Tema 5 Sistemas de Control de Versiones

1. 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, usando el correo @alu.edu.gva.es si no la tenemos aún.

1.1 Tu clave pública/privada

Muchos servidores Git utilizan la autentificación 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).

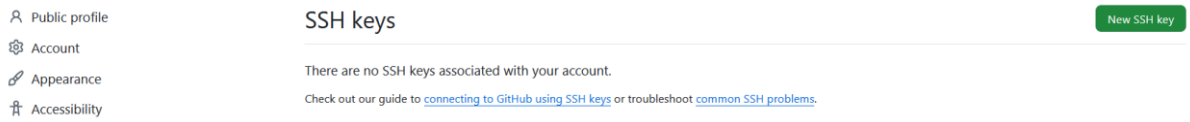
Para crear una nueva clave usamos la siguiente orden:

```
$ ssh-keygen -t rsa -C "Cuenta Github nombreApellidos"
```

1.2 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 tendriamos todo nuestro entorno para poder empezar a trabajar desde nuestro equipo.

1.3 Clientes 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/

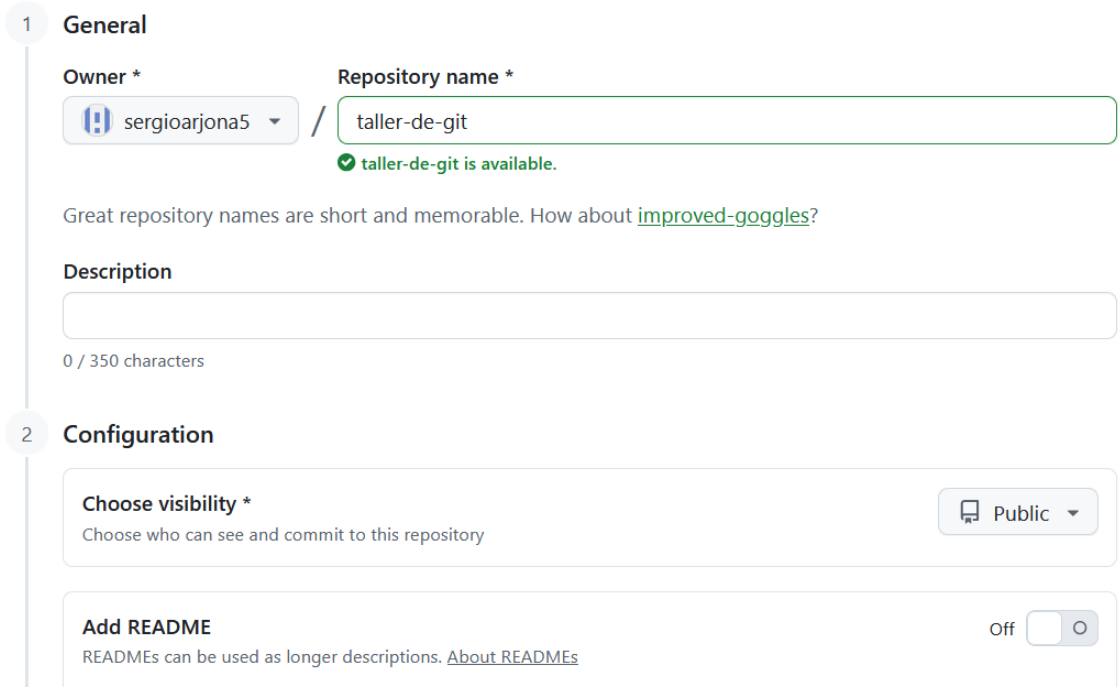
Para Linux no hay cliente propio, pero sí hay plugin para la mayoría de editores de texto como atom, netbeans, eclipe o los editores de jetbrains.

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.

1.4 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.

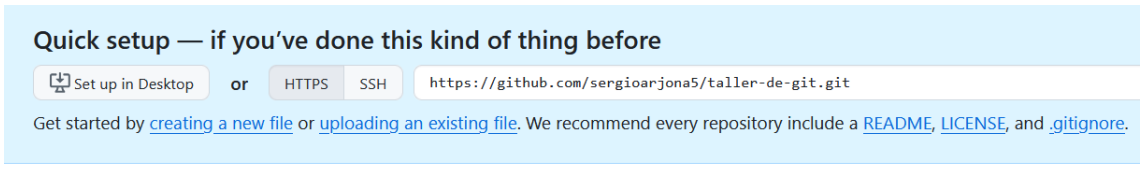
Ahora tenemos que designar un nombre para nuestro repositorio, por ejemplo: '*taller-de-git*'.



Nada más crear el repositorio nos saldrá una pantalla con instrucciones precisas de cómo 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.



...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 branch -M main
git remote add origin https://github.com/sergioarjona5/taller-de-git.git
git push -u origin main
```

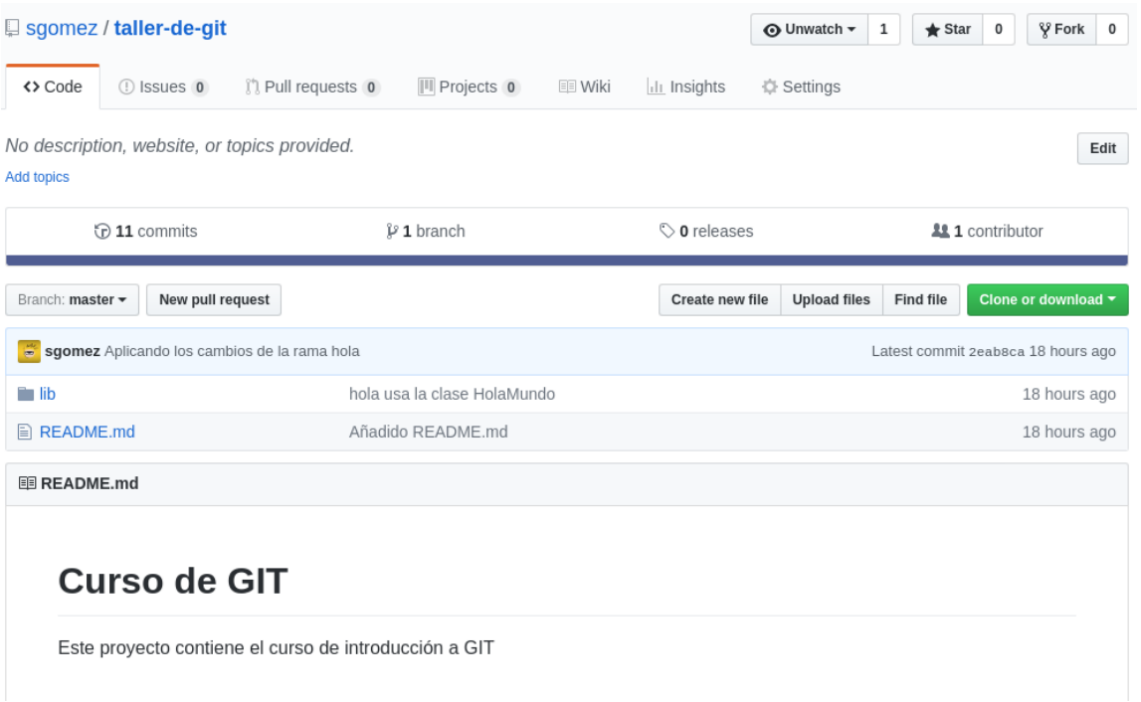
...or push an existing repository from the command line

```
git remote add origin https://github.com/sergioarjona5/taller-de-git.git
git branch -M main
git push -u origin main
```

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 órdenes 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.



1.5 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
```

1.6 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**[1]. Esta rama indica el estado de sincronización de nuestro repositorio con un repositorio remoto llamado *origin*. En este caso el de *Github*.

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.

1.7 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:
```

[1] Por norma se llama automáticamente *origin* al primer repositorio con el que sincronizamos nuestro repositorio.

```
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.
```

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
```

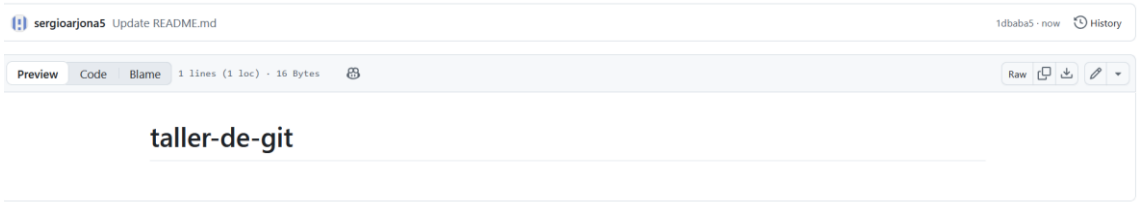
Si repasas las órdenes que te indicó Github que ejecutaras verás que el parámetro -u estaba presente y por eso no ha sido necesario indicar ningún parámetro al hacer push.

1.8 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.

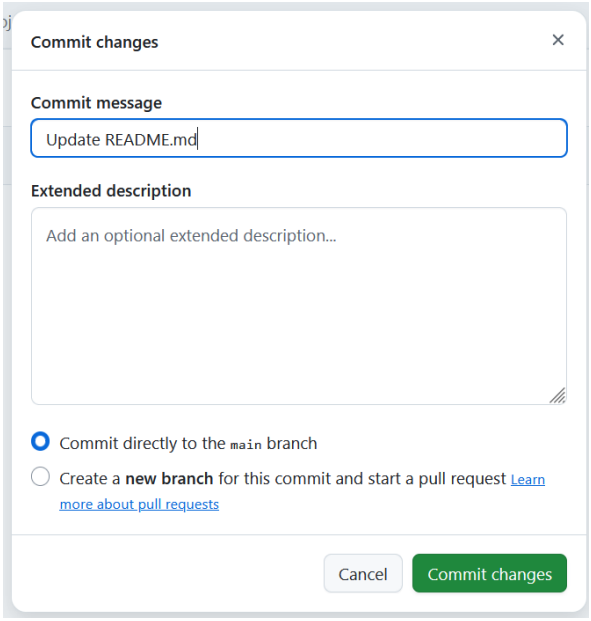
Para la prueba, Github nos permite editar archivos directamente desde la web. Pulsamos sobre el archivo `README.md`[2]. En la vista del archivo, veremos que aparece el icono de un lápiz. Esto nos permite editar el archivo.

[2] 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. Si quieres profundizar te dejo un tutorial: 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.
```



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
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 (en Github) y añadimos algo más:

```
# Curso de GIT


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


Desarrollado por Sergio.
```

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.

1.9 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.

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

```
Curso de GIT, 2025
```

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

```
Curso de GIT, febrero
```

Acordaros de hacer el commit en local y remoto.

Una vez hechos los commits, 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.

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

Para resolverlo debemos **modificar el archivo de conflicto** (ya sabemos cómo) y confirmar los cambios y enviarlos al servidor:

```
$ git add README.md

$ git rebase --continue

$ git push
```

¿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.

En remoto borra el año de la fecha y en local borra el mes. Pero esta vez no hagas commit en local (sí en remoto). 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
```

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.

2. Citar proyectos en GitHub

A través de una aplicación de terceros (Zenodo, financiado por el CERN), es posible crear un DOI para uno de nuestros proyectos. Un **DOI** (*Digital Object Identifier*) es un identificador único y permanente que se asigna a un documento digital (como un artículo científico, un libro, un conjunto de datos, un informe o un repositorio) para que pueda localizarse de manera estable en internet.

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.

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.

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, haremos clic en el botón publicar.

Paso 6. Publicar

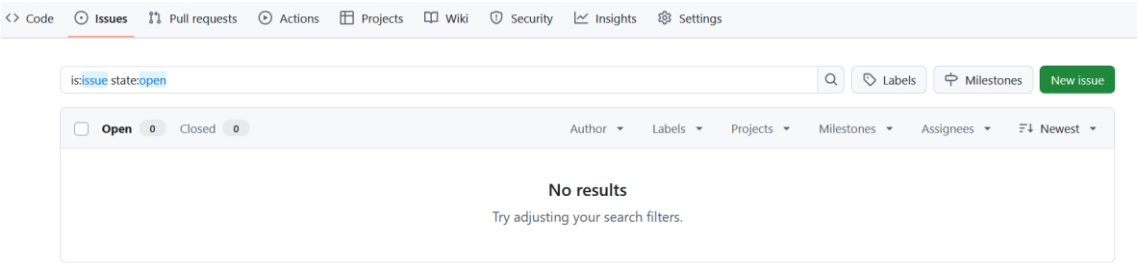
De vuelta a nuestra página de Zenodo, ahora deberíamos ver el repositorio listado con una nueva insignia que muestra nuestro nuevo DOI.

Podemos colocar la insigna 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*.

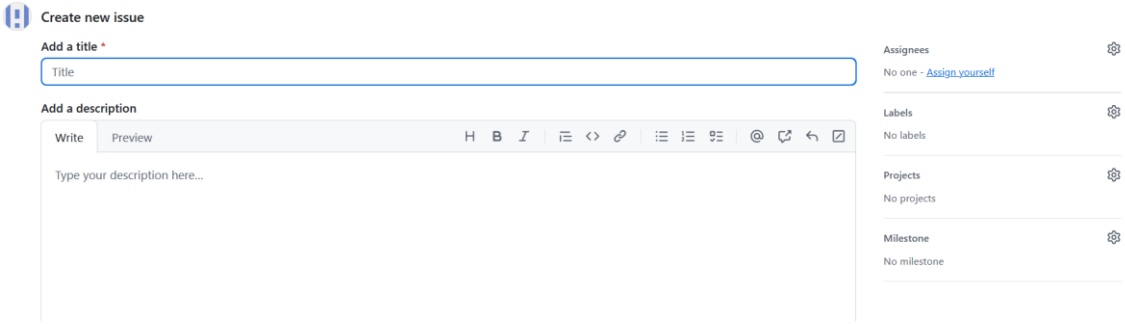
3. Flujo de trabajo en GitHub

Paso 0. Abrir una incidencia (issue)

Habitualmente el trabajo puede partir a raíz de un 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.

Actividad. Crea 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 (***feature***) al proyecto que estamos realizando. La misma puede estar provocada por una incidencia o no[3].

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:

[3] 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.

```
$ # tipo-número/descripción

$ git checkout -b feature-1/create-changelog

$ git checkout -b 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:

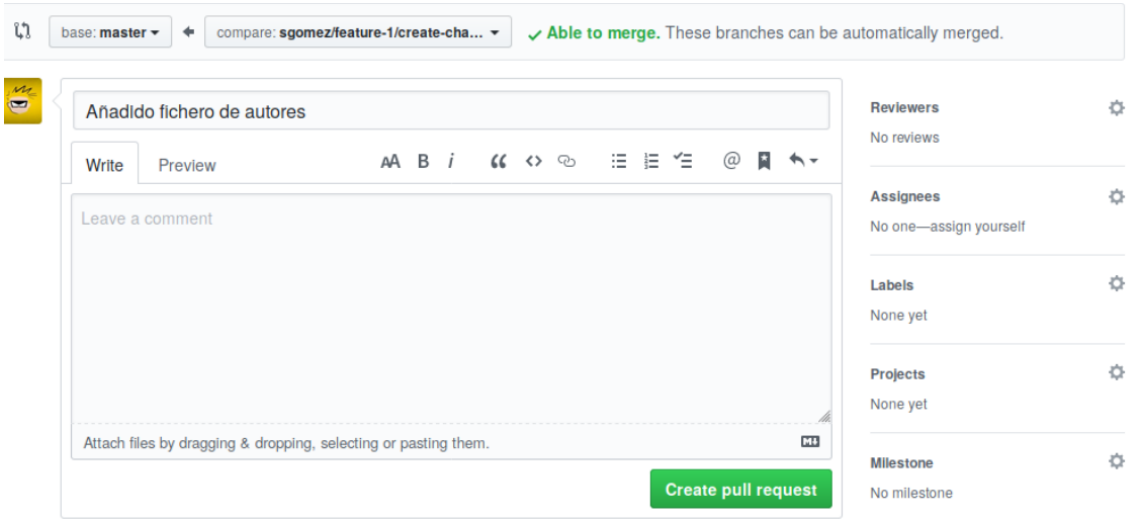
```
$ git push

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). Una PR es una solicitud para que los cambios realizados en una rama (branch) de un repositorio sean revisados e integrados en otra rama, normalmente la principal. Si vamos al enlace que nos aparece veremos lo siguiente:

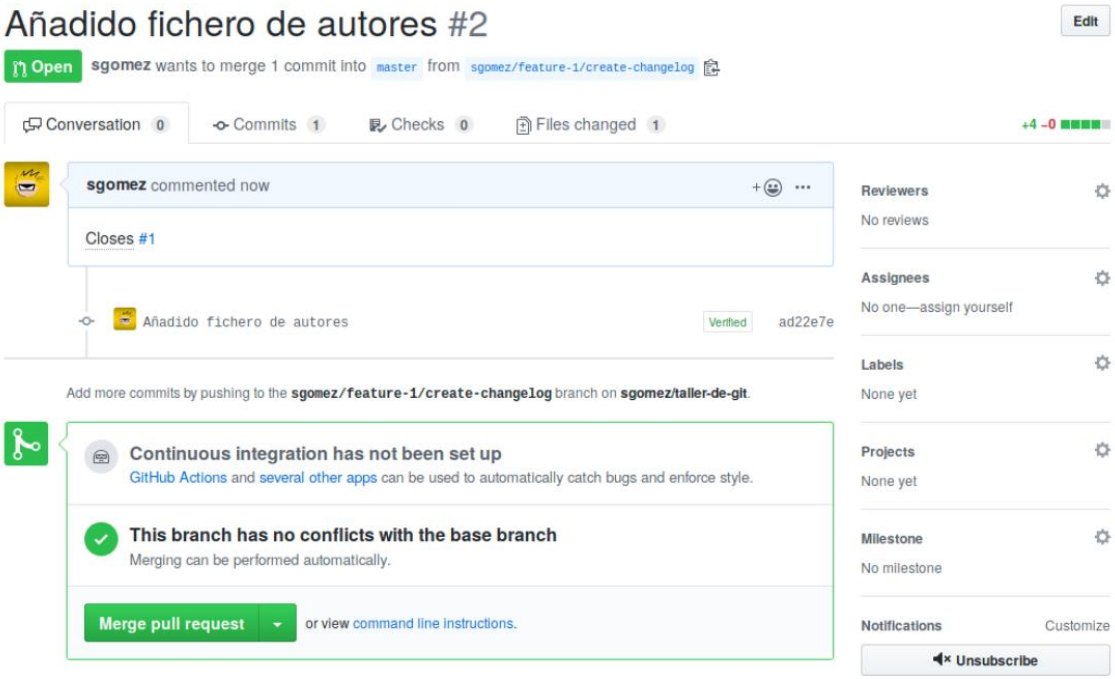


Aquí podemos informar de en qué consiste la rama que estamos enviando. Si ya tenemos una *issue* abierta, no es necesario repetir la misma información. Podemos hacer referencia con el siguiente texto:

```
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
```

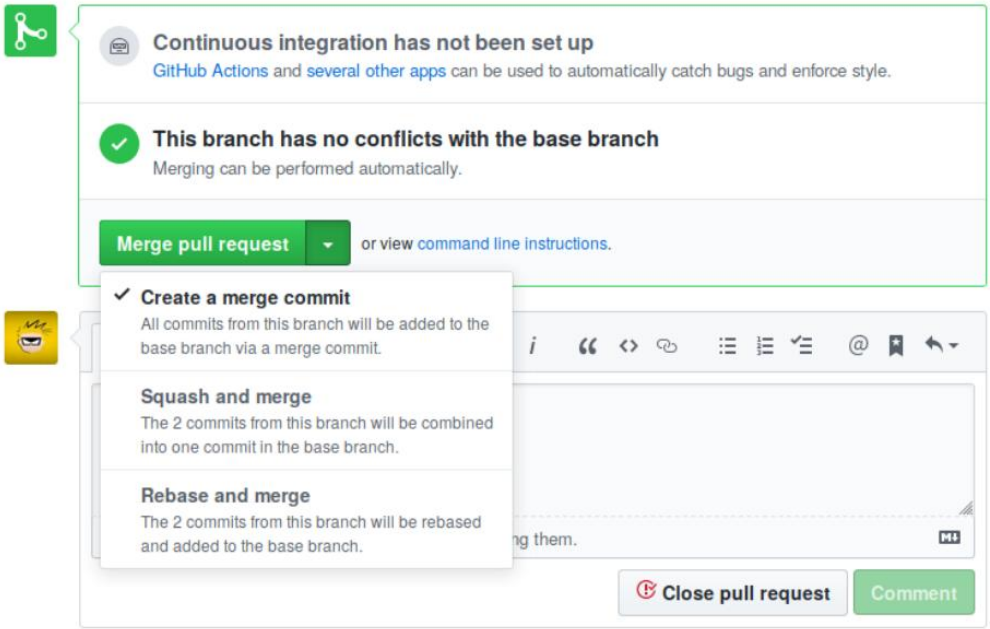
Si volvemos a la página de PR, veremos que aparece el nuevo commit que acabamos de enviar.

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:



Forma 1. 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 (Github siempre desactiva el fast forward).

Forma 2. 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.

Forma 3. 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 origin. 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
```

¿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.

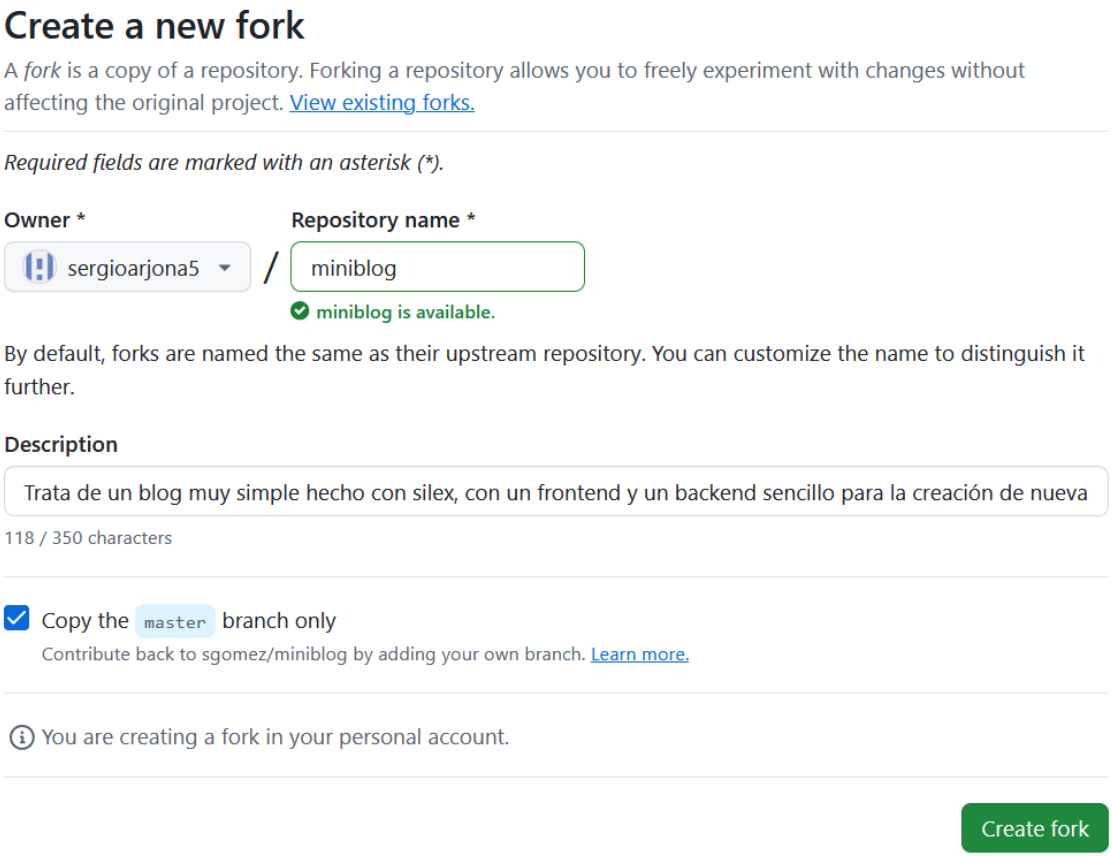
Para algunas personas estas características son unas ventajas, para otras no. Lo mejor es experimentar cada opción y cada uno decida como quiere trabajar.

4. GitHub avanzado

Esta sección trata de cómo colaborar con proyectos de terceros.

4.1 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 superior (más arriba) un selector para cambiar de contexto entre: explorador de código, pull requests, wiki, configuración, etc.

Github nos permite clonar localmente un proyecto por tres vías: HTTPS, SSH y GitHub Cli. 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:

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

Y probamos que nuestra aplicación funciona:

```
$ 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 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

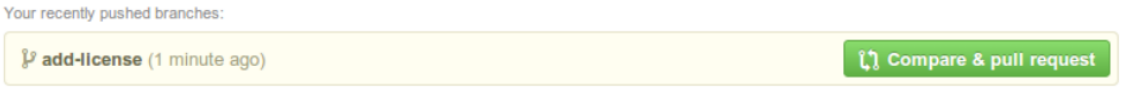
Vamos a crear una nueva funcionalidad: vamos a añadir una licencia de uso. Para ello preferentemente crearemos una nueva rama.

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

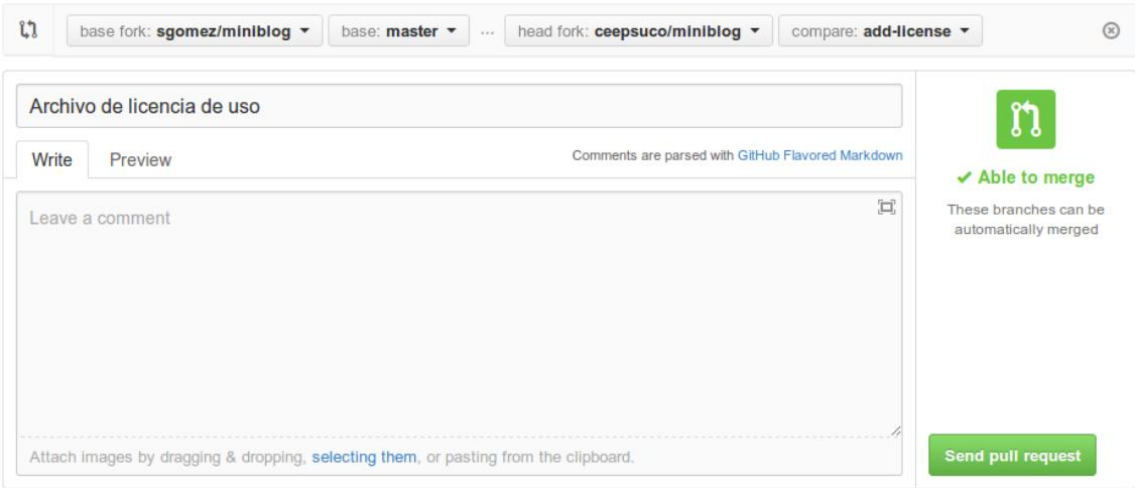
En principio habría que probar que todo funciona bien y entonces integraremos en la rama *master* de nuestro repositorio y enviamos los cambios a Github:

```
$ 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
```

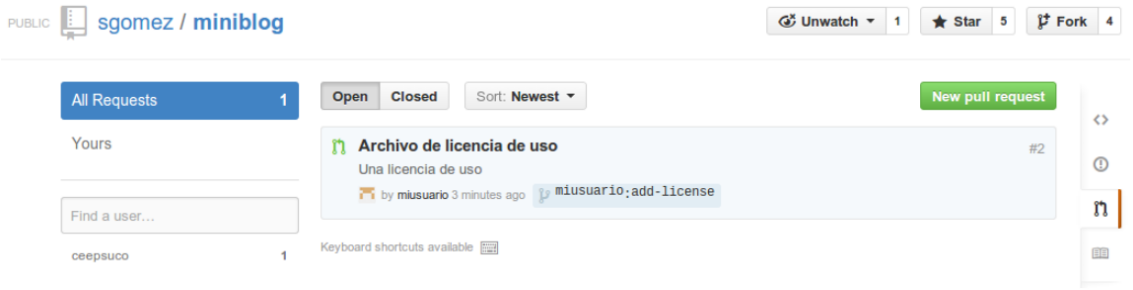
Si volvemos a Github, veremos que nos avisa de que hemos subido una nueva rama y si queremos crear un pull request.



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 correguimos 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.



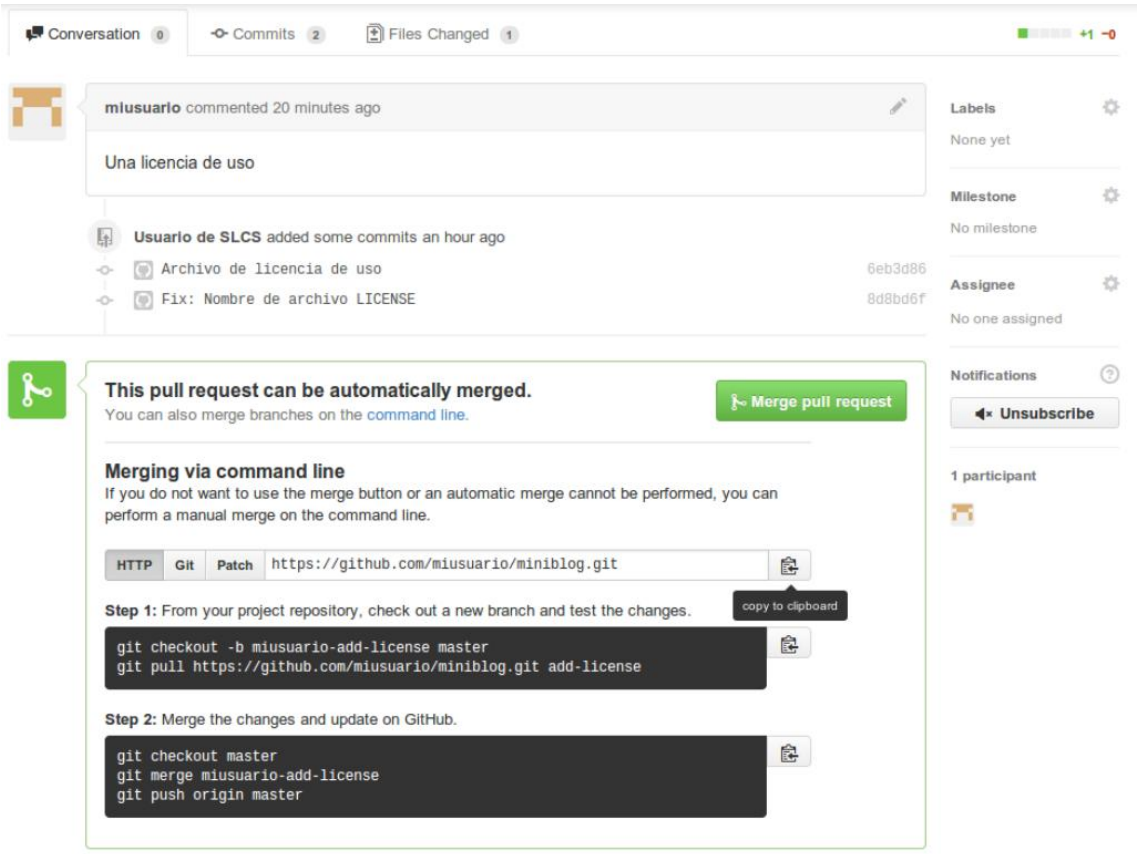
Una vez hemos terminado y nos aseguramos que todo está correcto, pulsamos *Send pull request* y le llegará nuestra petición al dueño del proyecto.



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 nuetra 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 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.

4.2 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