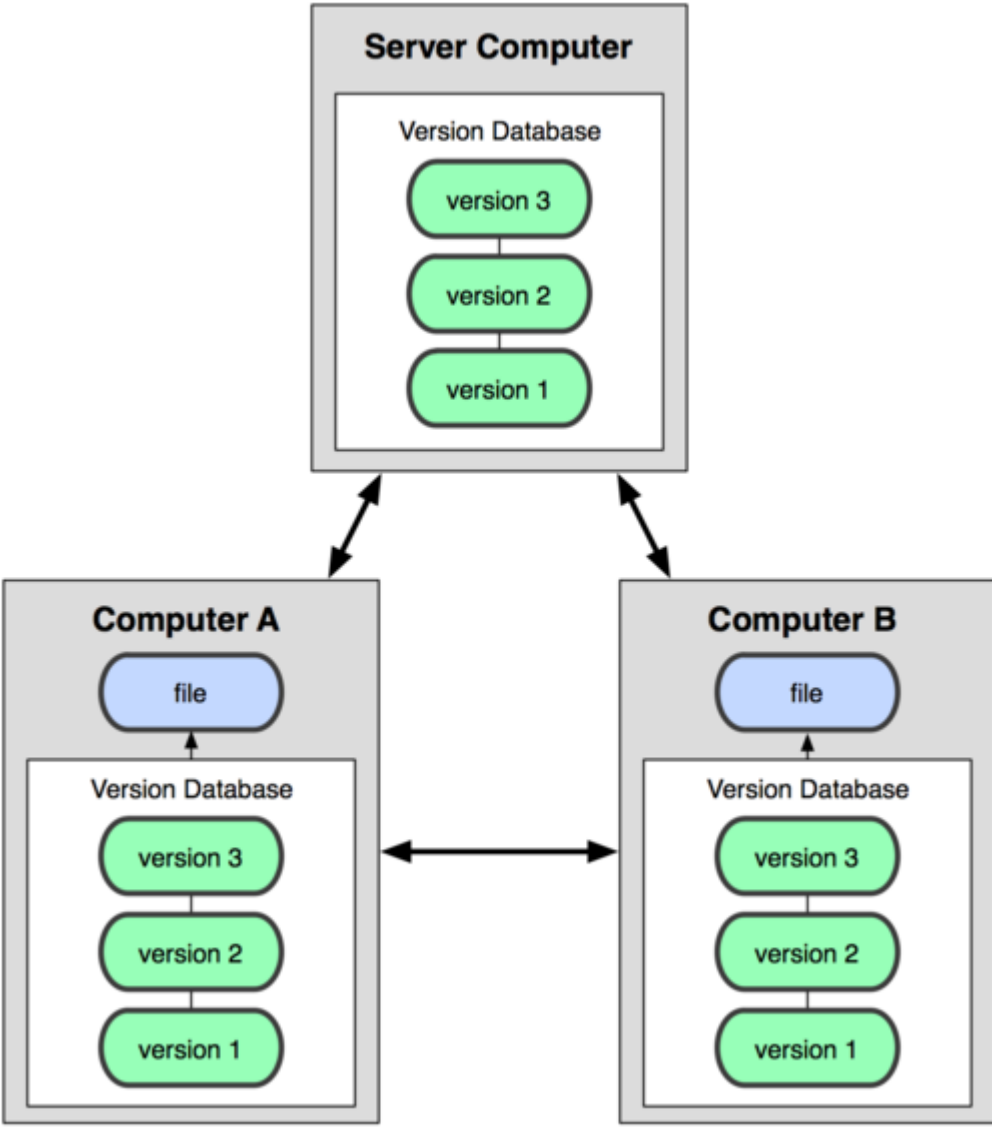
Existe un repositorio centralizado de todo el código, del cual es responsable un único usuario (o conjunto de ellos). Se facilitan las tareas administrativas a cambio de reducir flexibilidad, pues todas las decisiones fuertes (como crear una nueva rama) necesitan la aprobación del responsable. Algunos ejemplos son CVS y Subversion.



Distribuidos

Cada usuario tiene su propio repositorio. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Es frecuente el uso de un repositorio, que está normalmente disponible que sirve de punto de sincronización de los distintos repositorios locales. Ejemplos: Git y Mercurial

permanentemente disponible, que sirve de punto de sincronización de los distintos repositorios locales. Ejemplos: SVN y Mercurial.

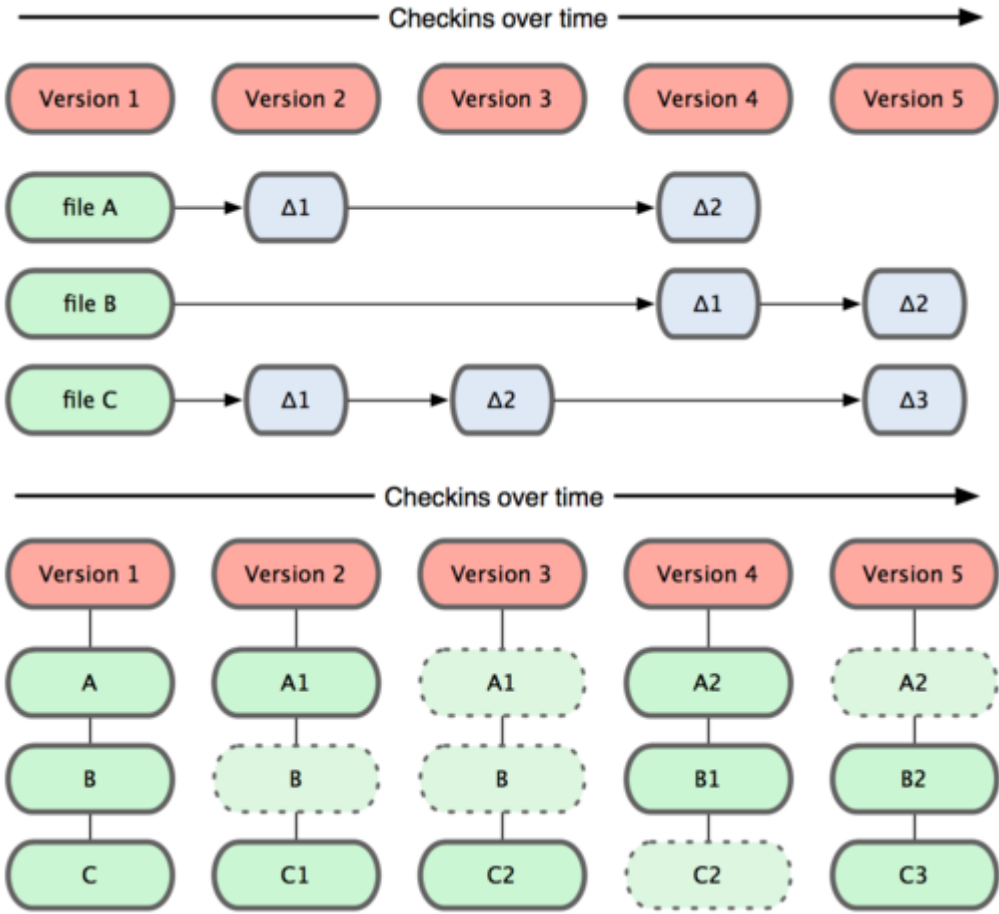


Ventajas de sistemas distribuidos

- No es necesario estar conectado para guardar cambios.
- Posibilidad de continuar trabajando si el repositorio remoto no está accesible.
- El repositorio central está más libre de ramas de pruebas.
- Se necesitan menos recursos para el repositorio remoto.
- Más flexibles al permitir gestionar cada repositorio personal como se quiera.

Introducción a git

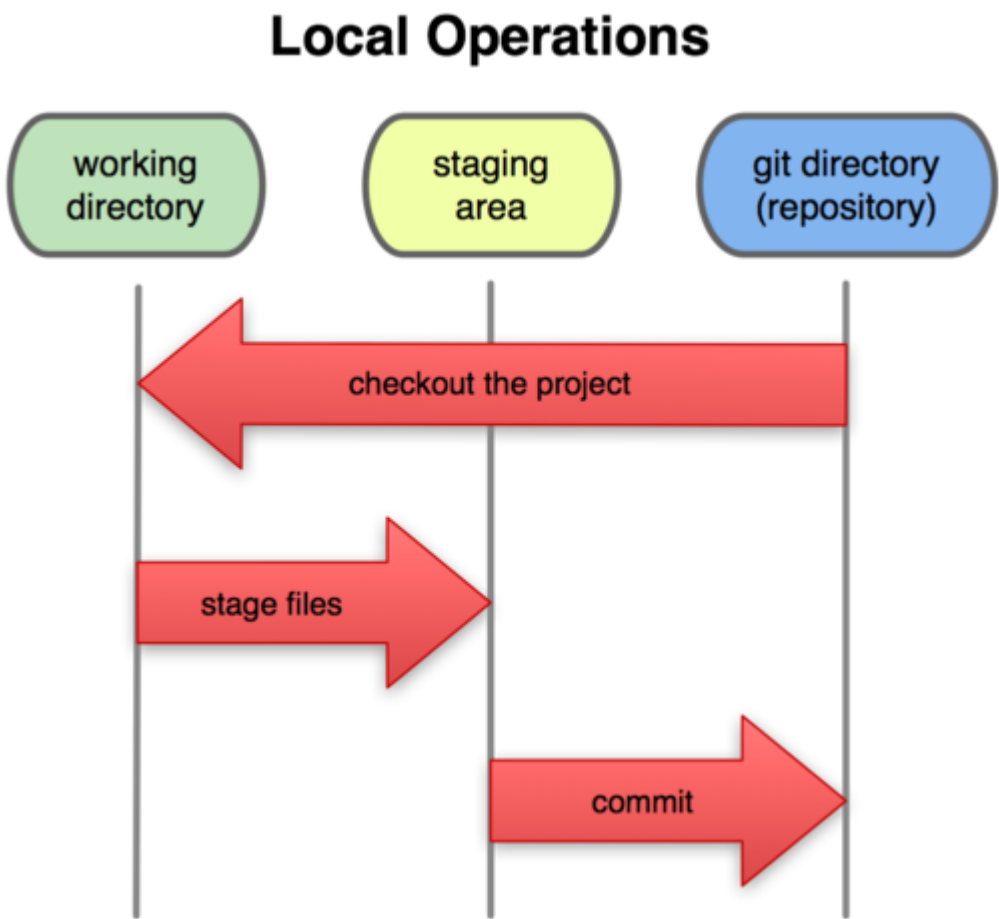
Git es un sistema de control de versiones distribuido que se diferencia del resto en el modo en que modela sus datos. La mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos, mientras que Git modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos.



Los tres estados

Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged). Confirmado significa que los datos están almacenados de manera segura en tu base de datos local. Modificado significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos. Preparado significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git: el directorio de Git (Git directory), el directorio de trabajo (working directory), y el área de preparación (staging area).

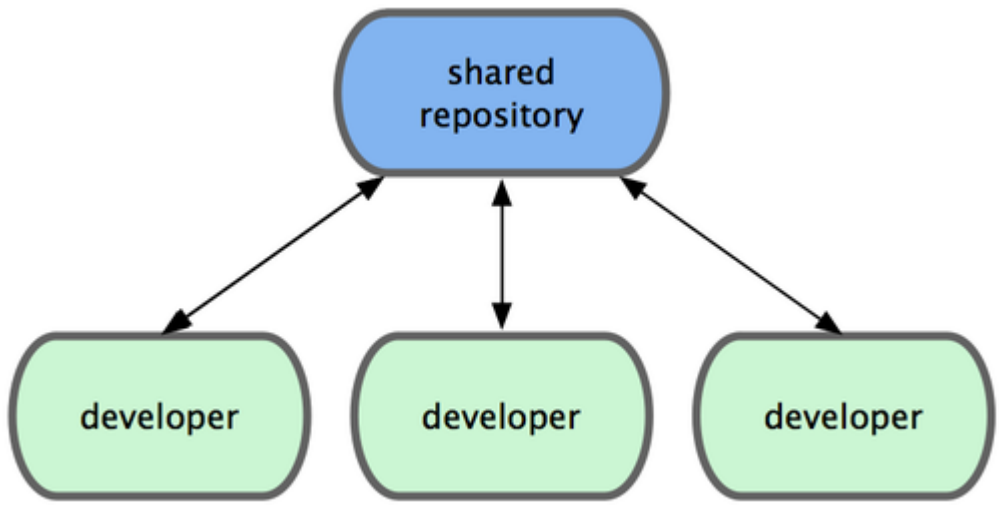


Flujos de trabajo distribuidos con git

Hemos visto en qué consiste un entorno de control de versiones distribuido, pero más allá de la simple definición, existe más de una manera de gestionar los repositorios. Estos son los flujos de trabajo más comunes en Git.

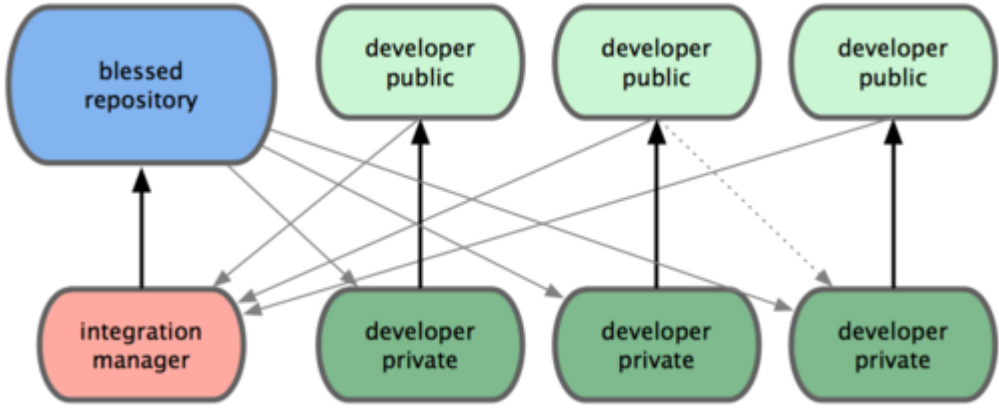
Flujo de trabajo centralizado

Existe un único repositorio o punto central que guarda el código y todo el mundo sincroniza su trabajo con él. Si dos desarrolladores clonan desde el punto central, y ambos hacen cambios; tan solo el primero de ellos en enviar sus cambios de vuelta lo podrá hacer limpiamente. El segundo desarrollador deberá fusionar previamente su trabajo con el del primero, antes de enviarlo, para evitar el sobrescribir los cambios del primero



Flujo de trabajo del Gestor-de-Integraciones

Al permitir múltiples repositorios remotos, en Git es posible tener un flujo de trabajo donde cada desarrollador tenga acceso de escritura a su propio repositorio público y acceso de lectura a los repositorios de todos los demás. Habitualmente, este escenario suele incluir un repositorio canónico, representante "oficial" del proyecto.

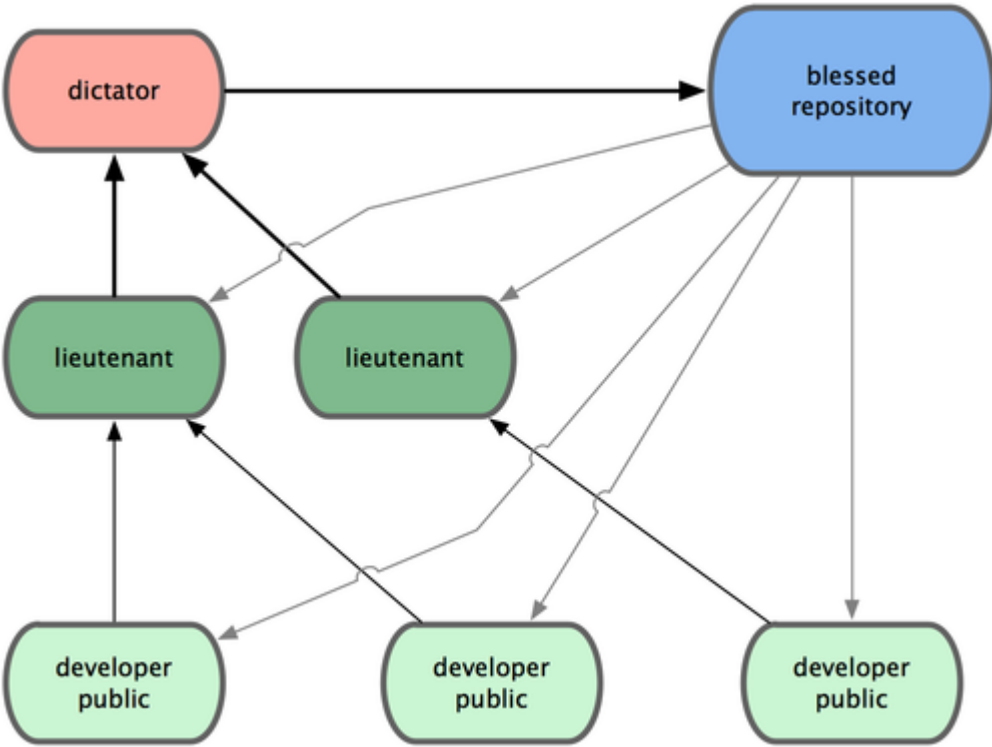


Info

Este modelo se puso muy de moda a raíz de la forja GitHub que se verá más adelante.

Flujo de trabajo con Dictador y Tenientes

Es una variante del flujo de trabajo con múltiples repositorios. Se utiliza generalmente en proyectos muy grandes, con cientos de colaboradores. Un ejemplo muy conocido es el del kernel de Linux. Unos gestores de integración se encargan de partes concretas del repositorio; y se denominan tenientes. Todos los tenientes rinden cuentas a un gestor de integración; conocido como el dictador benevolente. El repositorio del dictador benevolente es el repositorio de referencia, del que recuperan (pull) todos los colaboradores.





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:

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

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

```
$ 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:

```
$ 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:

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

También se recomienda configurar el siguiente parámetro:

```
$ 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"

Creamos un directorio donde colocar el código

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

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

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

Crear el repositorio

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

```
$ git init
Initialized empty Git repository in /home/cc@gobas/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.

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

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

Si modificamos el archivo `hola.php` :

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

Y volvemos a comprobar el estado del repositorio:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   hola.php
#
no 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.

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

Confirmar los cambios

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

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

Diferencias entre *workdir* y *staging*.

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

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

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

```
git add hola.php
```

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

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

Y vemos el estado en el que está el repositorio


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



```
#
#   modified:   hola.php

#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   hola.php
#
```


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` sobrescribiríamos los cambios previos que había en la zona de *staging*.

Almacenamos los cambios por separado:

```
$ git commit -m "Se añade un parámetro por defecto"
[master 3283e0d] Se añade un parámetro por defecto
 1 file changed, 2 insertions(+), 1 deletion(-)
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   hola.php
#
no changes added to commit (use "git add" and/or "git commit -a")
$ git add .
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   hola.php
#
$ git commit -m "Se añade un comentario al cambio del valor por defecto"
[master fd4da94] Se añade un comentario al cambio del valor por defecto
 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:

```
# .gitignore
dir1/          # ignora todo lo que contenga el directorio dir1
!dir1/info.txt # El operador ! excluye del ignore a dir1/info.txt (sí se guardaría)
dir2/*.txt     # ignora todos los archivos txt que hay en el directorio dir2
dir3/**/*.txt # ignora todos los archivos txt que hay en el dir3 y sus subdirectorios
*.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:

```
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:

```
# Compiled source #
#####
*.com
*.class
*.dll
*.exe
*.o
*.so

# Packages #
#####
# it's better to unpack these files and commit the raw source
# git has its own built in compression methods
*.7z
*.dmg
*.gz
*.iso
*.jar
*.rar
*.tar
*.zip

# Logs and databases #
#####
*.log
*.sql
*.sqlite

# OS generated files #
#####
.DS_Store
.DS_Store?
.*
.Spotlight-V100
.Trashes
```



```
ehthumbs.db
Thumbs.db
*~
*.swp

# IDEs
#####
.idea
.settings/
.classpath
.project
```

Trabajando con el historial

Observando los cambios

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

```
$ git log
commit fd4da946326f8b24e89282ad25a71721bf40f6
Author: Sergio Gómez <sergio@uco.es>
Date:   Sun Jun 16 12:51:01 2013 +0200

    Se añade un comentario al cambio del valor por defecto

commit 3283e0d306c8d42d55ffcb64e456f10510df8177
Author: Sergio Gómez <sergio@uco.es>
Date:   Sun Jun 16 12:50:00 2013 +0200

    Se añade un parámetro por defecto

commit efc252e11939351505a426a6e1aa5bb7dc1dd7c0
Author: Sergio Gómez <sergio@uco.es>
Date:   Sun Jun 16 12:13:26 2013 +0200

    Parametrización del programa

commit e19f2c1701069d9d1159e9ee21acaa1bbc47d264
Author: Sergio Gómez <sergio@uco.es>
Date:   Sun Jun 16 11:55:23 2013 +0200

    Creación del proyecto
```

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

```
$ git log --oneline
fd4da94 Se añade un comentario al cambio del valor por defecto
3283e0d Se añade un parámetro por defecto
efc252e Parametrización del programa
e19f2c1 Creación del proyecto
git log --oneline --max-count=2
git log --oneline --since='5 minutes ago'
git log --oneline --until='5 minutes ago'
git log --oneline --author=sergio
git 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
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (HEAD, master) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* 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:

```
git 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:

```
git config --global alias.br branch
git config --global alias.co checkout
git config --global alias.ci commit
git config --global alias.st "status -u"
git 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`.

```
$ git checkout e19f2c1
Note: checking out 'e19f2c1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b new_branch_name

HEAD is now at e19f2c1... Creación del proyecto
$ cat hola.php
<?php
echo "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 "perderse" 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:

```
$ git checkout master
Previous 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`.

```
$ 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:

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

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

```
$ git tag
v1
v1-beta
```

Y para verlas en el historial:

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

Borrar etiquetas

Para borrar etiquetas:

```
git 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:

```
$ git diff v1-beta v1
diff --git a/hola.php b/hola.php
index a31e01f..25a35c0 100644
--- a/hola.php
+++ b/hola.php
@@ -1,3 +1,4 @@
 <?php
+// El nombre por defecto es Mundo
 $nombre = isset($argv[1]) ? $argv[1] : "Mundo";
 @print "Hola, {$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:

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

Modificamos *hola.php* de la siguiente manera:

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

Y comprobamos:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#    modified:   hola.php
#
no 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:

```
$ git checkout hola.php
$ git status
# On branch master
nothing to commit, working directory clean
$ cat hola.php
<?php
// El nombre por defecto es Mundo
$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
@print "Hola, {$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:

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

Y lo añadimos al *staging*

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

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

```
$ git reset HEAD hola.php
Unstaged changes after reset:
M   hola.php
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#    modified:   hola.php
#
no changes added to commit (use "git add" and/or "git commit -a")
$ 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:

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

Pero ahora sí hacemos commit:

```
$ git add hola.php
$ git commit -m "Ups... este commit está mal."
master 5a5d067] Ups... este commit está mal
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
[master 817407b] Revert "Ups... este commit está mal"
1 file changed, 1 insertion(+), 1 deletion(-)
$ git hist
* 817407b 2013-06-16 | Revert "Ups... este commit está mal" (HEAD, master) [Sergio Gómez]
* 5a5d067 2013-06-16 | Ups... este commit está mal [Sergio Gómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* 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
HEAD is now at fd4da94 Se añade un comentario al cambio del valor por defecto
$ git hist
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (HEAD, tag: v1, master) [Sergio Góme
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* 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:

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

Y lo confirmamos:

```
$ git commit -a -m "Añadido el autor del programa"
[master cf405c1] Añadido el autor del programa
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:

```
<?php
// Autor: Sergio Gómez <sergio@uco.es>
// El nombre por defecto es Mundo
$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
@print "Hola, {$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
$ git commit --amend -m "Añadido el autor del programa y su email"
[master 96a39df] Añadido el autor del programa y su email
1 file changed, 1 insertion(+)
$ git hist
* 96a39df 2013-06-16 | Añadido el autor del programa y su email (HEAD, master) [Sergio Gómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* 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`:

```
$ mkdir lib
$ git mv hola.php lib
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   renamed:    hola.php -> lib/hola.php
#
```

Mover y borrar archivos.

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

```
$ mkdir lib
$ mv hola.php lib
$ git add lib/hola.php
$ git rm hola.php
```

Y, ahora sí, ya podemos guardar los cambios:

```
$ git commit -m "Movido hola.php a lib."
[master 8c2a509] Movido hola.php a lib.
1 file changed, 0 insertions(+), 0 deletions(-)
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:

```
git 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*:

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

 **Tip**

Hay una forma más rapida 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.

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

Modificaciones en la rama secundaria

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

```
<?php

class HolaMundo
{
    private $nombre;

    function __construct($nombre)
    {
        $this->nombre = $nombre;
    }

    function __toString()
    {
        return sprintf ("Hola, %s.\n", $this->nombre);
    }
}
```

Y modificamos `hola.php`:

```
<?php
// Autor: Sergio Gómez <sergio@uco.es>
// El nombre por defecto es Mundo
require('HolaMundo.php');

$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
print new HolaMundo($nombre);
```

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

```
$ git add lib/HolaMundo.php
$ git commit -m "Añadida la clase HolaMundo"
[hola 6932156] Añadida la clase HolaMundo
 1 file changed, 16 insertions(+)
 create mode 100644 lib/HolaMundo.php
$ git add lib/hola.php
$ git commit -m "hola usa la clase HolaMundo"
[hola 9862f33] hola usa la clase HolaMundo
 1 file changed, 3 insertions(+), 1 deletion(-)
```

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

```
$ git checkout master
Switched to branch 'master'
$ git checkout hola
Switched to branch 'hola'
```

Modificaciones en la rama master

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

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

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

```
# Curso de GIT

Este proyecto contiene el curso de introducción a GIT
```

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

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

```
$ git checkout hola
Switched to branch 'hola'
$ git merge master
Merge made by the 'recursive' strategy.
 README.md | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 README.md
$ git hist --all
* 9c6ac06 2013-06-16 | Merge commit 'c3e65d0' into hola (HEAD, hola) [Sergio Gómez]
|\
* | 9862f33 2013-06-16 | hola usa la clase HolaMundo [Sergio Gómez]
* | 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
| |
| * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
|/
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* 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.

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

Modificamos nuestro archivo *hola.php* de nuevo:

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

Y guardamos los cambios:

```
$ git add lib/hola.php
$ git commit -m "Programa interactivo"
[master 9c85275] Programa interactivo
 1 file changed, 2 insertions(+), 2 deletions(-)
$ git hist --all
* 9c6ac06 2013-06-16 | Merge commit 'c3e65d0' into hola (hola) [Sergio Gómez]
|\
* | 9862f33 2013-06-16 | hola usa la clase HolaMundo [Sergio Gómez]
* | 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
| | * 9c85275 2013-06-16 | Programa interactivo (HEAD, master) [Sergio Gómez]
| |
| | * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
|/
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Volvemos a la rama hola y fusionamos:

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

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

```
<?php
// Autor: Sergio Gómez <sergio@uco.es>
<<<<<< HEAD
// El nombre por defecto es Mundo
require('HolaMundo.php');

$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
print new HolaMundo($nombre);
=====
print "Introduce tu nombre:";
$nombre = trim(fgets(STDIN));
@print "Hola, {$nombre}\n";
>>>>>> 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:

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

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

Y resolvemos el conflicto confirmando los cambios:

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

```
$ git checkout hola
Switched to branch 'hola'
$ git hist
* a36af04 2013-06-16 | Solucionado el conflicto al fusionar con la rama master (HEAD, hola) [Sergio Gómez]
|\
```



```
| * 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
* | 9c6ac06 2013-06-16 | Merge commit 'c3e65d0' into hola [Sergio Gómez]

|\ \
|  | /
| * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
* | 9862f33 2013-06-16 | hola usa la clase HolaMundo [Sergio Gómez]
* | 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
| /
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
$ git reset --hard 9862f33
HEAD is now at 9862f33 hola usa la clase HolaMundo
```

Y nuestro estado será:

```
$ git hist --all
* 9862f33 2013-06-16 | hola usa la clase HolaMundo (HEAD, hola) [Sergio Gómez]
* 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
| * 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
| * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
| /
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* 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
First, rewinding head to replay your work on top of it...
Applying: Añadida la clase HolaMundo
Applying: hola usa la clase HolaMundo
Using index info to reconstruct a base tree...
M   lib/hola.php
Falling back to patching base and 3-way merge...
Auto-merging lib/hola.php
CONFLICT (content): Merge conflict in lib/hola.php
error: Failed to merge in the changes.
Patch failed at 0002 hola usa la clase HolaMundo
The copy of the patch that failed is found in: .git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
```

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

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

print "Introduce tu nombre:";
$nombre = trim(fgets(STDIN));
print new HolaMundo($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*:

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

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   lib/hola.php
$ git rebase --continue
Applying: hola usa la clase HolaMundo
```

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

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

```
$ git checkout master
Switched to branch 'master'
$ git merge hola
Updating c3e65d0..491f1d2
Fast-forward
 lib/HolaMundo.php | 16 ++++++
 lib/hola.php       |  4 +++
 2 files changed, 19 insertions(+), 1 deletion(-)
 create mode 100644 lib/HolaMundo.php
$ git hist --all
* 9862f33 2013-06-16 | hola usa la clase HolaMundo (HEAD -> master, hola) [Sergio Gómez]
* 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
* 9c85275 2013-06-16 | Programa interactivo [Sergio Gómez]
* c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* 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*. Resetearnos *master* al estado *"Programa interactivo"*.

```
$ git reset --hard 9c85275
$ git hist --all
```

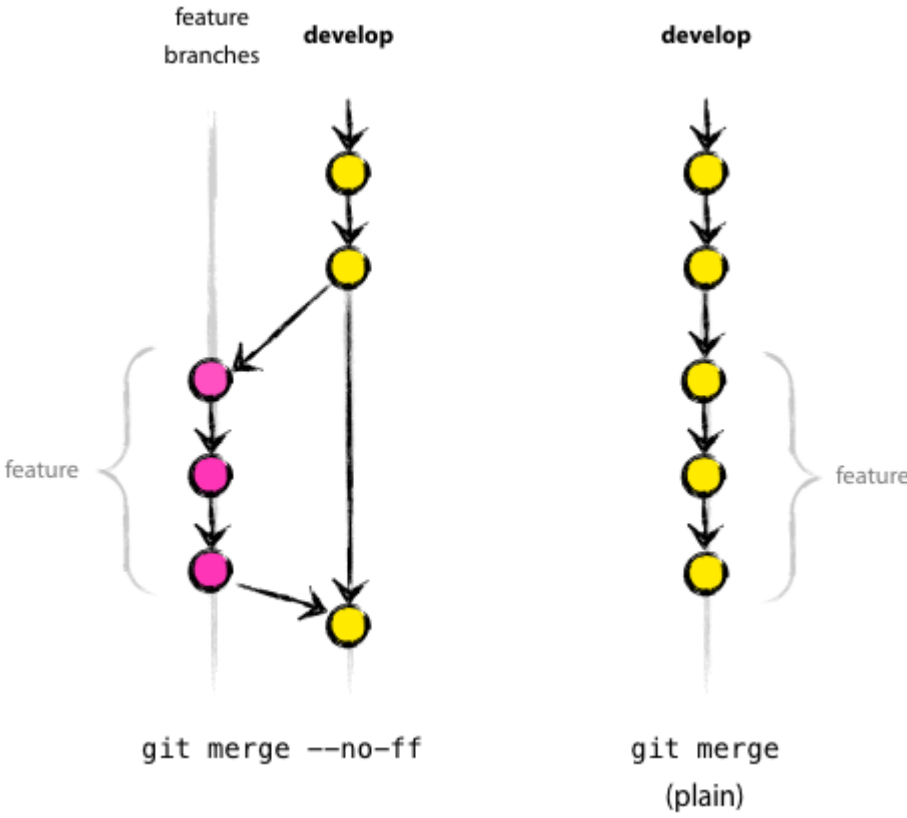
```
* 9862f33 2013-06-16 | hola usa la clase HolaMundo (HEAD -> hola) [Sergio Gómez]
* 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]

* 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
* c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

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

```
$ git merge -m "Aplicando los cambios de la rama hola" --no-ff hola
Merge made by the 'recursive' strategy.
 lib/HolaMundo.php | 16 ++++++
 lib/hola.php       |  4 +++-
 2 files changed, 19 insertions(+), 1 deletion(-)
 create mode 100644 lib/HolaMundo.php
$ git hist --all
*    2eab8ca 2013-06-16 | Aplicando los cambios de la rama hola (HEAD -> master) [Sergio Gomez]
*\
| * 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Gómez]
| * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
|/
* 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
* c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* 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:

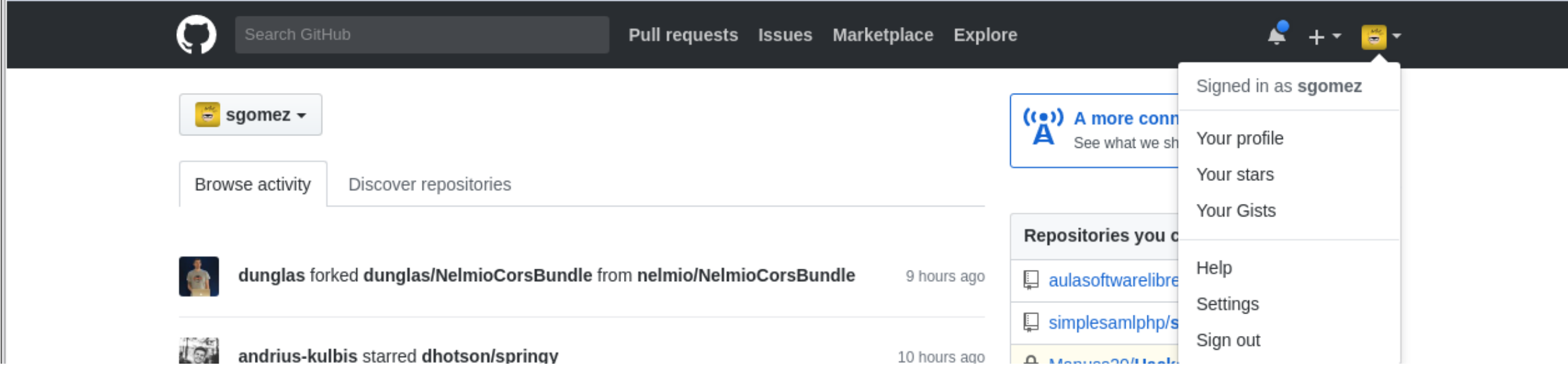
```
$ ssh-keygen -t rsa -C "Cuenta Thinstation"
```

Warning

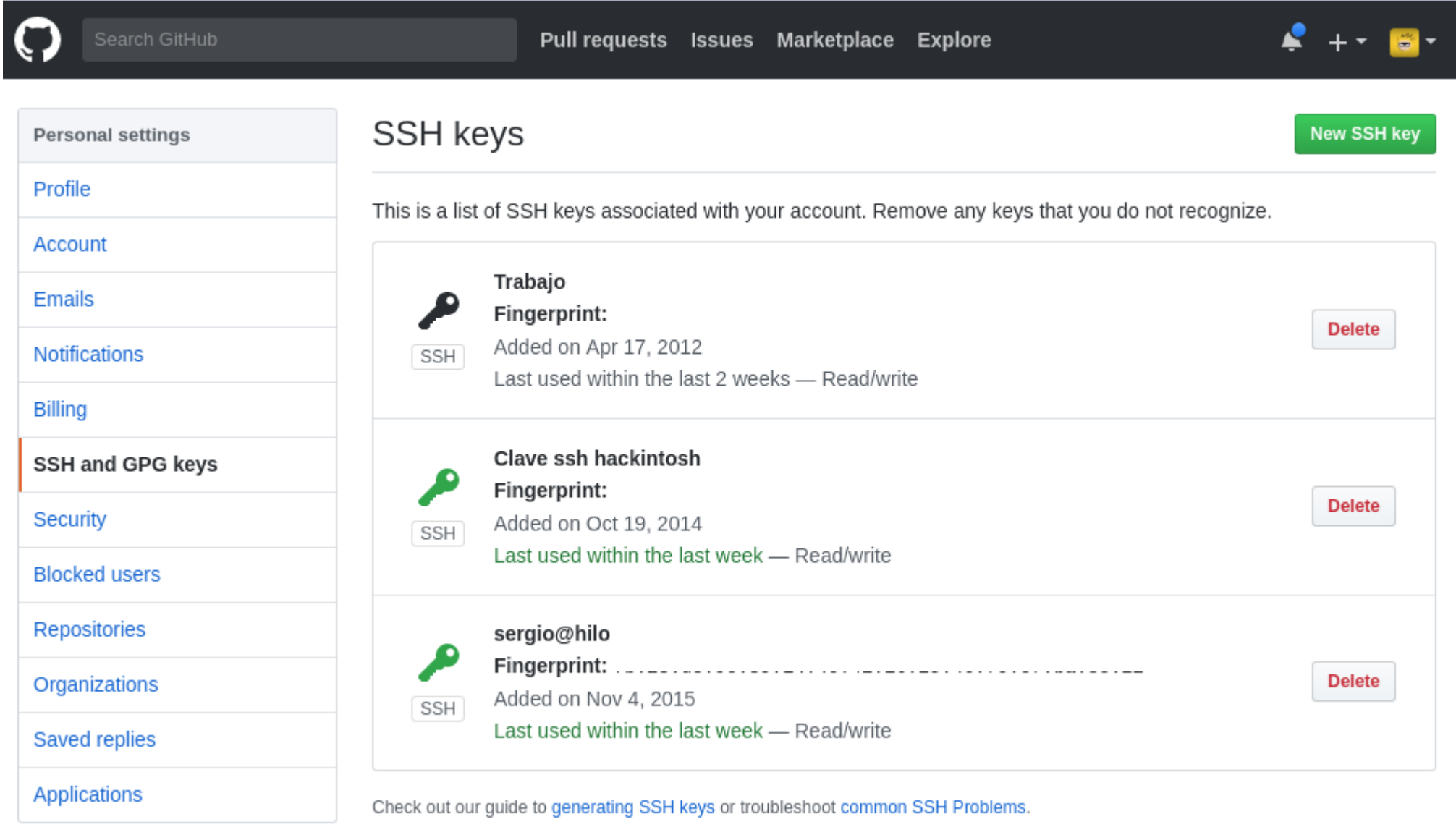
Tu clave RSA te identifica contra los repositorios remotos, asegúrate de no compartir la clave privada con nadie. Por defecto la clave se crea como *solo lectura*.

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'`.

[Pull requests](#) [Issues](#) [Marketplace](#) [Explore](#)

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.

This repository

[Pull requests](#) [Issues](#) [Marketplace](#) [Explore](#)

sgomez / **taller-de-git**

Unwatch ▾

1

★ Star

0

Fork

0

Code

Issues **0**

Pull requests **0**

Projects **0**

Wiki

Insights

Settings

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

or **HTTPS** **SSH**

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.

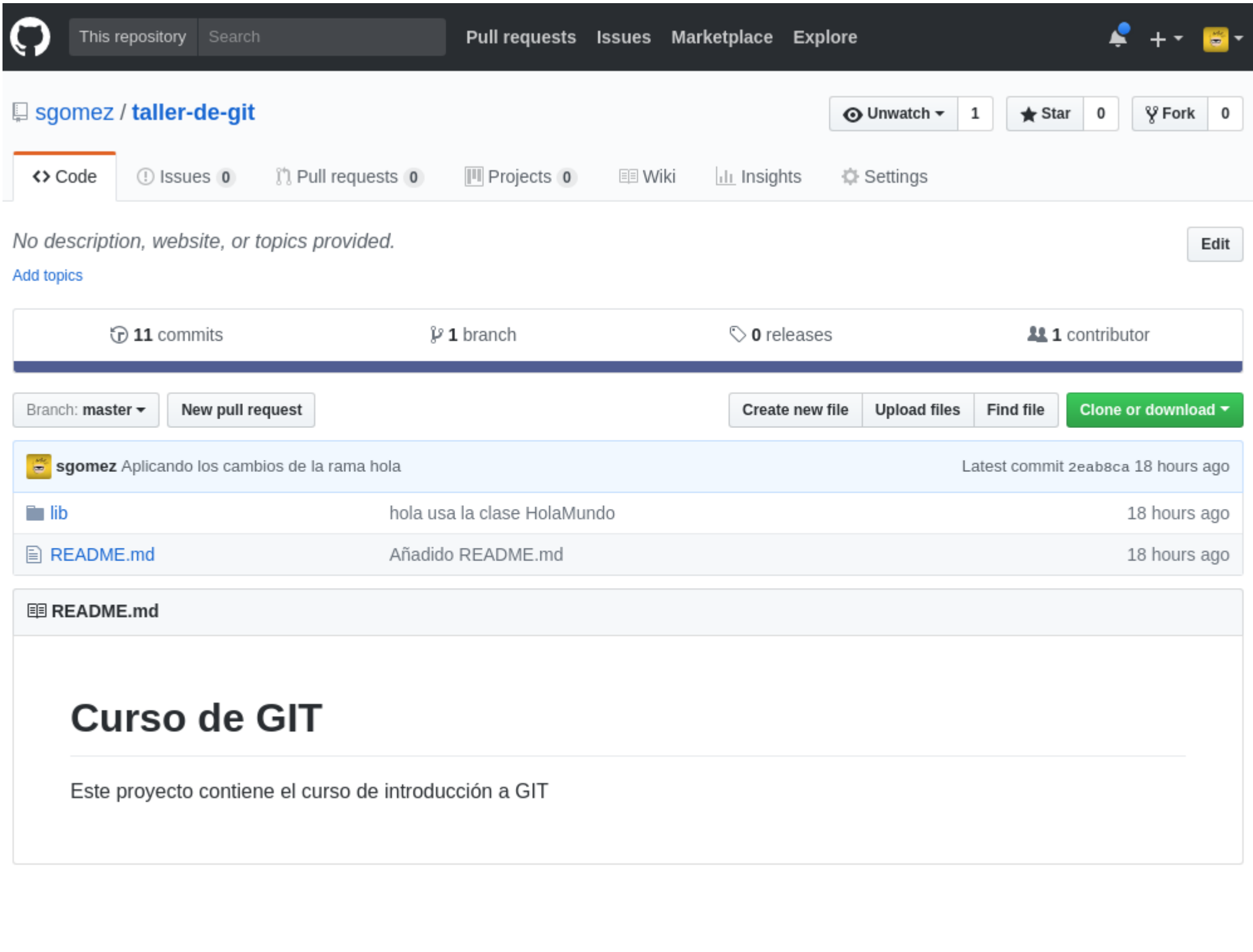
Import code

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.

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

Si recargamos la página veremos que ya aparece nuestro proyecto.



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 para 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:

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

Ramas remotas

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

```
$ git hist --all
* 2eab8ca 2013-06-16 | Aplicando los cambios de la rama hola (HEAD -> master, origin/master) [Sergio Gomez]
*\
| * 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Gómez]
| * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
|/
* 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
* c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* 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`:

```
$ git remote show origin
* remote origin
  Fetch URL: git@github.com:sgomez/taller-de-git.git
  Push URL: git@github.com:sgomez/taller-de-git.git
  HEAD branch: master
  Remote branch:
    master tracked
  Local ref configured for 'git push':
    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:

MIT License

Copyright (c) [year] [fullname]

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Y añadidos y confirmamos los cambios:

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

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

```
$ 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.

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

Si repasas las órdenes que te indicé 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.

4 lines (2 sloc) | 71 Bytes

RawBlameHistory

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:

```
# Curso de GIT

Este proyecto contiene el curso de introducción a GIT

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



```
> git merge origin/master
Updating 3f5cb1c..cbaf831
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)
$ git hist --all
* cbaf831 2013-06-16 | Actualizado README.md (HEAD -> master, origin/master) [Sergio Gómez]
* 3f5cb1c 2013-06-16 | Añadida licencia [Sergio Gómez]
* 2eab8ca 2013-06-16 | Aplicando los cambios de la rama hola [Sergio Gomez]
*\
| * 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Gómez]
| * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
|/
* 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
* c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* 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 pull` que realiza las dos acciones simultáneamente.

Para probar, vamos a editar de nuevo el archivo `README.md` y añadimos algo más:

```
# Curso de GIT

Este proyecto contiene el curso de introducción a GIT del Aula de Software Libre.

Desarrollado por Sergio Gómez.
```

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

Y ahora probamos a actualizar con `git pull`:

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:sgomez/taller-de-git
 cbaf831..d8922e4  master    -> origin/master
First, rewinding head to replay your work on top of it...
Fast-forwarded master to d8922e4ffa4f87553b03e77df6196b7e496bfec4.
$ git hist --all
* d8922e4 2013-06-16 | Indicado que se realiza en el ASL (HEAD -> master, origin/master) [Sergio Gómez]
* cbaf831 2013-06-16 | Actualizado README.md [Sergio Gómez]
* 3f5cb1c 2013-06-16 | Añadida licencia [Sergio Gómez]
* 2eab8ca 2013-06-16 | Aplicando los cambios de la rama hola [Sergio Gomez]
*\
| * 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Gómez]
| * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
|/
* 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
* c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* 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:

```
$ git push
git push
To git@github.com:sgomez/taller-de-git.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git@github.com:sgomez/taller-de-git.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: 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.

```
Curso de GIT, 2020
```

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

```
Curso de GIT, febrero
```

? Question

Haz el commit para guardar el cambio en local.

📖 Respuesta al ejercicio anterior

Añadimos el fichero actualizado:

```
$ git commit -am "Añadido el mes al README"
[master 1e8c0b7] Añadido el mes al README
1 file changed, 1 insertion(+), 1 deletion(-)
```

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`.

i Info

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

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

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

```
$ git pull --rebase
First, rewinding head to replay your work on top of it...
Applying: Añadido el mes al README
```

```
Using index info to reconstruct a base tree...
M   README.md

Falling back to patching base and 3-way merge...
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
error: Failed to merge in the changes.
Patch failed at 0001 Añadido el mes al README
hint: Use 'git am --show-current-patch' to see the failed patch

Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To 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

El contenido del fichero final podría ser:

```
Curso de GIT, febrero, 2020
```

A continuación confirmamos los cambios y los enviamos al servidor

```
$ git add README.md
$ git rebase --continue
$ git push
```

 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:

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


O como este:

```
$ git pull
error: 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
$ git pull # Sincronizamos con el repositorio remoto, -r para hacer rebase puede ser requerido
$ 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:

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

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

```
$ 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.

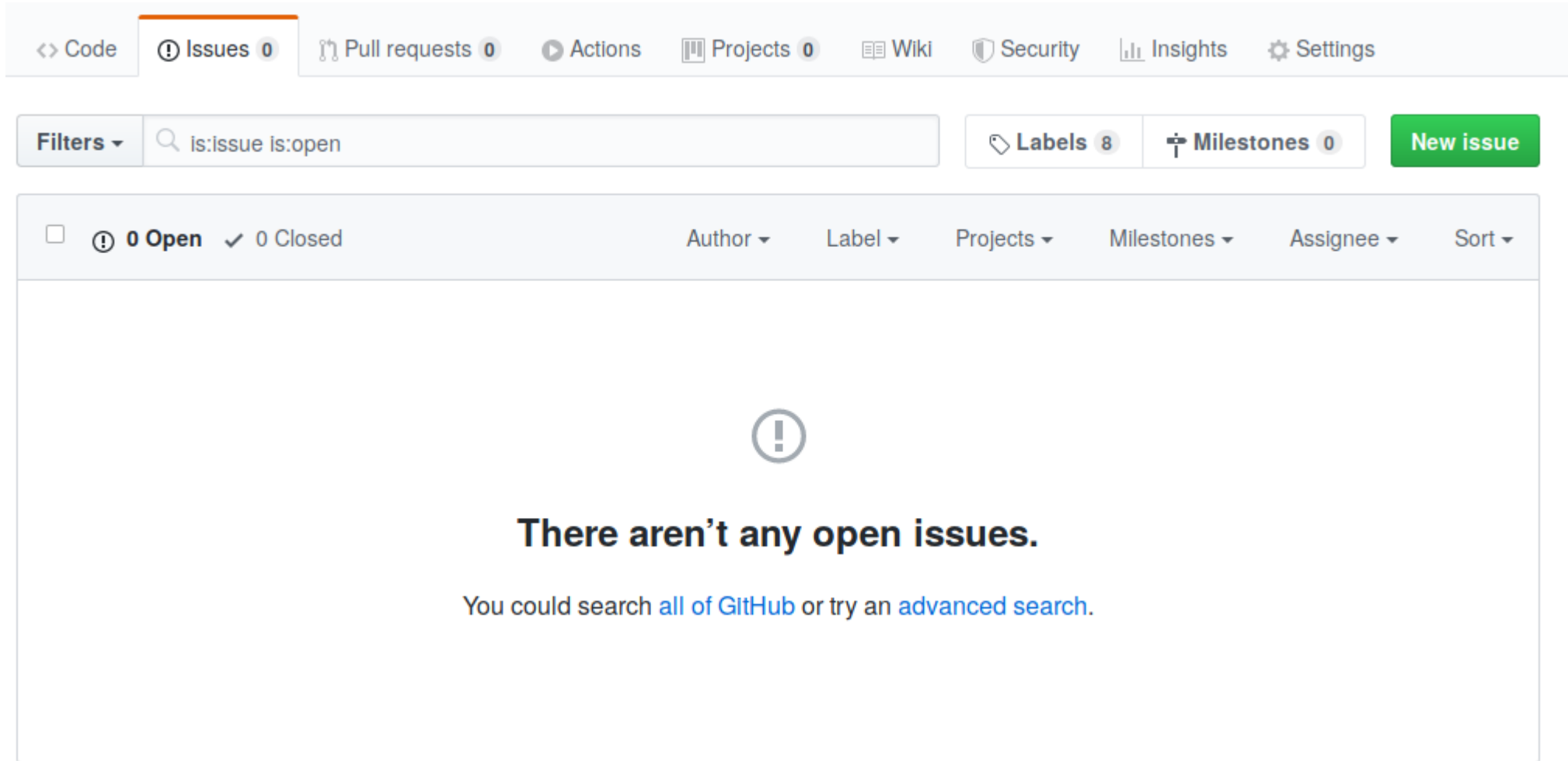
 **Tip**

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*.

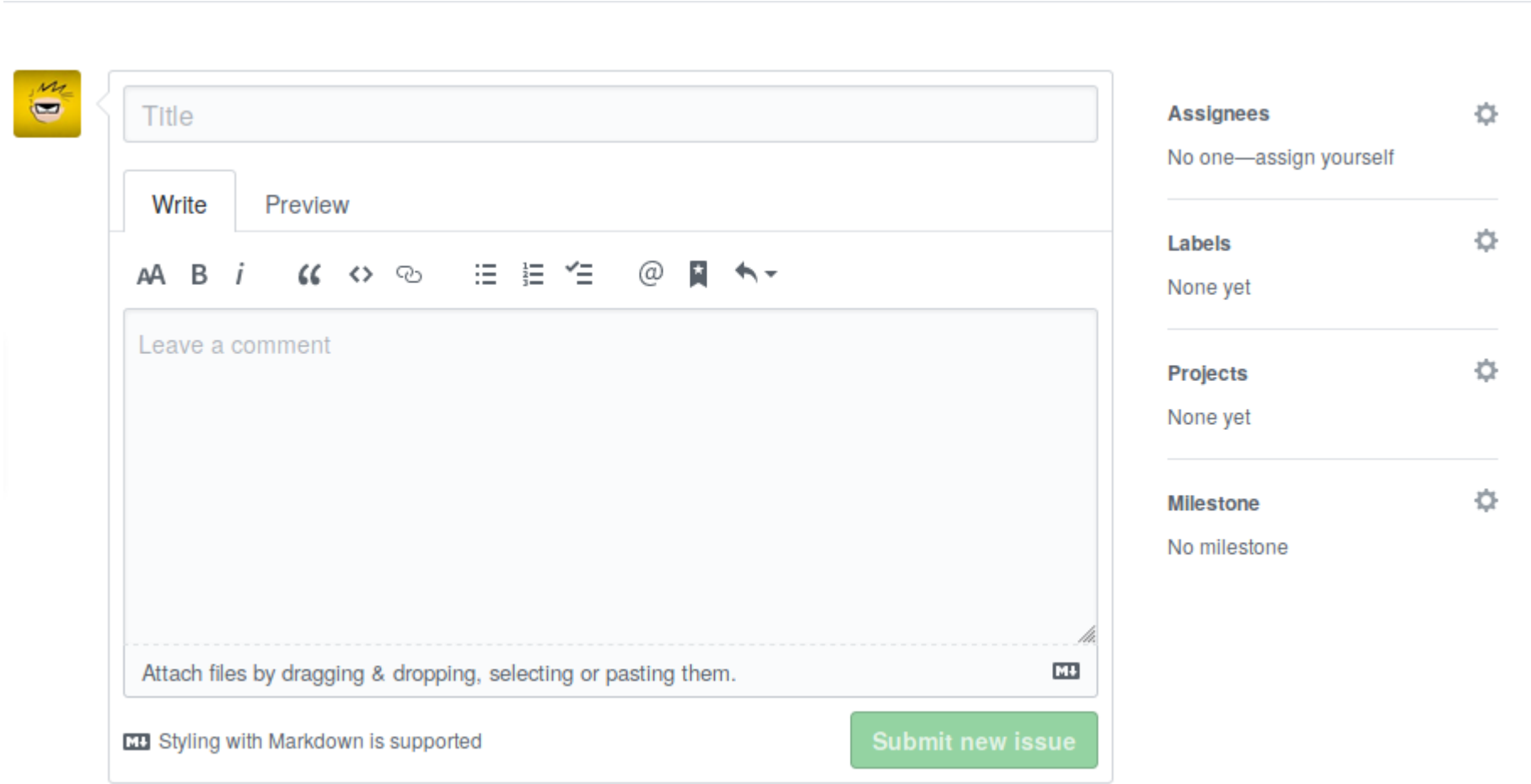
Flujo de trabajo en GitHub

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:

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

En entornos de trabajo multiusuario se puede usar el siguiente:

```
$ # usuario/tipo-número/descripción
$ git checkout -b sgomez/feature-1/create-changelog
$ 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.

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

El archivo puede contener, por ejemplo, lo siguiente:

```
# AUTHORS

* Sergio Gómez <sergio@uco.es>
```

Hacemos push y obtenemos algo como esto:

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

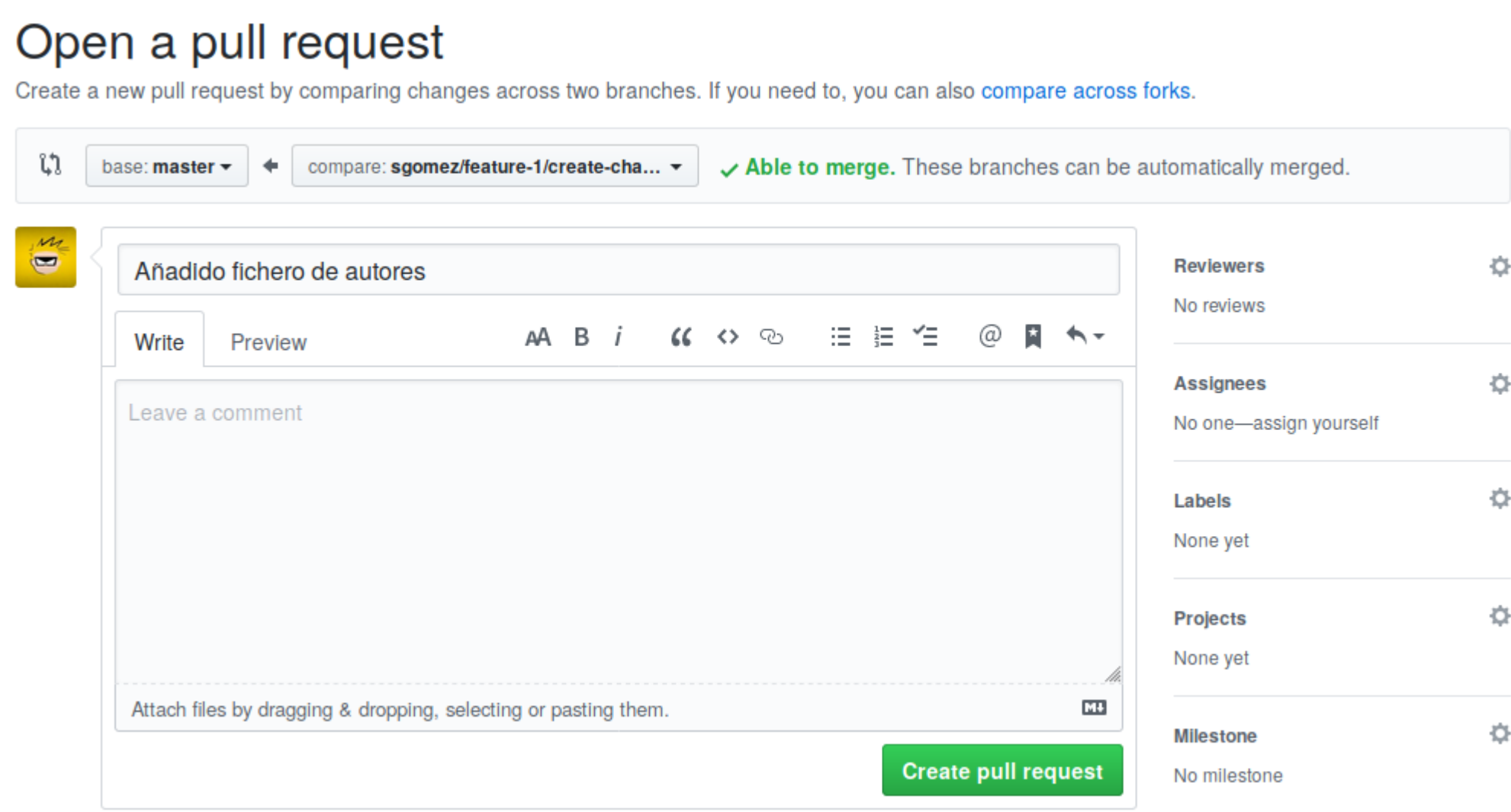
    git push --set-upstream origin sgomez/feature-1/create-changelog
```

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:

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

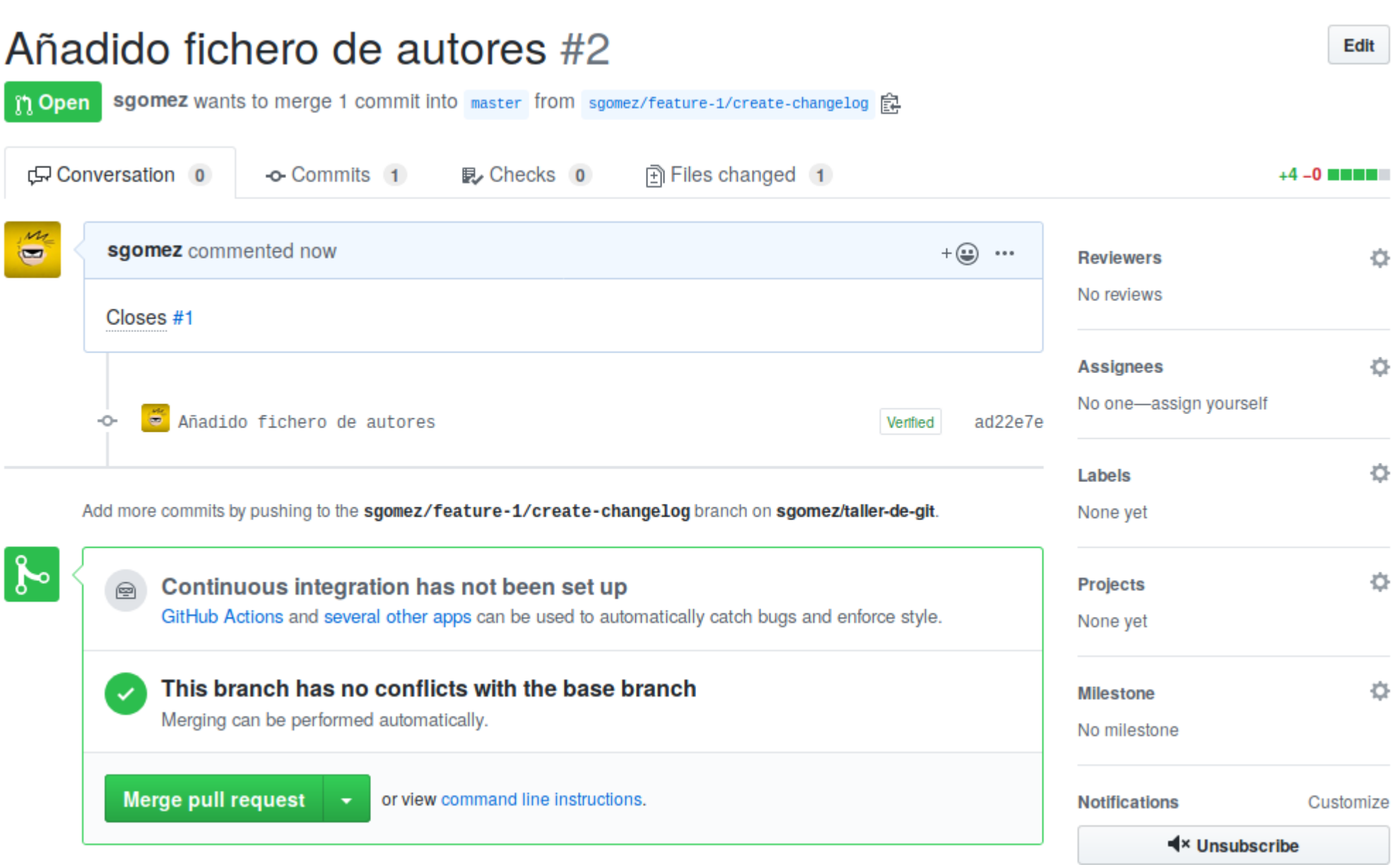


Aquí podemos informar de en qué consiste la rama que estamos enviando. Si ya tenemos una *issue* abierta, no es necesario repetir la misma información. Podemos hacer referencia con el siguiente texto:

```
Closes #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.



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 .

```
# AUTHORS

* Sergio Gómez <sergio@uco.es>
* John Doe
```

Y mandamos otro commit

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

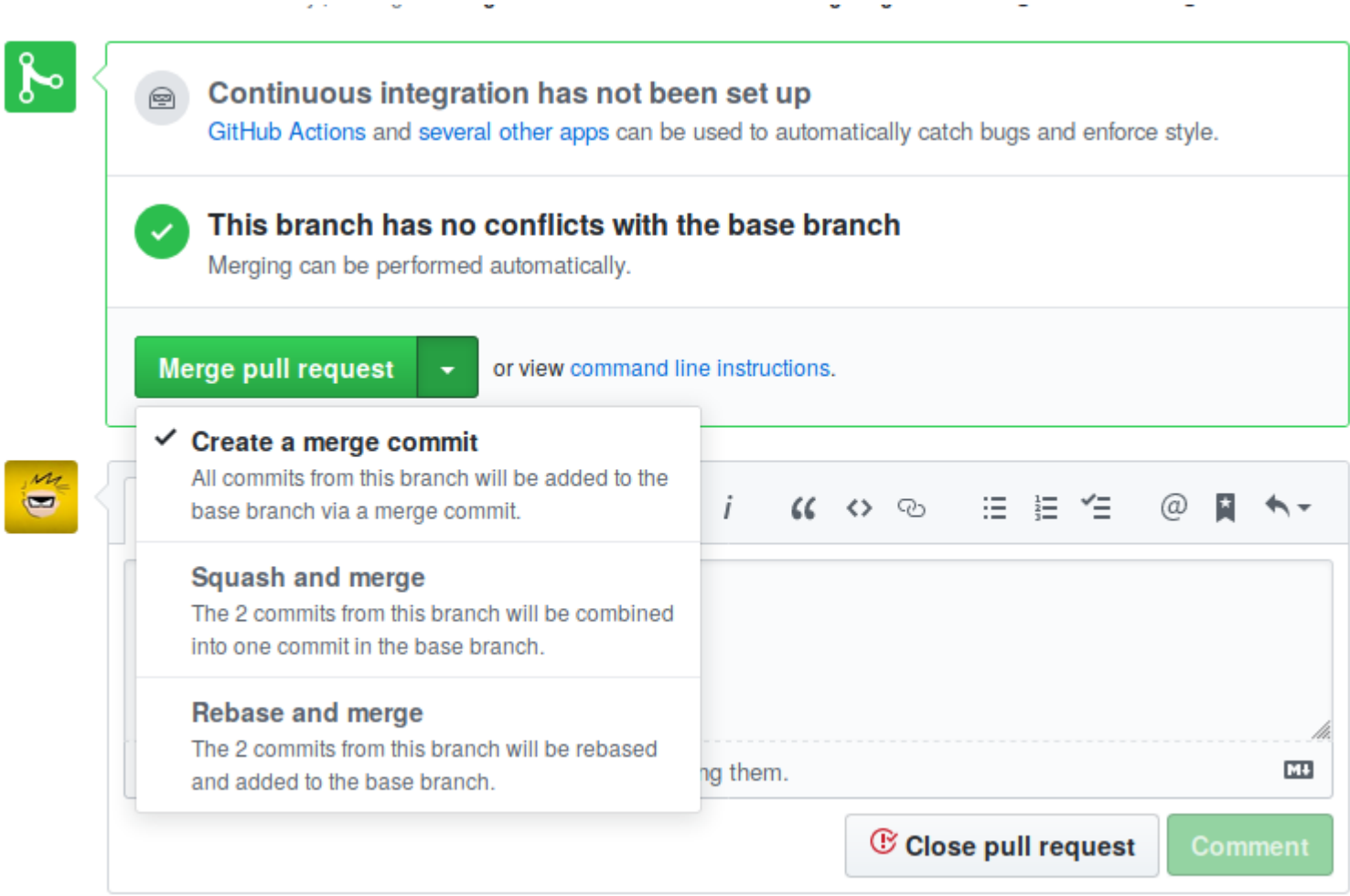
Si volvemos a la página de PR, veremos que aparece el nuevo commit que acabamos de enviar.

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:

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

```
$ git rebase master
$ git checkout master
$ git merge --no-ff sgomez/feature-1/create-changelog
$ 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.

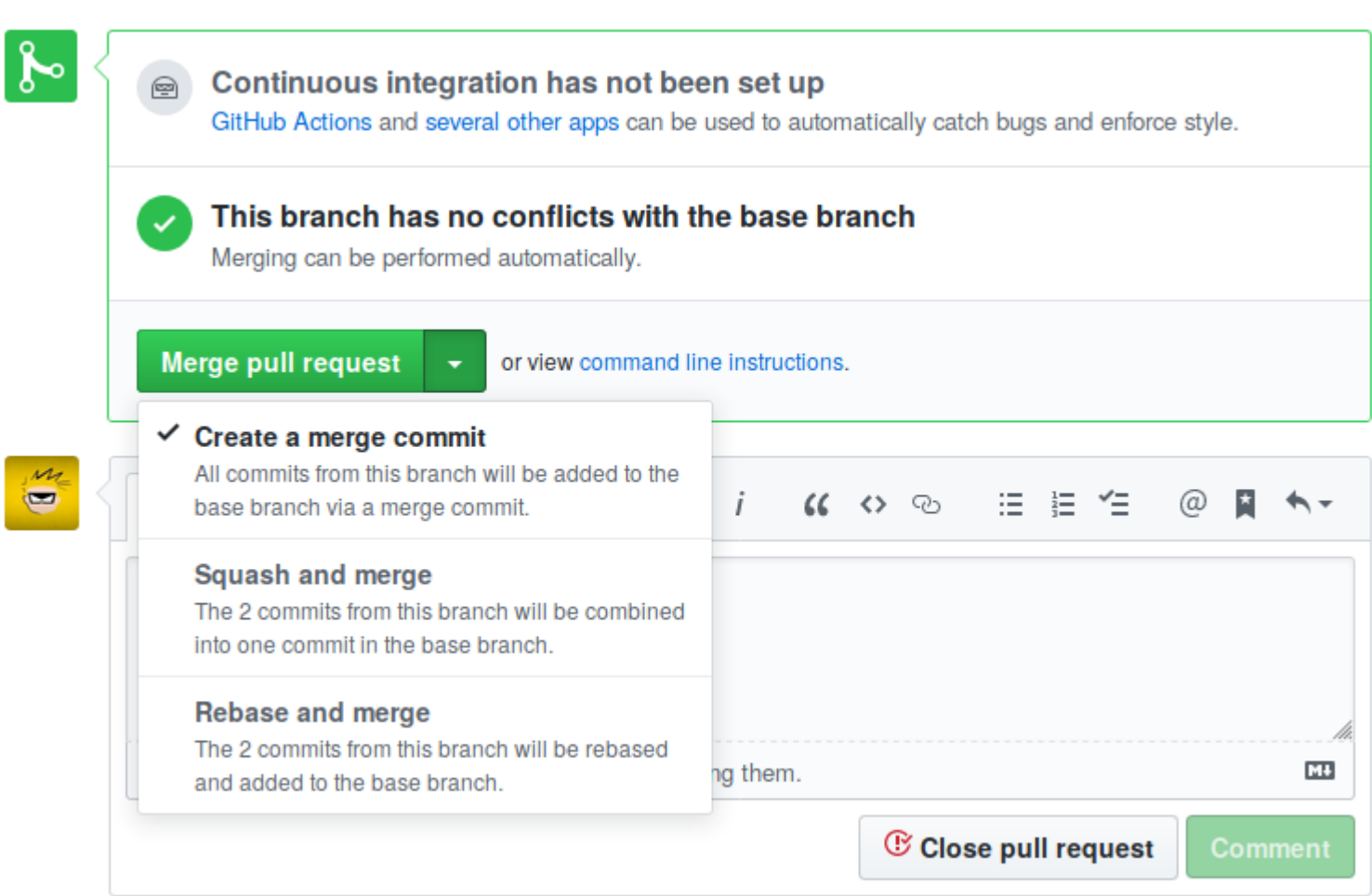
Crear un squash commit y un merge

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

```
$ git checkout master
$ git merge --squash sgomez/feature-1/create-changelog
$ 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:

```
$ git checkout master
$ git pull --rebase --autostash
$ 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 cracterísticas 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.

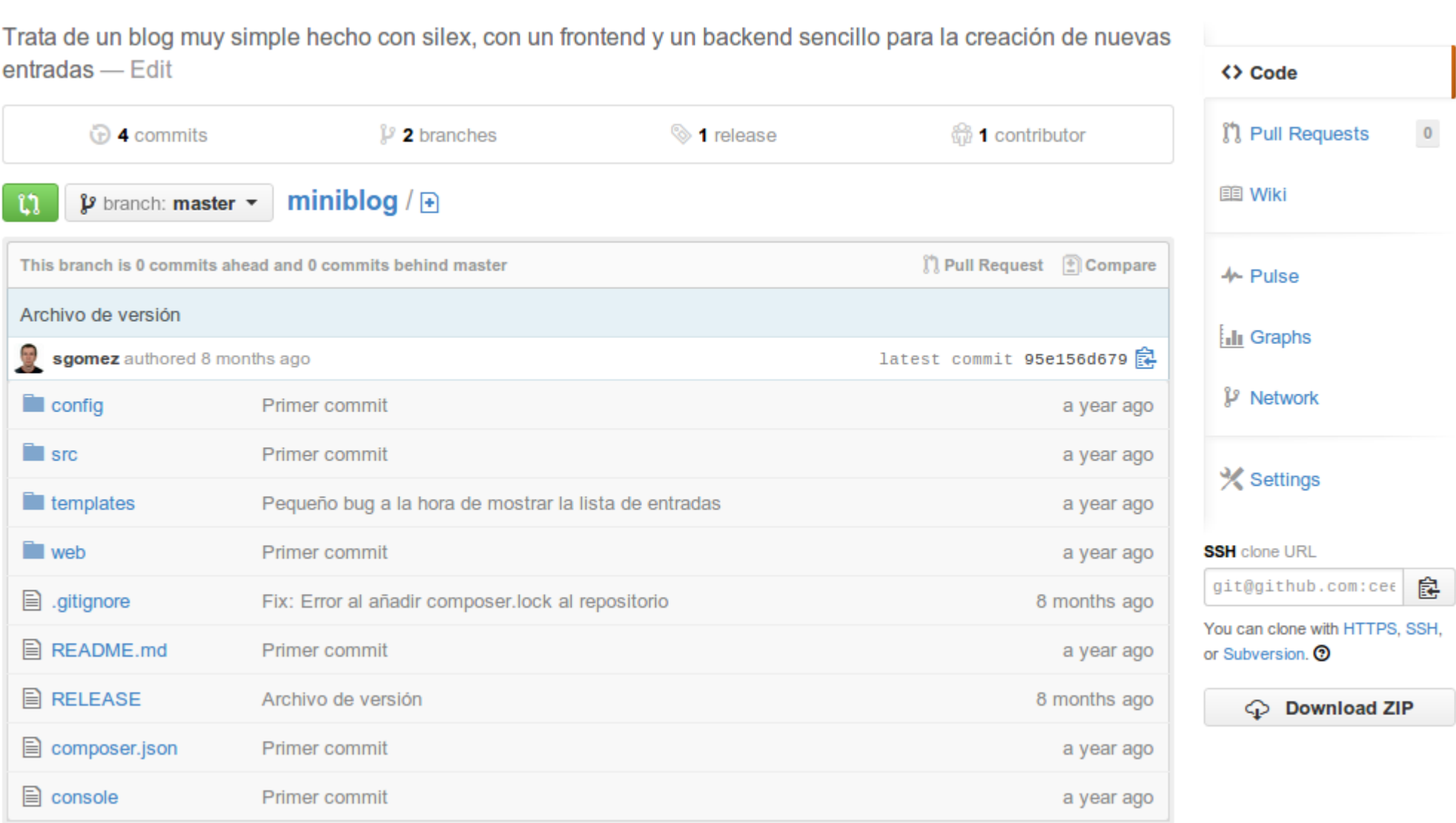
Clonar un repositorio

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.



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:

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

Lo que hace el código anterior es:

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

Y probamos que nuestra aplicación funciona:

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

Podemos usar dos direcciones para probarla:

- Frontend: `http://localhost:9999/index_dev.php`
- Backend: `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:

```
$ git remote show origin
* remote origin
Fetch URL: git@github.com:miusuario/miniblog.git
Push URL: git@github.com:miusuario/miniblog.git
HEAD branch (remote HEAD is ambiguous, may be one of the following):
develop
master
Remote branches:
develop tracked
master tracked
Local branch configured for 'git pull':
master merges with remote master
Local ref configured for 'git push':
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:

```
$ git remote add upstream git@github.com:sgomez/miniblog.git
$ git remote show upstream
* remote upstream
Fetch URL: git@github.com:sgomez/miniblog.git
Push URL: git@github.com:sgomez/miniblog.git
HEAD branch: master
Remote branches:
develop new (next fetch will store in remotes/upstream)
master new (next fetch will store in remotes/upstream)
Local ref configured for 'git push':
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:

```
$ git fetch upstream
```

```
$ git fetch upstream
$ 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

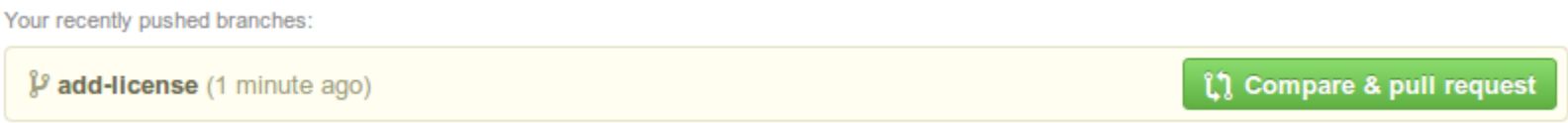
Vamos a crear una nueva funcionalidad: vamos a añadir una licencia de uso. Para ello preferentemente crearemos una nueva rama.

```
$ git checkout -b add-license
$ echo "LICENCIA MIT" > LICESE
# el error es intencionado
$ git add LICESE
$ git commit -m "Archivo de licencia de uso"
```

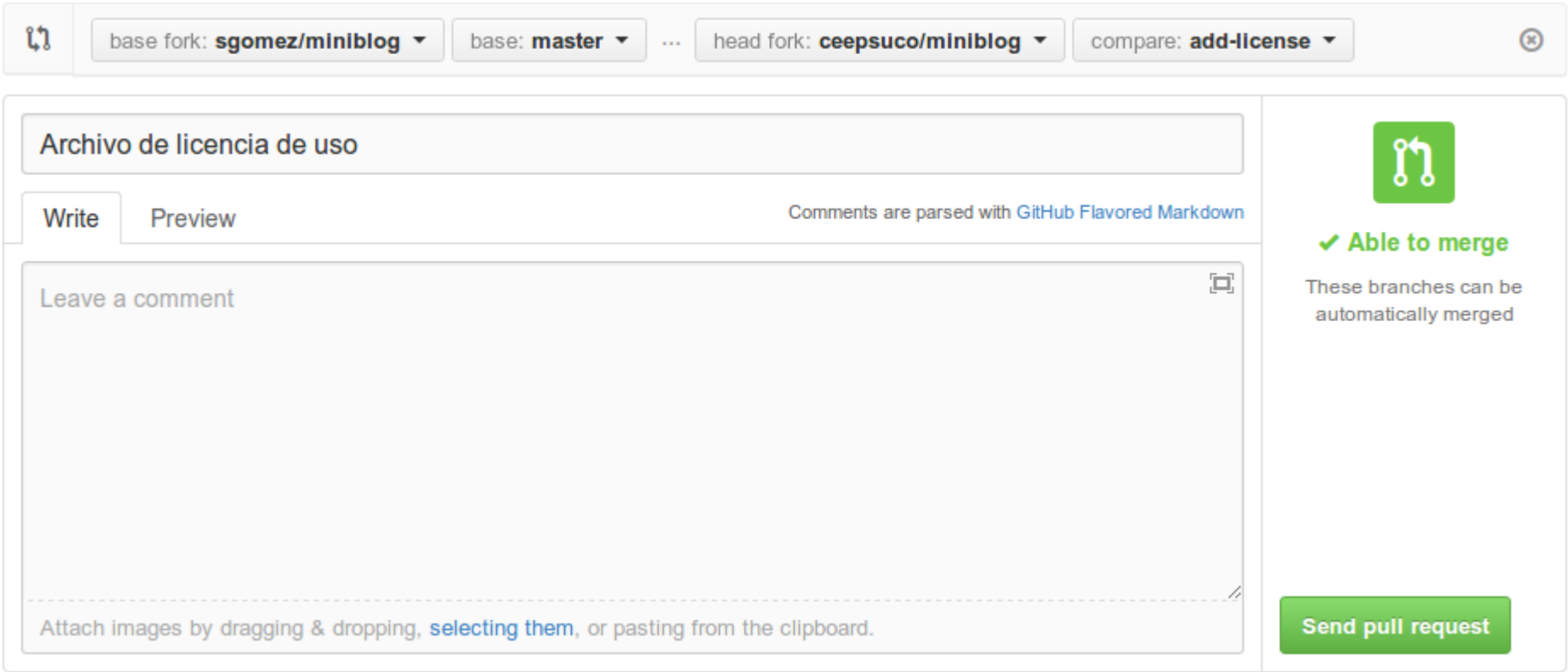
En principio habría que probar que todo funciona bien y entonces integraremos en la rama *master* de nuestro repositorio y enviamos los cambios a Github:

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

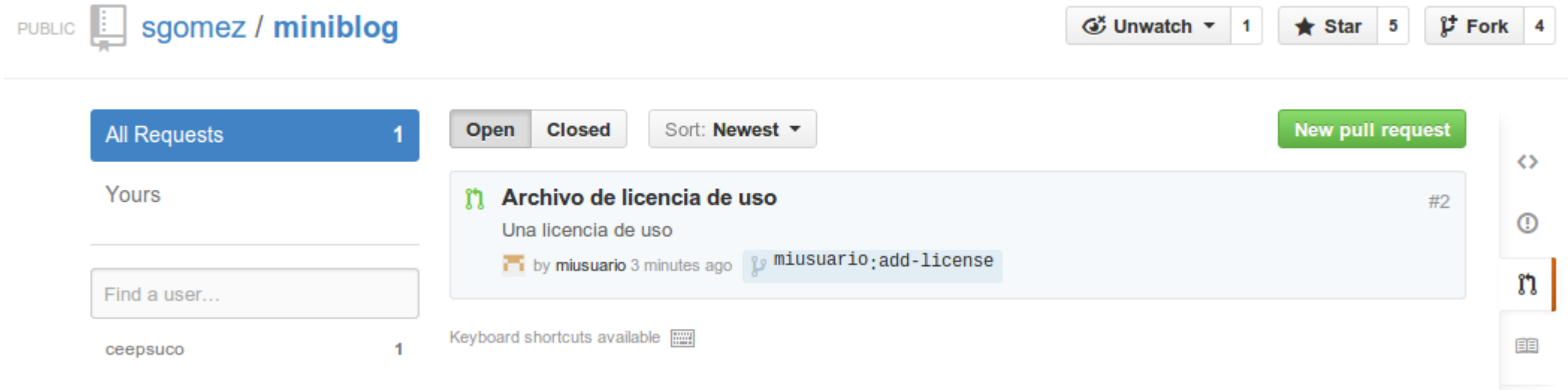
Si volvemos a Github, veremos que nos avisa de que hemos subido una nueva rama y si queremos crear un pull request.



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.



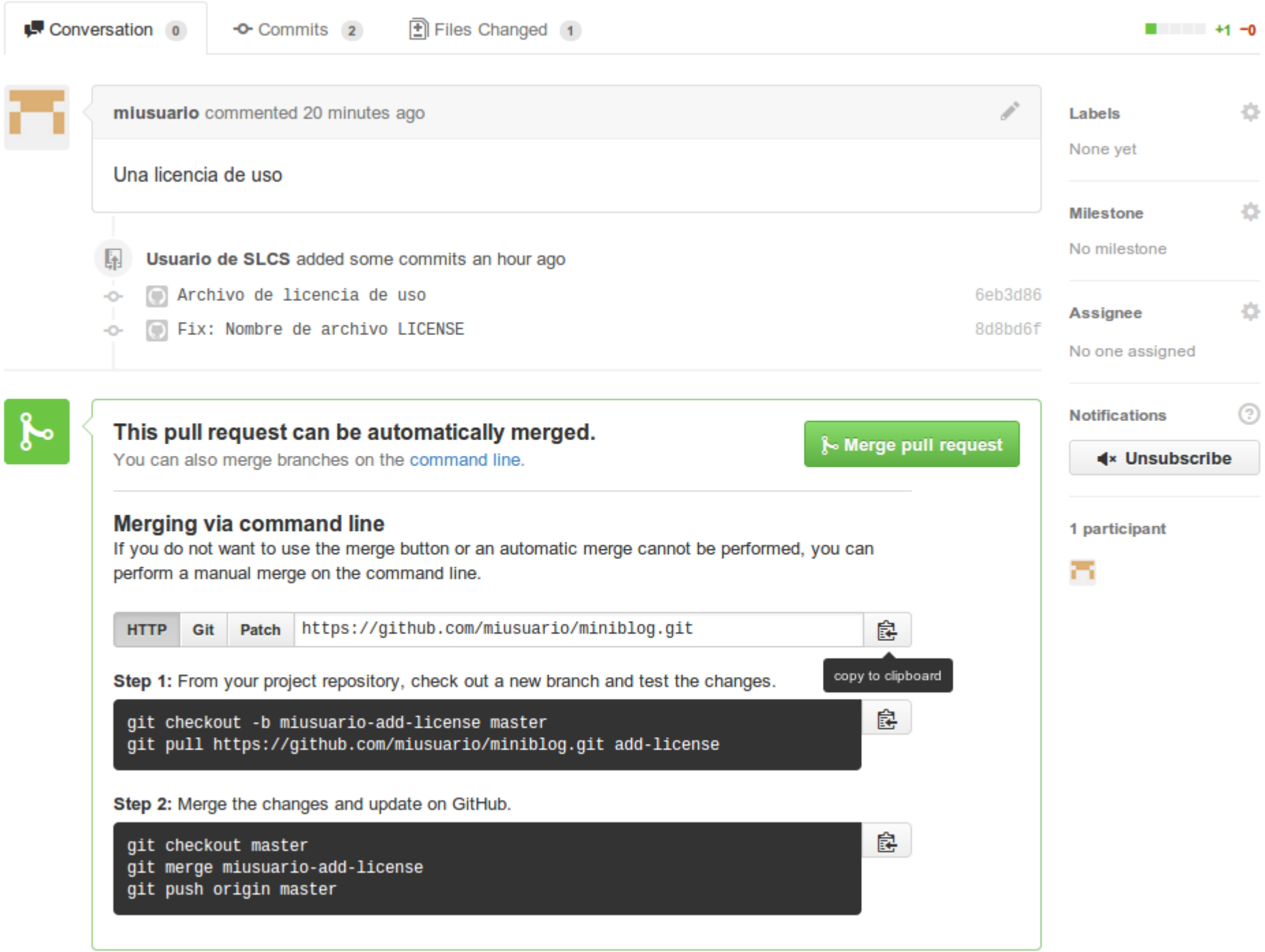
Una vez hemos terminado y nos aseguramos que todo está correcto, pulsamos *Send pull request* y le llegará nuestra petición al dueño del proyecto.



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.

```
$ git mv LICESE LICENSE
$ git commit -m "Fix: Nombre de archivo LICENSE"
$ 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.



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.

```
$ git checkout master
```

```
$ git branch -d add-license
# Esto borra la rama local

$ git push origin --delete add-license
# 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](#). 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:

```
git stash save "Tu mensaje"
git stash save -u
```

El parámetro `-u` permite que se almacén 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.

```
$ git stash list
stash@{0}: On master: Stash con mensaje
stash@{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 es `stash@{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.

```
$ git stash show
A.txt | 1 +
B.txt | 3 +++
2 file changed, 4 insertions(+)
```

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

```
$ git stash show -p
--- a/A.txt
+++ b/A.txt
@@ -45,6 +45,7 @@ nav:
+ This is a change
```

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

```
$ 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.

```
git 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` :

```
$ git worktree add ../project-master master
$ 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.

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

```
$ 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.

Git blame

Lo ideal en un equipo de desarrollo es que el código pase por todas las manos para así mejorar su calidad.

Con git blame podemos saber quién fue el último en modificar una línea concreta de código, en qué commit y en qué fecha lo hizo.

```
$ git blame ejemplo.php
33cdd02c (Sergio Gómez 2020-01-20 16:58:52 +0100 8)    name: "material"
33cdd02c (Sergio Gómez 2020-01-20 16:58:52 +0100 9)    language: "es"
```

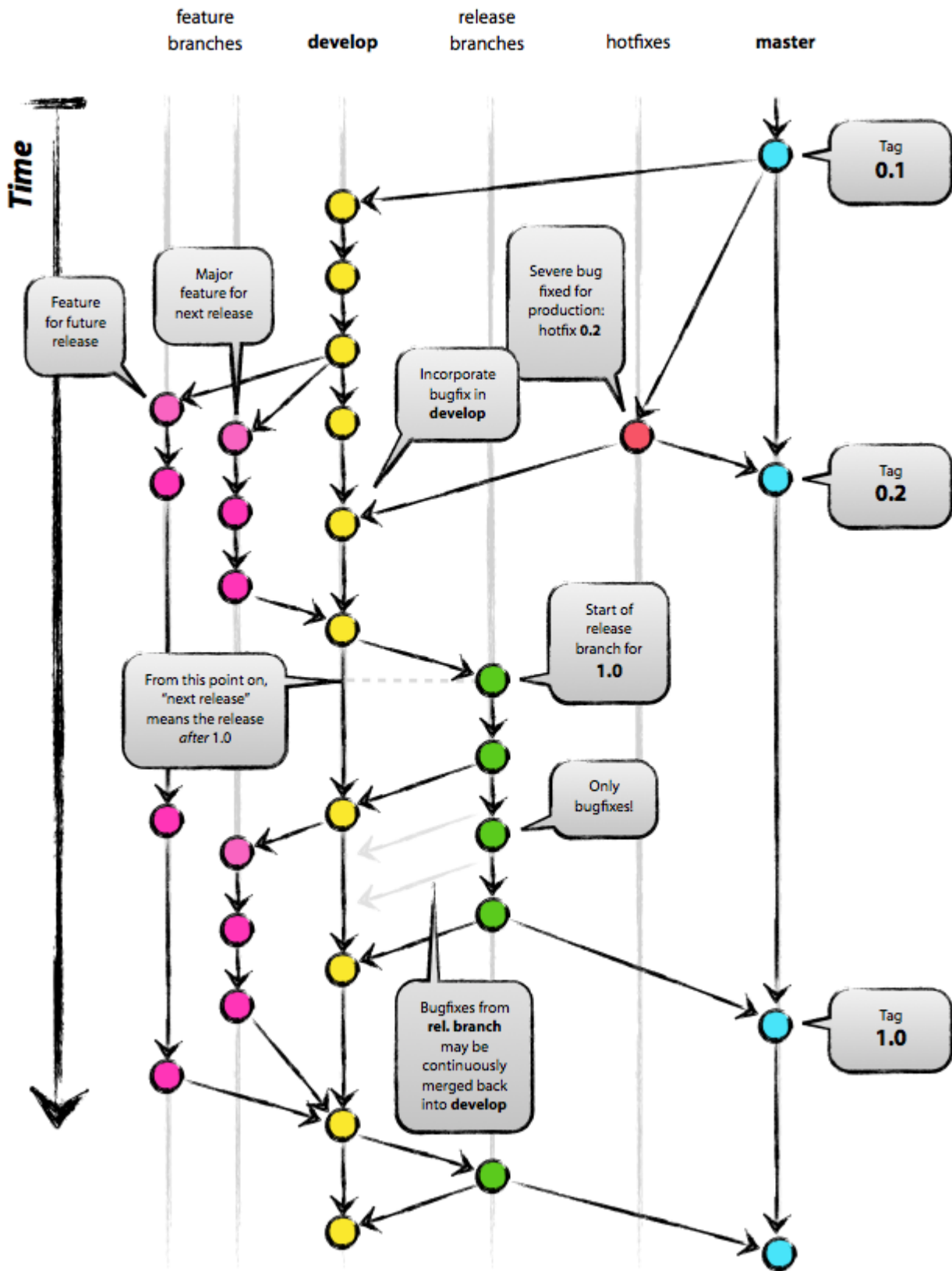

Flujo de trabajo con Git (git flow)

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 usabamos una rama máster y creábamos ramas para añadir funcionalidades que luego integrábamos. Es un 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
- Convención de nombres: feature-*NUM*issue-***.

Release branches

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

Hotfix branches

- Pueden partir de: master
- Deben fusionarse con: develop y master
- Convención 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:**

Inicializa el espacio de trabajo. De forma automática, crea las ramas que necesitamos y permite configurar el nombre de las mismas.

```
$ git flow init
Initialized empty Git repository in ~/project/.git/
No branches exist yet. Base branches must be created now.
Branch name for production releases: [master]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []

$ git branch
* develop
  master
```

Podemos ver que por defecto (usando intro en vez de escribir nada) pone nombres por defecto a cada rama. Con `git branch` comprobamos que ramas existen y en cual nos encontramos.

- git flow feature:

Permite crear y trabajar con ramas de funcionalidades.

```
$ git flow feature start feature_branch
```

Así creamos una rama 'feature/feature_branch' y nos mueve automáticamente a ella. En esta haremos los cambios que queramos en nuestro repositorio. Cuando queramos acabar de usar la rama, haremos un commit y la finalizaremos:

```
$ git flow feature stop feature_branch
```

Esto finaliza nuestra rama y la integra automáticamente a la rama develop. Si queremos seguir cambiando nuestro repositorio abriremos una nueva rama feature.

- git flow release:

Permite crear y trabajar con ramas de versiones. Cuando entendemos que despues de todas las funcionalidades (features, cambios en nuestro repositorio) nuestro trabajo esta listo para ser publicado, abriremos una rama release, que nacera de nuestra rama develop.

```
$ git flow release start 0.1.0
Switched to a new branch 'release/0.1.0'
```

Usaremos un tag para identificar de que release se trata. Ahora podemos hacer los cambios que estimemos oportuno para integrar todas las features que el repositorio ha sufrido hasta el momento. Tras hacer commit a todo el proceso, podemos cerrar la rama release.

```
$git flow release finish '0.1.0'
```

Esto la integrará de forma automática con master (con esto finalizamos el proceso de 'subir a producción' nuestro codigo) y con la rama develop, para que las futuras features estén al día.

- git flow hotfix:

Permite crear y trabajar con ramas de parches. Esto lo usaremos para hacer cambios rapidos que no puedan esperar a la proxima integracion de una release.

```
$ git flow hotfix start hotfix_branch
```

Tras hacer commit finalizamos la rama hotfix. Esta se fusionará con nuestra rama master y con nuestra rama develop para que esta también esté al día de los últimos cambios.

```
$ git flow hotfix finish hotfix_branch
```



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](#)
- [Flujo de trabajo de Gitflow](#)