

Git - GitHub

1. Git y GitHub: Sistema de Control de Versiones

¿Qué son [Git](#) y GitHub? Antes de responder a esta pregunta hay que entender el concepto de *Sistema de Control de Versiones*. ¿Qué es y para qué sirve?

Sea lo que sea a lo que te dediques, cuando llevas a cabo un proyecto, ya sea un documento, una presentación, escribes un libro, desarrollas una aplicación o creas un juego, **el proceso no es inmediato** ni directo. Sabes cuando empiezas, pero no cuando acabas. Y una vez terminado, cuando lo veas, **lo corregirás y lo modificarás**, pero no una vez, ni dos, lo vas a modificar y corregir en **repetidas ocasiones**.

Esto de las **correcciones y revisiones** será porque detectas errores. Desde una falta de ortografía en la redacción de un artículo, pasando por una diapositiva de la presentación que no tiene ningún sentido, o un [bug](#) en el juego. O bien, porque quieres **modificar** una parte del mismo para **mejorarla**. Por ejemplo, porque el libro que estás escribiendo es sobre una determinada aplicación, que se ha actualizado y han liberado una nueva versión con nuevas características.

Llegados a este punto, el problema lo encontrarás cuando quieras **volver atrás**. Por ejemplo, una vez has introducido una modificación en *tu libro*, puede ser que no te guste como ha quedado y quieras deshacer los cambios. Si son pocos cambios, con **deshacer** tienes suficiente, pero ¿qué sucede cuando has introducido cambios sustanciales?

¿Cómo podemos **regresar al pasado**? Para esto es necesario llevar un **control de versiones**.

Sistemas de Control de Versiones

Para poder gestionar todos los cambios que se producen en nuestro proyecto, tenemos que recurrir a los **sistemas de control de versiones**.

Un sistema de control de versiones no es más que un **registro histórico de los cambios producidos**.

Sin embargo, existen diferentes modalidades de sistemas de control de versiones, en función de su funcionamiento.

Sistemas de Control de Versiones Locales



Durante la ejecución del proyecto es fundamental registrar los cambios que se van produciendo. Este registro de cambios te puede ayudar a volver a una versión anterior, en el caso de que la actual no sea de tu agrado, o a comprender por qué tu proyecto se encuentra donde se encuentra actualmente. Es de gran utilidad porque nos da un gran abanico de posibilidades como:

- Volver a una versión anterior de un documento del proyecto
- Volver a una versión anterior de todo el proyecto.
- Comparar los cambios entre dos versiones

Esto se puede resolver haciendo copias de nuestro proyecto. Es decir, cada vez que cambiamos un archivo, ya sea un documento, una imagen o lo que sea, creamos una nueva copia del proyecto.

En pequeños proyectos, lo que hago es exactamente esto. Por ejemplo, cada vez que realizo una modificación de un plano de un proyecto, creo una copia del mismo. Sobre esta copia hago las modificaciones que necesito, y genero una nueva versión. Ahora bien, aunque la mayor parte de las veces registro la fecha del cambio, no siempre, o más bien, casi nunca, registro a qué se han debido los cambios. En otras ocasiones, también me ha sucedido que mi modificación la he hecho no sobre la última versión, sino sobre una previa.

Efectivamente, esto es un **problema**, y es que en un momento podemos estar **trabajando en la versión que no toca** o simplemente borrar la última versión, o no guardar una versión anterior.

Para resolver este primer problema, se desarrollaron los Sistemas de Control de Versiones Locales. Que no es, ni más ni menos, que una **base de datos** donde se **registran cada uno de los cambios** que se producen en nuestro proyecto.

Sistemas de Control de Versiones Centralizado



Pasemos al siguiente nivel. ¿Qué pasa cuando trabajáis varios en el mismo proyecto? ¿Cuando sois varios los que intervenís en el proyecto? Cada uno de vosotros irá realizando cambios en el proyecto, de forma simultánea (o no). Si en el caso anterior ya era importante registrar los cambios, imagínate ahora. En este caso,

- Podemos averiguar quién ha introducido un determinado cambio
- Quién ha resuelto un problema o lo ha creado

Habrás comprendido claramente por qué es importante llevar un **sistema de control de versiones en un grupo**. Si en ocasiones, cuando trabajo solo, me equivoco y no parto de la última versión para realizar mis modificaciones, con varios es misión, prácticamente, imposible.

Pero aún nos queda un nivel más. ¿Qué pasa cuando cada uno de los **miembros** que trabajan en el proyecto está en **diferentes partes del mundo**?

Sistemas de Control de Versiones Distribuidos



Aquí entramos en los trabajos colaborativos. Varios miembros trabajando sobre un mismo proyecto, y deslocalizados, es decir, no todos se encuentran en la misma oficina (por ejemplo).

A priori, la solución es tan sencilla como tener el proyecto en la nube, ya sea en un servicio propio o en un servicio ajeno. Cuando necesitamos realizar una modificación del proyecto, descargamos la última versión de lo que queremos modificar, lo actualizamos y lo volvemos a subir. Esto tiene varios problemas:

- Estás limitado por el **ancho de banda** de tu conexión. Si cada vez que tienes que realizar una modificación tienes que mirar en la nube la última versión, descargarla, hacer las modificaciones, volverla a subir, y tu conexión a internet es tremendamente lenta, tu **trabajo será muy lento**.
- ¿Qué sucede cuando otro realiza modificaciones a la vez que tú?
¿Cuando dos estáis trabajando a la vez sobre un mismo archivo?
¿Cuál es la versión válida?

La solución a todo esto son los sistemas de **Control de Versiones Distribuidos**. Esto no es más que cada miembro del grupo tiene una **copia local del control de versiones**. Es decir, una copia del historial de cambios que se han ido realizando.

Cada vez que realizamos una modificación en nuestra copia ésta se **reporta** a la **copia principal**, donde es **gestionada** convenientemente por el **administrador** del proyecto.

Git

Git es un sistema de control de versiones distribuido pensado para proyectos donde existe un **gran número de archivos** de código fuente.

Los fundamentos

Una secuencia de instantáneas

Mientras que en otros sistemas de control de versiones lo que se *guarda en la base de datos* son los cambios que se producen, **Git** guarda **instantáneas**. Es decir, cada vez que **guardas el estado de tu proyecto**, *Git* realiza una *fotografía* de tu proyecto, de cada uno de los archivos de tu proyecto. Esta *instantánea*, refleja el estado de ese archivo.

En el caso de que un archivo determinado no haya sufrido ninguna modificación, no hace una copia de él, sino que guarda un enlace hacia el último archivo guardado.

De esta manera *Git* trabaja con tu proyecto como una **secuencia de instantáneas**. Cada *plano* de la película está constituido por cada una de las instantáneas que se tomaron de cada uno de los archivos que constituyen el proyecto.

Funcionamiento en local

Como *Git* guarda toda la historia del proyecto en tu ordenador, en una **base de datos local**, no necesita conectarse al servidor central para ver la evolución de un determinado archivo.

Igualmente, cuando realizas un cambio, éste lo puedes hacer en local, no necesitas conectarte a la base de datos central para registrar este cambio.

Garantía de Integridad

Cada vez que realizas un cambio, cuando se va a guardar el registro se realiza una **suma de comprobación** (*hash*). Esta suma de comprobación es la que se utiliza para referirse al cambio. Evidentemente, esto es una **garantía de integridad**.

Por otro lado, *Git* solo añade información. Si has realizado un cambio y éste se lo has confirmado a *Git*, se puede recuperar.

Los tres estados

El **flujo de trabajo** con Git es el siguiente:

- Realizas un cambio en un archivo: estado **modificado**
- Le indicas a Git que has modificado ese archivo: estado **preparado** [**git add nombre_archivo**]
- Confirmas todos los cambios realizados en los diferentes archivos: estado **confirmado** [**git commit -m "descripción del commit"**]

Así, cualquier archivo que se encuentre en nuestro sistema de control de versiones, **pasa por cada uno de estos tres estados**.

Es importante comprender esto de los tres estados, puesto que es el trabajo habitual con el que te enfrentarás cuando *pelees* con las revisiones.

Cuando estás trabajando en tu proyecto, vas modificando archivos (cada vez que modificas uno de ellos pasa al **estado modificado**). Cuando consideras que uno de los archivos modificados ya está terminado, llega el momento de comunicárselo al sistema de control de versiones. En ese momento este archivo pasa al **estado preparado**. Por último, cuando ya tienes todos tus archivos en estado preparado, llega el momento de confirmarlo, pasando a partir de ese momento al **estado confirmado**.

<https://colaboratorio.net/atareao/developer/2017/git-y-github-introduccion/>

2. Instalar Git

Este primer paso depende, evidentemente, de tu [sistema operativo](#).

Instalar Git en Linux

Instalar Git en [Linux](#) es realmente sencillo, pero la forma de instalarlo dependerá de tu distribución. A continuación te muestro cómo instalarlo en las distribuciones mas habituales:

- Debian/Ubuntu/Linux Mint y derivados:

```
sudo apt install git
```

- Opensuse:

```
sudo zypper install git-core
```

- Fedora:

```
sudo yum -y install git
```

- Arch/Manjaro:

```
sudo pacman -S git
```

Por otro lado, si además quieres una interfaz gráfica para lidiar con *Git*, tienes algunas muy interesantes como:

- [GitG](#). Esta es la interfaz gráfica por defecto de GNOME para gestionar repositorios GIT.
- [GitEye](#). Una herramienta gratuita multiplataforma.

También existen diferentes utilidades para *Nautilus* que nos ayudan con la gestión de los repositorios Git, como puede ser [RabbitVCS](#).

Instalar Git en Windows

Existen diferentes métodos para instalar *Git* en [Windows](#). Lo más sensato es recurrir al sitio oficial, en particular a la página de descarga de [git para windows](#), donde directamente al entrar en ella desde Windows comenzará la instalación, pues detecta la versión que tienes instalada. Por otro lado

también puedes descargar una versión **portable**. El inconveniente de esta versión, es que no tienes un interfaz gráfico, debes recurrir a un emulador de [BASH](#).

También puedes instalar otra opción de [Git para Windows](#), en la que tienes todas las herramientas necesarias, tanto para usuarios más experimentados como para usuarios novatos. Esta opción te ofrece tanto un **emulador BASH** en los que todos los usuarios Linux nos sentiremos realmente cómodos, como una interfaz gráfica, que tiene la versión gráfica de cada uno de las funciones de la versión de la línea de comandos.

Por último, otra opción muy interesante para instalar Git es instalar [GitHub para Windows](#). Esta opción te instalará una versión de línea de comandos así como una interfaz gráfica. Pero además, esto estará plenamente integrado con **GitHub**, con lo que si utilizas este repositorio remoto esta solución probablemente sea la más cómoda.

Instalar Git en MacOSX

Existen diferentes formas de instalar Git en Mac. Lo más sencillo es instalar **Xcode Command Line Tool**. En la versión 10.9 de Mac y superiores, solo tienes que intentar ejecutar *git* desde el *Terminal*, en el caso de que no esté instalado, directamente te **preguntará** si quieres instalarlo.

Por otro lado, si quieres instalar una versión más actualizada de Git, en el sitio web de la aplicación encontrarás un [instalador Git de Mac OSX](#)

Configurar Git

Una vez instalado nuestro cliente *Git*, lo primero que tenemos que hacer es **configurar** nuestro nombre de usuario y dirección de correo electrónico.

Si utilizamos la opción *—global*, esto solo lo tenemos que hacer una vez, puesto que a partir de entonces lo tomará por defecto. Así por ejemplo,

```
git config --global user.name "atareao"
git config --global user.email atareao@atareao.es
```


Si en un proyecto concreto queremos utilizar otro usuario o *email*, lo que debemos hacer es no añadir la opción *—global*.

Estos dos parámetros son importantes porque las confirmaciones de cambios utilizan esta información.

El resto de parámetros a configurar ya son **opcionales**. Algunos que te pueden ser de utilidad son:

- El editor de texto. En mi caso particular utilizo *nano* cuando edito en el emulador de terminal:

```
git config --global core.editor nano
```

- Herramienta de diferencias. La herramienta de diferencias (que comentaremos más adelante) es la que se encarga para resolver conflictos de unión:

```
git config --global merge.tool vimdiff
```

- Por último si quieres ver tu configuración, puedes ejecutar la siguiente orden en el emulador de terminal:

```
git config --list
```

Primeros pasos. Tu primer repositorio.

Una vez instalado y configurado Git, ha llegado el momento de tener nuestro **primer repositorio** en el equipo.

Para ello tenemos dos opciones: crear un repositorio desde cero o bien clonar un repositorio existente.

Crear un repositorio desde cero

Para crear un repositorio desde cero, crearemos un **nuevo directorio** con el nombre de nuestro proyecto. Por ejemplo *ejemplo00*. Posteriormente inicializamos el repositorio con *git init*.

Así, las órdenes a ejecutar en nuestro primer repositorio serían las siguientes,

```
mkdir ejemplo00  
  
cd ejemplo00  
  
git init
```

Si ahora listas el contenido del directorio *ejemplo00*, utilizando la orden (`ls -la`), verás que hay un nuevo directorio oculto *.git*. En este directorio se encuentran todos los archivos necesarios para el control.

Clonar un repositorio

Otra opción para tener nuestro primer repositorio sería **clonar uno existente**. Con esto tienes una copia idéntica de un repositorio que se encuentre en un servidor remoto.

Para esta serie de artículos estoy utilizando varios repositorios en [GitHub](https://github.com) que voy creando conforme los voy necesitando.

Así para clonar el primer ejemplo que he creado, la orden a ejecutar en un emulador de terminal será:

```
git clone https://github.com/atareao/ejemplo01.git
```

Esto creará un directorio *ejemplo01* donde hayas ejecutado la orden anterior. Dentro de ese directorio encontrarás además del directorio *.git* comentado en el apartado anterior (con toda la información del repositorio), una copia de trabajo de la última versión.

Es decir, verás varios archivos (algunos ocultos y otros no), que son los que se ha creado en el repositorio remoto.

Conclusión

En este segundo capítulo, hemos visto los siguientes aspectos:

- cómo instalar y configurar *Git* en tu equipo, con independencia del sistema operativo que tengas.
- cómo configurar *Git*.
- cómo crear tu primer repositorio, ya sea desde cero o bien clonando un repositorio remoto.

En el siguiente apartado se abordarán los siguientes puntos:

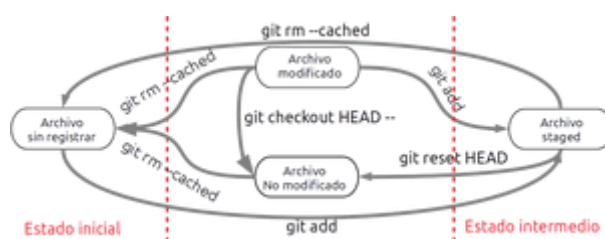
- Añadir nuestros primeros archivos al repositorio.
- El ciclo de vida de un archivo.
- Cómo realizar comparativas.

3. Trabajando con repositorios locales.

En nuestro repositorio local, creado como vimos en el artículo anterior, tenemos tres **depósitos** o estados:

- Depósito o estado **inicial**. Este es el directorio de trabajo, donde se encuentran todos nuestros archivos, aquellos que editamos y modificamos.
- Depósito o estado **intermedio**. Un almacén o depósito llamado **Index** o **Stage**. Esto es una especie de *memoria caché*. Aquí vamos registrando todos nuestros cambios antes de guardarlos en el depósito final
- Depósito o estado final. Un depósito llamado **HEAD** donde se guarda lo último.

En el siguiente diagrama o esquema vemos todas las posibles *transacciones* que podemos realizar con un archivo dentro de nuestro repositorio local.



Así, cualquier archivo que modificamos, tenemos que transferirlo primero al depósito intermedio y, posteriormente, guardarlo.

Pongamos que **creamos o copiamos** un nuevo archivo en nuestro directorio de trabajo. Lo llamaremos `mi_archivo`. Este archivo, nada más crearlo, no está **registrado** en nuestro sistema de control de versiones. Sin embargo, está en el directorio de trabajo, en el **depósito inicial**.

Para **registrarlo** y seguirle la pista (**rastrear** la traducción de **track**) ejecutaremos la siguiente orden:

```
git add mi_archivo
```

Al ejecutar este comando, nuestro archivo ha pasado del **depósito inicial**, o directorio de trabajo, al depósito intermedio, o **Stage**. Si queremos hacer marcha atrás y sacarlo de *Stage*, ejecutaremos la orden:

```
git reset mi_archivo
```

Todas estas operaciones siempre las estamos haciendo en nuestro repositorio local. Para que estos cambios se vean reflejados en el repositorio remoto hay que ir un paso más allá. Debemos llevar nuestros archivos al depósito o estado final, al **HEAD**.

Para pasar del **Stage** al **HEAD** es necesario realizar un *commit*. Con esta acción todo lo que se encuentra en el *Stage*, y solo eso, pasa al depósito HEAD. Así, para realizar un *commit*, ejecutaremos la siguiente orden:

```
git commit -m "Mi primer commit"
```

Como te puedes imaginar, "*Mi primer commit*", es un mensaje que añadimos para saber a qué se refiere ese *commit*. Por ejemplo, si has añadido una nueva característica, has corregido un error... o, simplemente, una falta ortográfica. En fin, lo que sea.

Igualmente, como sucedía en el caso anterior, también puedes devolver un archivo del estado final al estado intermedio. Para ello tan solo tenemos que ejecutar la orden:

```
git reset HEAD~
```

Ahora bien, ¿qué pasa una vez registrado `mi_archivo`, es decir, cuando se encuentra en el estado intermedio o *Stage*, y lo modificamos? Es decir, una vez registrado nuestro archivo le añadimos un nuevo apartado, o modificamos algo, o lo corregimos. En ese momento, es necesario realizar de nuevo un registro del cambio. Para ello, igual que al inicio de nuestro camino, ejecutaremos la orden:

```
git add mi_archivo
```

Ahora ya tenemos de nuevo nuestro archivo modificado registrado en el *Stage*. Si queremos que los cambios se vean reflejados para una próxima sincronización con el repositorio remoto, será necesario hacer un *commit*, como vimos en la primera parte.

```
git commit -m "Mi segundo commit. He modificado mi archivo"
```

Evidentemente, hacer esto archivo por archivo puede ser muy tedioso. Aunque esto ya es totalmente opcional, depende de ti. Tú eres el que le da más o menos importancia a los cambios a considerar. Sin embargo, para registrar toda una serie de nuevos archivos o nuevas modificaciones en archivos, en lugar de ir uno por uno puedes ejecutar la siguiente orden:

```
git add .
```

Esta acción, no es exactamente la misma que *git add*. Por diferenciar,

- * *git add -A* lo pasa **todo** al estado intermedio (Stage)
- * *git add .* pasa todo lo nuevo y lo modificado al estado intermedio, pero **no lo que se ha borrado**.
- * *git add -u* para al estado intermedio o stage todo lo que se ha modificado o borrado, pero **no lo nuevo**.

En ocasiones, te puede resultar interesante saltarte el depósito o estado intermedio, el *stage*. Si simplemente lo que estás haciendo es pasar de una a otra sin más, no tiene sentido utilizar este estado intermedio. En este caso, puedes realizar directamente el *commit*, añadiendo la opción *-a*. Así para realizar una confirmación o *commit* saltando el estado intermedio, tan solo deberás ejecutar la orden,

```
git commit -a -m "Saltandome el stage"
```

Conociendo el estado del repositorio

Inicialmente, cuando tienes pocos archivos, es posible, que conozcas en qué estado o situación se encuentra cada uno de ellos. Sin embargo, te puedo asegurar que a lo largo del proyecto, llegará el momento en que desconozcas la situación de todos los archivos. ¿Cómo puedes conocer la situación de tus archivos?. Para ello tan solo tienes que ejecutar la orden:

```
git status
```

Así por ejemplo, en un proyecto en el que estoy trabajando, al ejecutar esta orden, me devuelve el siguiente resultado:

```
root@kali:~/Programas/Debian/changelog# git status
On la rama master:
  tu rama está actualizada con «origin/master».
  cambios no preparados para el commit:
  (use «git add -p» para actualizar lo que se confirmó)
  (use «git checkout -- <archivo>...» para descartar cambios en el directorio de trabajo)

modificado:   debian/changelog
modificado:   debian/control
modificado:   debian/copyright
modificado:   debian/rules
modificado:   debian/TODO
modificado:   debian/TODO.in
modificado:   debian/TODO.in.in
modificado:   debian/TODO.in.in.in
modificado:   debian/TODO.in.in.in.in
modificado:   debian/TODO.in.in.in.in.in

no hay cambios agregados al commit (use «git add» o «git commit -a»)
root@kali:~/Programas/Debian/changelog#
```

Como ves, el propio *git*, me indica las acciones que puedo tomar. O bien, me indica utilizar `git add` para actualizar los cambios que se confirmarán en el *commit*, o bien, utilizar `git checkout --` para descartar los cambios que hicimos en el directorio.

Voy a confirmar todos los cambios, para lo que ejecutaré la orden:

```
git add .
```

Si ahora ejecuto `git status`, el resultado que devuelve es el siguiente:

```
root@kali:~/Programas/Debian/changelog# git add .
root@kali:~/Programas/Debian/changelog# git status
On la rama master:
  tu rama está actualizada con «origin/master».
  cambios preparados para el commit:
  (use «git reset HEAD <archivo>...» para sacar del stage)

modificado:   debian/changelog
modificado:   debian/control
modificado:   debian/copyright
modificado:   debian/rules
modificado:   debian/TODO
modificado:   debian/TODO.in
modificado:   debian/TODO.in.in
modificado:   debian/TODO.in.in.in
modificado:   debian/TODO.in.in.in.in
modificado:   debian/TODO.in.in.in.in.in

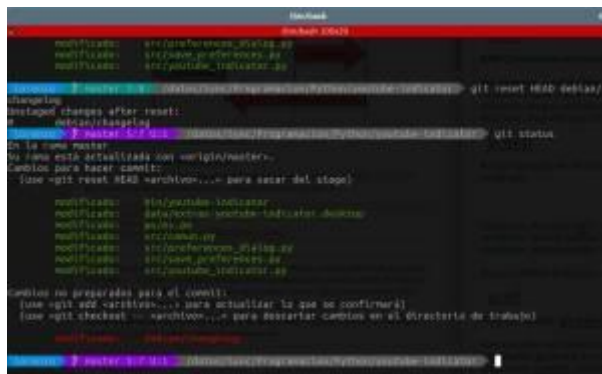
root@kali:~/Programas/Debian/changelog#
```

Como puedes ver todos los cambios están registrados en el *stage*, lo que hemos llamado el estado intermedio. Igual que en el caso anterior, *git* nos da dos nuevas posibilidades para hacer. Por un lado realizar un *commit*, o bien sacar uno o varios archivos del *stage*. Para sacar un archivo del *stage*, por ejemplo `debian/changelog` ejecutaremos la siguiente orden:

```
git reset HEAD debian/changelog
```

Si volvemos a ver el estado de nuestro repositorio, ejecutando la orden `git status` ahora el resultado que nos devuelve es significativamente distinto. Por un lado, tenemos los archivos sobre los que no hemos hecho nada, y que están preparados para el *commit* (en color verde), y por otro lado,

tenemos el archivo que hemos devuelto a su estado anterior, y que se encuentra en el estado inicial (en color rojo).



```

$ git checkout -- debian/changelog
$ git status
On branch master
nothing to commit, working tree clean

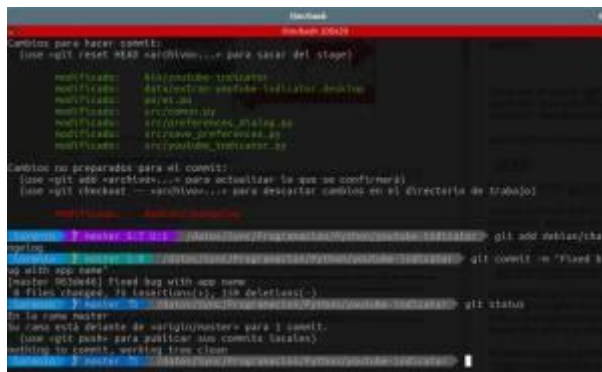
```

Para este segundo archivo, tenemos dos opciones distintas. O bien registramos los cambios pasandolo al *stage* y preparándolo para el *commit*, o bien, lo devolvemos a su estado anterior, ejecutando la orden:

```
git checkout -- debian/changelog
```

De esta forma, se desharian los cambios que hayamos realizado sobre el archivo en cuestión.

Voy a registrar los cambios en `debian/changelog` utilizando `git add` `debian/changelog` y posteriormente voy a realizar un *commit*:



```

$ git add debian/changelog
$ git commit -m "Fixed bug with app name"
[master 0b0d0d1] Fixed bug with app name
1 file changed, 75 insertions(+), 10 deletions(-)

```

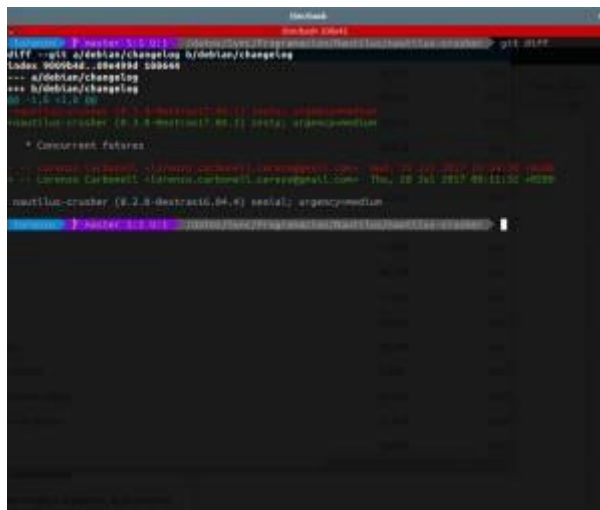
Si te fijas, al realizar el *commit*, nos da una estadística mínima, en la que se recoge:

- Número de archivos modificados
- Número de líneas añadidas
- Número de líneas borradas

Ahora, al ejecutar la orden `git status` nos indica que la rama está preparada para *publicar el commit*, es decir, para **sincronizar nuestro repositorio** con el repositorio remoto. Además, nos indica que no hay nada más que hacer.

Al detalle con los cambios diff

Evidentemente, como has podido ver, `git status` solo nos da una idea de la situación de cada archivo pero nos informa de lo que ha sucedido con él. Para ello es necesario de entrar al detalle. Así por ejemplo, en otro proyecto que llevo entre manos al ejecutar la orden `git diff src/nautilus-crusher.py` nos devuelve el siguiente resultado:



```
diff --git a/debian/changelog b/debian/changelog
index 0000000..0000000
+++ b/debian/changelog
@@ -1,5 +1,5 @@
nautilus-crusher (0.2.0-0ubuntu1) unstable; urgency=medium
+ nautilus-crusher (0.2.0-0ubuntu1) unstable; urgency=medium
- nautilus-crusher (0.2.0-0ubuntu1) unstable; urgency=medium
+ nautilus-crusher (0.2.0-0ubuntu1) unstable; urgency=medium
```

En color rojo, estamos viendo las líneas que se han eliminado, mientras que en color verde, veremos las líneas que se han añadido. Hay que indicar que cuando se modifica una línea se considera como si se hubiera borrado por completo, y se añadiera la línea modificada.

Si ejecutamos `git diff` sin indicar ningún archivo, nos mostrará todos los cambios ocurridos en nuestro repositorio local. Indicar, que `git diff` muestra **solo** los cambios realizados pero que están pendientes de registrar. Para ver los cambios que están registrados y pendientes de confirmar, debemos ejecutar la orden `git diff --cached` o `git diff --staged`, indiferentemente.

4. Trabajando con repositorios remotos.

Es posible que o bien hayamos creado nuestro repositorio de cero:

```
mkdir ejemplo00  
  
cd ejemplo00  
  
git init
```

O bien lo hayamos clonado de un repositorio remoto existente:

```
git clone git@github.com:atareao/ejemplo01.git
```

En el primero de los casos evidentemente no tendremos repositorio remoto, mientras que en el segundo sí.

Para ver que repositorios remotos tenemos, deberemos ejecutar la siguiente orden:

```
git remote -v
```

Que en el segundo caso, nos devolverá el siguiente resultado:

```
origin  git@github.com:atareao/ejemplo01.git (fetch)
```

```
origin  git@github.com:atareao/ejemplo01.git (push)
```

En el caso de que tuviéramos más de un repositorio remoto aparecerían listados también. Por ejemplo:

```
launchpad  git+ssh://lorenzo-carbonell@git.launchpad.net/my-  
weather-indicator (fetch)
```

```
launchpad  git+ssh://lorenzo-carbonell@git.launchpad.net/my-  
weather-indicator (push)
```

```
origin  git@github.com:atareao/my-weather-indicator.git (fetch)
```

```
origin  git@github.com:atareao/my-weather-indicator.git (push)
```

Gestión de repositorios remotos

A lo mejor a priori, puedes pensar que no tiene mucho sentido **mantener varios repositorios remotos**, pero todo tiene su razón de ser. Por ejemplo, en el caso de [Launchpad.net](https://launchpad.net), además de permitir hospedar el código, tiene diferentes sistemas de seguimiento. Además del típico de errores, también tiene uno para especificaciones y nuevas características, otro para ayuda a la comunidad y un sitio específico para traducir aplicaciones a múltiples idiomas.

He sido un fiel seguidor y usuario de **Launchpad**, pero inicialmente utilizaban otro sistema de control de versiones, **Bazaar**, y esto es lo que me hizo decantarme por GitHub. Ahora que ha dado opción para utilizar [git](https://git.launchpad.net), supongo que empezaré a utilizar de nuevo *Launchpad*, pero en la modalidad que estamos viendo aquí, es decir, utilizando dos repositorios remotos. Creo que es la mejor forma de **aprovechar las ventajas** que nos **ofrece cada** uno de los dos **repositorios**.

Añadir nuevos repositorios

Ya hemos visto cómo se pueden ver los repositorios remotos que tenemos referenciados. Si queremos **añadir un repositorio remoto**, ejecutaremos la siguiente orden:

```
git remote add [nombre] [url]
```

Por ejemplo:

```
git remote add launchpad git+ssh://lorenzo-carbonell@git.launchpad.net/my-weather-indicator
```

Donde:

- `[nombre]` es *launchpad*
- `[url]` es *git+ssh://lorenzo-carbonell@git.launchpad.net/my-weather-indicator*

Eliminar repositorios existentes

Otra situación que se nos puede dar es la necesidad de querer eliminar uno de estos repositorios remotos, sea por la circunstancia que sea. Para hacer esto, ejecutaremos la siguiente orden:

```
git remote remove [nombre]
```

Por ejemplo:

```
git remote remove launchpad
```

Donde:

- `[nombre]` es *launchpad* en este ejemplo.

Cambiar el nombre a repositorios existentes

Otra circunstancia que se te puede dar es querer cambiar el nombre un determinado repositorio que tienes añadido. Puedes hacerlo a lo bruto, es decir, quitar el repositorio y luego añadirlo, o bien utilizar la siguiente orden:

```
git remote rename [nombre] [nuevo_nombre]
```

Por ejemplo:

```
git remote rename launchpad launchpad_v2
```

Donde:

- `[nombre]` es *launchpad* en este ejemplo.
- `[nuevo_nombre]` es *launchpad_v2*.

Información de repositorios

Por último, nos queda obtener información de cada uno de los repositorios remotos que hemos añadido a nuestro repositorio local. Para hacer esto, ejecutaremos la siguiente orden:

```
git remote show [nombre]
```

Por ejemplo:

```
git remote show launchpad
```

Donde:

- `[nombre]` es *launchpad* en este ejemplo.

En el caso que nos ocupa, al ejecutar esta orden, el resultado que nos devuelve es el siguiente:

```
* remote launchpad
```

```
Fetch URL: git+ssh://lorenzo-carbonell@git.launchpad.net/my-
weather-indicator

Push URL: git+ssh://lorenzo-carbonell@git.launchpad.net/my-
weather-indicator

HEAD branch: master

Remote branch:

master tracked

Local ref configured for 'git push':

master pushes to master (up to date)
```

Sincronizar con los repositorios remotos

Una vez ya dominamos la gestión de los repositorios remotos, vamos a lo que realmente nos interesa que es la sincronización de nuestro repositorio local, con los repositorios remotos.

Descargando de los repositorios remotos

La primera de las operaciones a realizar es obtener la información de los repositorios remotos (*actualizar nuestras referencias a las ramas remotas*). Para esto, ejecutaremos la siguiente orden:

```
git fetch [nombre]
```

Por ejemplo:

```
git fetch origin
```

Donde,

- `[nombre]` es *origin* en este ejemplo.

Para este ejemplo, he modificado directamente en GitHub un archivo. Al ejecutar `git remote show origin` me muestra la siguiente información:

```
* remote origin

Fetch URL: git@github.com:atareao/my-weather-indicator.git

Push URL: git@github.com:atareao/my-weather-indicator.git
```

```
HEAD branch: master

Remote branch:

    master tracked

Local branch configured for 'git pull':

    master merges with remote master

Local ref configured for 'git push':

    master pushes to master (local out of date)
```

Indicándonos en la última línea que nuestro repositorio local **no está actualizado**. Vamos a obtener la información del repositorio remoto, utilizando la orden indicada anteriormente, `git fetch origin`. Al hacer esto vemos lo siguiente:

```
remote: Counting objects: 3, done.

remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0

Unpacking objects: 100% (3/3), done.

De github.com:atareao/my-weather-indicator

    548f4c7..b932d81  master    -> origin/master
```

Si ahora ejecutamos `git status` para ver la situación de nuestro repositorio nos encontraremos con el siguiente resultado:

```
En la rama master

Your branch is behind 'origin/master' by 1 commit, and can be
fast-forwarded.

    (use «git pull» para actualizar su rama local)

nothing to commit, working tree clean
```

Ahora el repositorio remoto está **accesible localmente**, pero todavía no lo hemos *unido* a nuestro repositorio local. De hecho, si revisamos los archivos modificados en remoto en nuestro repositorio local, veremos que **no están actualizados**. Para actualizarlos debemos ejecutar la orden `git pull`. En mi caso, ejecuto la siguiente orden:

```
git pull origin
```

Y me devuelve el siguiente resultado:

```
Updating 548f4c7..b932d81

Fast-forward

 test.txt | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)
```

Si ahora revisamos el contenido de los archivos modificados en remoto, veremos que coinciden con nuestros archivos locales.

Si la **rama** actual en la que estamos trabajando está configurada para trabajar con la **rama** remota, en lugar de hacer las dos operaciones secuenciales `git fetch` y `git pull` podemos hacerlo de un solo golpe, directamente con `git pull`. Pero, **¿Qué es esto de las ramas?**. *Esto lo veremos en el próximo artículo.*

Subiendo al repositorio remoto

Y ahora, ¿en qué situación se encuentra nuestro repositorio local respecto al **otro** repositorio remoto?. Es decir, al modificar el repositorio local, evidentemente, ha quedado desactualizado el otro repositorio remoto. Para comprobarlo ejecutaremos la orden `git remote show launchpad`, que nos devuelve el siguiente resultado:

```
* remote launchpad

Fetch URL: git+ssh://lorenzo-carbonell@git.launchpad.net/my-
weather-indicator

Push URL: git+ssh://lorenzo-carbonell@git.launchpad.net/my-
weather-indicator

HEAD branch: master

Remote branch:

master tracked

Local ref configured for 'git push':
```



```
master pushes to master (fast-forwardable)
```

Como podemos observar en la última línea ha pasado de decir (up to date) a decir (fast-forwardable). Pues vallamos allá, hagamos un push a este repositorio. Esto es tan sencillo como ejecutar la orden:

```
git push launchpad
```

Y si de nuevo, volvemos a comprobar el estado del repositorio remoto `launchpad` ejecutando la orden `git remote show launchpad` veremos el siguiente resultado:

```
* remote launchpad

Fetch URL: git+ssh://lorenzo-carbonell@git.launchpad.net/my-
weather-indicator

Push URL: git+ssh://lorenzo-carbonell@git.launchpad.net/my-
weather-indicator

HEAD branch: master

Remote branch:

master tracked

Local ref configured for 'git push':

master pushes to master (up to date)
```

Aquí hemos simplificado, porque la orden correcta es:

```
git push [nombre] [rama]
```

Donde:

- `[nombre]` es el nombre de nuestro repositorio remoto.
- `[rama]` es el nombre de la rama con la que queremos sincronizar.

5. Gestión de ramificaciones con Git

Como he comentado en la introducción, la gestión de ramificaciones es parte fundamental de cualquier sistema de control de versiones, permitiéndonos seguir diferentes caminos *de forma simultánea* para alcanzar el objetivo final.

De esta manera podemos añadir nuevas características a nuestro proyecto, ya sea un libro, una imagen o una aplicación... y ver el resultado. Una vez comprobado cómo ha quedado cada una de estas ramas en nuestro proyecto, **fusionaremos** cada una de las ramas con la **rama principal**.

Crear una nueva rama

Así, para crear una rama de nuestro proyecto, tan solo tenemos que ejecutar la siguiente orden:

```
git branch nueva_rama
```

Cambiando de rama

Para cambiar de esa rama principal a la rama `nueva_rama`, ejecutaremos la orden:

```
git checkout nueva_rama
```

Y si queremos volver a nuestra rama principal, de nuevo, ejecutaremos la orden:

```
git checkout master
```

¿En qué rama estamos?

Llegados a este punto, suponiendo que has creado varias ramas y estás trabajando en todas ellas de forma simultánea, por ejemplo probando diferentes técnicas para dar respuesta a un determinado *bug*, ¿cómo podemos saber en qué rama nos encontramos?

La solución se encuentra en ejecutar la orden:

```
git log --oneline
```

Este nos mostrará un *puntero* denominado `HEAD` que es el que nos señala dónde nos encontramos. Como muestra, en este ejemplo, al ejecutar la orden `git log --oneline` vemos el puntero `HEAD` en la rama `master`.

```
lorenzo@xps13:/datos/Sync/Programacion/Python/my-weather-indicator$ git log --oneline
612b1cf (HEAD -> master, origin/master) Merge branch 'Jalakas-master'
c031dab (Jalakas-master) Merge branch 'master' of https://github.com/Jalakas/my-weather-indicator into Jalakas-master
54071ee Merge branch 'http_connection'
a2b8930 (origin/http_connection, http_connection) added
1876090 Merge branch 'master' into master
37174d3 Merge branch 'master' into master
4a95088 setup.py: move to python3, update language generation * use --sort-by-files flag to generate POT, * remove lon
g filepaths from POT, * use proper msgfmt tool to generate MO files, * so msgfmt.py can be removed.
b33dcdb HTTP Connection
d9b632f (origin/check_connection, check_connection) Check connection
c0f3233 Check connection
5b3526e Check conectivity updated
0a499b7 flake8: varios fixes
35938db setup.py: flake8 fixes
2f3157 flake8: style fixes (whitespace around arithmetic operator, except:)
e09cb99 flake8: style fixes (indent, whitespace after ',')
b6dd00f translations
a12befc osm
1351eb9 Merge branch 'vpslavskyi-patch-1'
832fa71 (vpslavskyi-patch-1) Merge branch 'patch-1' of https://github.com/vpslavskyi/my-weather-indicator into vpsl
avskyi-patch-1
59bad61 update
a61eea7 Update uk.po
:...skipping...
612b1cf (HEAD -> master, origin/master) Merge branch 'Jalakas-master'
c031dab (Jalakas-master) Merge branch 'master' of https://github.com/Jalakas/my-weather-indicator into Jalakas-master
```

Si ahora cambiamos de rama `git checkout http_connection` y volvemos a ejecutar la orden `git log --oneline`, el resultado es el siguiente:

```
lorenzo@xps13:/datos/Sync/Programacion/Python/my-weather-indicator$ git checkout http_connection
Switched to branch 'http_connection'
Su rama está actualizada con «origin/http_connection».
lorenzo@xps13:/datos/Sync/Programacion/Python/my-weather-indicator$ git log --oneline
a2b8930 (HEAD -> http_connection, origin/http_connection) added
b33dcdb HTTP Connection
d9b632f (origin/check_connection, check_connection) Check connection
c0f3233 Check connection
5b3526e Check conectivity updated
b6dd00f translations
a12befc osm
1351eb9 Merge branch 'vpslavskyi-patch-1'
832fa71 (vpslavskyi-patch-1) Merge branch 'patch-1' of https://github.com/vpslavskyi/my-weather-indicator into vpsl
avskyi-patch-1
59bad61 update
a61eea7 Update uk.po
e5c76af Update test.txt
b086409 Local
c452089 Update test.txt
b932d81 Update test.txt
548f4c7 Merge branch 'master' of git+ssh://git.launchpad.net/my-weather-indicator
5f9d56d Merge branch 'master' of github.com:atareaao/my-weather-indicator
c0196e7 Osm
bdc5fb9 Merge pull request #4 from gitter-badger/gitter-badger
3d93ab3 ss
```

Si comparas las dos capturas de pantalla, te darás cuenta de que hemos vuelto hacia atrás. En el primer `HEAD` apuntaba al `commit` `612b1cf`, mientras que en la segunda captura de pantalla `HEAD` apunta a un `commit` anterior, el `a2b8930`.

¿Qué podemos hacer en cada rama?

En cada rama trabajaremos normalmente, como hacemos cuando estamos en la rama principal `master`. Nuestro procedimiento de trabajo es el habitual:

- añadimos, borramos o modificamos archivos.
- confirmamos los cambios realizados en nuestra rama ejecutando la orden `git add .`
- realizamos el `commit` ejecutando la orden `git commit -m "cambios realizados"`.

Todos estos cambios se realizan en la rama en la que nos encontramos, sin afectar al resto de ramas. De esta forma, si cambiamos a otra rama, no encontraremos esos cambios.

Una vez realizados todos los cambios necesarios en nuestra *ramificación*, y considerado que ésta ya es adecuada para pasarla a producción, ha llegado el momento de realizar la fusión. ¿Cómo hacemos este paso?

Fundiendo ramas

Para fundir una rama con la rama principal `master` o con cualquier otra rama, tenemos que proceder como sigue:

- nos movemos a la rama con la que queremos *fundirnos*.

Por ejemplo, si queremos fundir `nueva_rama` con la rama `master` nos desplazaremos a la rama `master`, ejecutando la orden `git checkout master`.

- realizamos la mezcla ejecutando la orden `git merge nueva_rama`

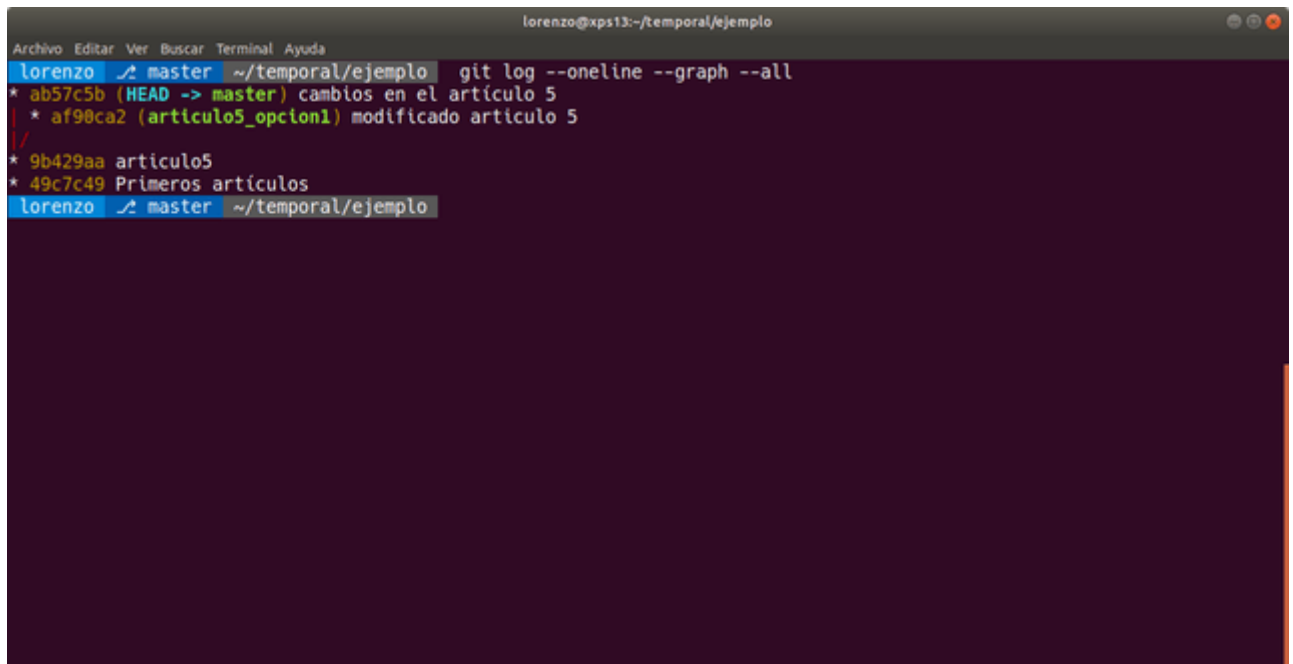
En general es tan sencillo como lo que acabo de comentar, salvo que haya conflictos en la fusión.

Conflictos de fusión

Para comprender esto de los conflictos de fusión vamos a partir de que tenemos nuestra rama principal `master` y una rama derivada `nueva_rama`. Cuando nos decidimos a realizar la fusión hemos realizado cambios tanto en la rama derivada `nueva_rama` como en la rama principal. Sin embargo, las

modificaciones realizadas en la rama principal se efectuaron posteriormente a haber realizado la ramificación.

En este caso, si hay cambios que afectan a un mismo archivo se producirá un **conflicto**. El sistema de control de versiones no sabrá que hacer, que parte del código o del archivo prevalecerá, si la correspondiente a la rama principal, o la correspondiente a la ramificación `nueva_rama`.

A screenshot of a terminal window with a dark background. The title bar at the top reads 'lorenzo@xps13:~/temporal/ejemplo'. The terminal shows the command 'git log --oneline --graph --all' being executed. The output is as follows:

```
lorenzo  master ~/temporal/ejemplo  git log --oneline --graph --all
* ab57c5b (HEAD -> master) cambios en el artículo 5
| * af90ca2 (articulo5_opcion1) modificado artículo 5
|/
* 9b429aa artículo5
* 49c7c49 Primeros artículos
lorenzo  master ~/temporal/ejemplo
```

En el ejemplo anterior, he modificado un artículo tanto en la rama principal `master` como en la ramificación `articulo5_opcion1`. Cuando la he fusionado con la rama principal `master`, me indica que hay conflictos y es necesario resolverlos.

Resolviendo conflictos

El primer paso es averiguar dónde están los conflictos que nos impiden concluir la fusión. Para ello, ejecutaremos la orden `git status`, tal y como puedes ver en la siguiente captura de pantalla:

```
lorenzo@xps13:~/temporal/ejemplo
Archivo Editar Ver Buscar Terminal Ayuda
lorenzo master ~/temporal/ejemplo git log --oneline --graph --all
* ab57c5b (HEAD -> master) cambios en el artículo 5
| * af90ca2 (articulo5_opcion1) modificado artículo 5
| /
* 9b429aa artículo5
* 49c7c49 Primeros artículos
lorenzo master ~/temporal/ejemplo git merge articulo5_opcion1
Automatezclado articulo5.md
CONFLICTO(contenido): conflicto de fusión en articulo5.md
Automatic merge failed; fix conflicts and then commit the result.
lorenzo master S:1 U:1 ~/temporal/ejemplo 1 git status
En la rama master
Tiene rutas sin fusionar.
(solucione los conflictos y ejecute «git commit»)
(use "git merge --abort" to abort the merge)

Rutas no combinadas:
(use «git add <archivo>...» para marcar resolución)

modificado por ambos: articulo5.md

no hay cambios agregados al commit (use «git add» o «git commit -a»)
lorenzo master S:1 U:1 ~/temporal/ejemplo
```

Vamos a editar `articulo5.md`, utilizando `nano` por ejemplo. El contenido del archivo en este caso será el siguiente:

```
<<<<<<< HEAD

Esta es la modificación que he realizado en la rama principal
después de

haber realizado la ramificación

=====

Esta es la opción 1

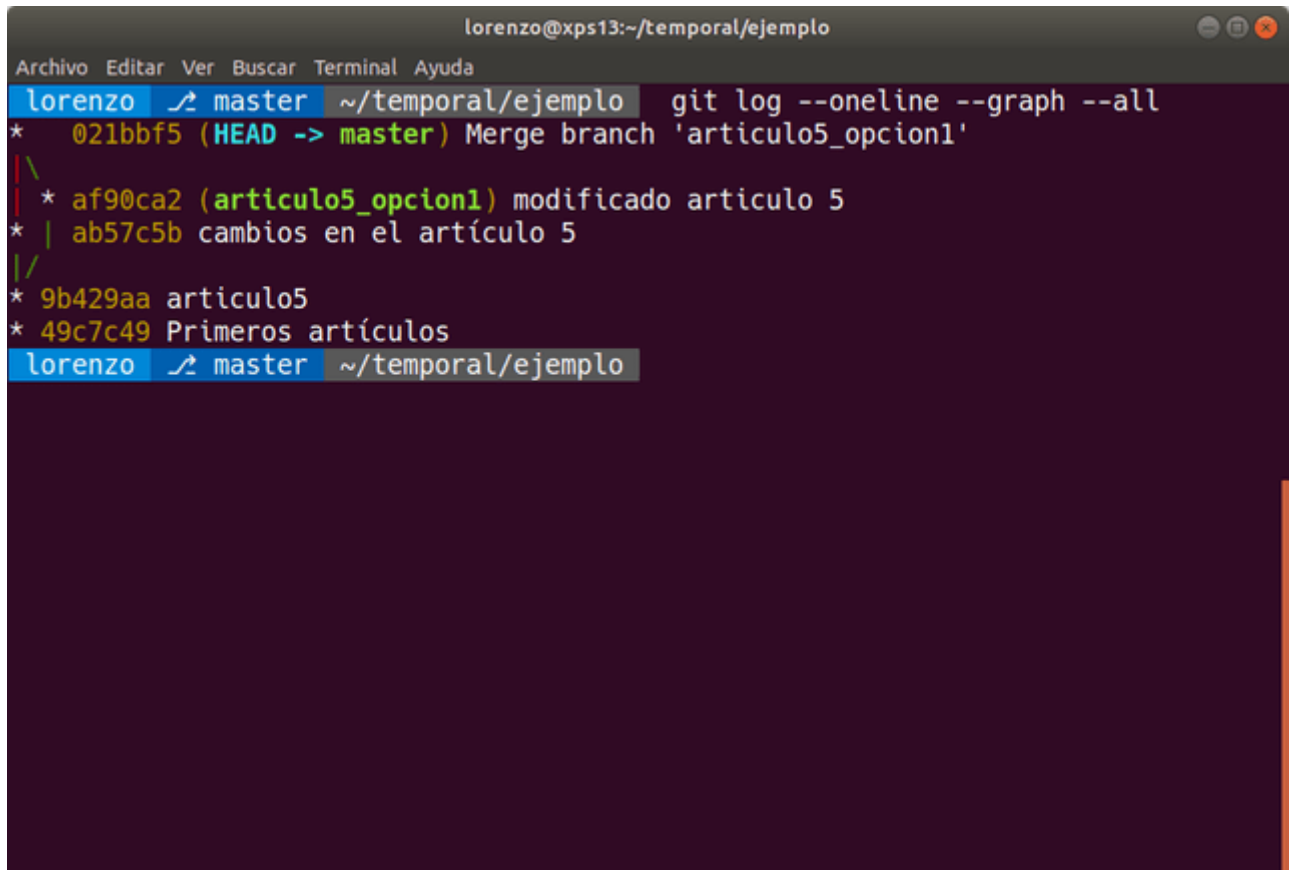
>>>>>>> articulo5_opcion1
```

Efectivamente esto es una combinación de la rama principal `master` y de la ramificación `articulo5_opcion1`. Debemos modificar el archivo, hasta

dejarlo como deseamos. Una vez terminada nuestra edición, procedemos como de costumbre:

```
git add .  
  
git commit
```

Ahora ya podemos ver el resultado, ejecutando la orden, `git log --oneline --graph --all`, en la siguiente captura de pantalla:



```
lorenzo@xps13:~/temporal/ejemplo  
Archivo Editar Ver Buscar Terminal Ayuda  
lorenzo master ~/temporal/ejemplo git log --oneline --graph --all  
* 021bbf5 (HEAD -> master) Merge branch 'articulo5_opcion1'  
|\   
* af90ca2 (articulo5_opcion1) modificado articulo 5  
* | ab57c5b cambios en el artículo 5  
|/  
* 9b429aa articulo5  
* 49c7c49 Primeros artículos  
lorenzo master ~/temporal/ejemplo
```

Conclusiones

Hasta aquí hemos visto todo lo necesario para **crear y fusionar ramas**. Con esto ya te puedes hacer una idea del potencial del control de versiones y sobre todo del **potencial de las ramificaciones**. Y como he comentado e insistido en la introducción, no se trata de algo exclusivo para los desarrolladores de aplicaciones, sino que se puede utilizar en cualquier campo.

Gestión de ramificaciones

El comando `git branch`

Hasta el momento, lo que habíamos realizado con este comando era única y exclusivamente la creación de una nueva rama con `git branch nueva_rama`. Sin embargo, este comando nos proporciona más opciones interesantes,

- Mostrar un listado de las ramas del proyecto. Para ello, ejecutaremos la orden `git branch`. En la siguiente captura de pantalla verás el resultado de ejecutarlo. Si te fijas, delante de una de las ramas vemos un `*` y, además, está en otro color. Esto nos indica que la rama señalada es la **rama activa**.
- Si ejecutamos la orden `git branch -v`, además de esto, nos mostrará el último *commit* que hemos realizado.
- Por otro lado, tenemos la opción de conocer cuáles de las ramas presentes están fusionadas, y cuáles no. Para ello, utilizaremos las ordenes `git branch --merged` y `git branch --no-merged`, para ver las ramas fusionadas y las no fusionadas, respectivamente.

Flujo de trabajo con ramificaciones

Supongamos que hemos desarrollado una aplicación y hemos liberado la versión 1.0. Ésta se encuentra en nuestro sistema de control de versiones. Tras la celebración posterior al éxito conseguido, xD, pongamos que decidimos incorporar una serie de mejoras a la aplicación.

La rama de desarrollo

Al pensar en esa nueva versión, la 2.0, decidimos, evidentemente, seguir utilizando nuestro sistema de control de versiones Git. Y para ello, creamos una nueva rama. Esta nueva rama, sobre la que vamos incorporando las mejoras, no tiene por qué ser completamente funcional en un principio.

Aunque, finalmente, en el momento en que la *fusionemos* con la rama principal, sí deberá serlo, claro.

Esta rama de **desarrollo**, a su vez, puede tener **pequeñas ramificaciones**, bien para añadir nuevas **características**, o bien para intentar diferentes alternativas que, una vez aprobadas, se *fusionarán*.

La rama principal o rama de producción

Pero, ¿porque no editar directamente la *rama principal*? Nosotros nos las habíamos dado muy felices cuando liberamos la versión 1.0. Creíamos que habíamos hecho una aplicación libre de fallos. Pero, nuestra **amiga Realidad**, nos rescató y nos mostró que teníamos varios errores, fallos, *bugs*, *bichitos*...

Los usuarios de la versión 1.0, no tan felices por la aplicación, comenzaron una campaña para que resolviéramos esos errores, urgentemente... Menos mal que dejamos intacta la rama principal o **rama de producción** porque, de otra forma **¿cómo localizar el fallo en un código que ya hemos modificado?**

Corrigiendo errores

De esta manera, cada error que un usuario de nuestra versión 1.0 nos reporta, se materializa en una **nueva rama** de la **rama principal**. Una vez resuelto el problema, la fusionaremos a ésta.

Es conveniente borrar estas ramas que, simplemente, sirven para corregir errores. Borrar una rama es sencillo: `git branch -d ramaaborrar`.

Fusión

Llegados a este punto, y una vez estamos seguros de que la rama de desarrollo es estable y podemos lanzarla a producción, realizamos la **fusión**.

Etiquetas

¿Y esto de las versiones? ¿Cómo podemos llevar un control de la versión de nuestra aplicación, libro o lo que sea? En Git, al igual que en otros **CMS**, disponemos de **etiquetas**.

Las **etiquetas** nos **sirven para** lo que hemos visto hasta ahora, **indicar** dónde hemos *liberado una versión, cambios importantes en el código*, o lo que consideremos importante.

Trabajando con etiquetas...

Disponemos de una serie de herramientas que nos permiten trabajar con etiquetas:

- `git tag` **lista** las etiquetas existentes.
- `git tag etiqueta` **crea** una etiqueta ligera llamada `etiqueta`.
- `git tag -a etiqueta -m "anotación"` crea una etiqueta anotada llamada `etiqueta` con una anotación. Si no añades la opción `-m`, abrirá el editor de texto que tengas configurado.
- `git show etiqueta` muestra los **datos** asociados con la etiqueta `etiqueta`.
- `git -s etiqueta -m "anotación"` crea una etiqueta firmada con tu firma GPG.
- `git tag -v etiqueta` nos permite verificar una etiqueta. firmada. Debemos disponer de la clave GPG pública del autor para poder verificarla.

También es posible etiquetar un `commit` posteriormente.

Las etiquetas y los servidores remotos

Por último, indicar que la orden `git push` **no sube las etiquetas a un servidor remoto**, es necesario subirlas de forma explícita. Para ello utilizaremos alguna de las siguientes órdenes:

- `git push origin etiqueta` para subir una etiqueta en concreto.
- `git push origin --tags` para subir todas las etiquetas.