GUÍAS DE USO

**GUÍA INICIAL DE GIT Y GITHUB**

ÍNDICE

[INTRODUCCIÓN 4](#_Toc151803416)

[A. ¿QUÉ ES GIT? 4](#_Toc151803417)

[B. INSTALACIÓN DE GIT EN NUESTROS EQUIPOS 5](#_Toc151803418)

[C. COMPROBACIÓN DE INSTALACIÓN CORRECTA 6](#_Toc151803419)

[CREANDO NUESTRO DIRECTORIO DE TRABAJO 9](#_Toc151803420)

[CONFIGURACIÓN DE GIT (git config) 9](#_Toc151803421)

[INICIALIZANDO NUESTRO REPOSITORIO (git init) 11](#_Toc151803422)

[CONCEPTO DE RAMA EN GIT 12](#_Toc151803423)

[AÑADIR ARCHIVOS (git add) Y GUARDAR CAMBIOS (git commit) 13](#_Toc151803424)

[VISUALIZACIÓN DEL HISTORIAL DE COMMITS (git log) 15](#_Toc151803425)

[CANCELAR CAMBIOS SOBRE UN FICHERO (git checkout) O SOBRE TODOS LOS FICHEROS (git reset) 16](#_Toc151803426)

[PERSONALIZACIÓN DEL LOG Y GENERACIÓN DE ALIAS (git alias) 18](#_Toc151803427)

[ARCHIVO .gitignore 20](#_Toc151803428)

[COMPROBAR QUÉ SE HA CAMBIADO DESDE EL ÚLTIMO COMMIT (git diff) 21](#_Toc151803429)

[DESPLAZAMIENTO DENTRO DE UNA RAMA (git checkout) 22](#_Toc151803430)

[VUELTA A UN ESTADO ANTERIOR (git reset --hard) Y LOG AMPLIADO (git reflog) 25](#_Toc151803431)

[GENERACIÓN DE ETIQUETAS (git tag) 28](#_Toc151803432)

[CREACIÓN DE NUEVAS RAMAS (git branch) Y MOVIMIENTO ENTRE RAMAS (git switch) 32](#_Toc151803433)

[FUSIÓN DE RAMAS (git merge) 35](#_Toc151803434)

[A. CONFLICTOS EN LA FUSIÓN DE RAMAS 37](#_Toc151803435)

[ALMACENAR TEMPORALMENTE CAMBIOS (git stash) 40](#_Toc151803436)

[REINTEGRACIÓN DE RAMAS 43](#_Toc151803437)

[ELIMINACIÓN DE RAMAS EN GIT 44](#_Toc151803438)

[GITHUB 46](#_Toc151803439)

[A. ¿QUÉ ES GITHUB? 46](#_Toc151803440)

[B. PRIMEROS PASOS EN GITHUB. CONOCIENDO LA INTERFAZ 47](#_Toc151803441)

[C. CREANDO UN REPOSITORIO 47](#_Toc151803442)

[D. LOCAL Y REMOTO 50](#_Toc151803443)

[ENLAZAR UN REPOSITORIO LOCAL CON UN REPOSITORIO REMOTO (git remote) 51](#_Toc151803444)

[SUBIR EL REPOSITORIO LOCAL AL REPOSITORIO REMOTO (git push) 51](#_Toc151803445)

[SUBIDA DE CAMBIOS A GITHUB (git push) 52](#_Toc151803446)

[COMPROBACIÓN DE EXISTENCIA DE CAMBIOS EN REMOTO (git fetch) Y ACTUALIZACIÓN DEL REPOSITORIO LOCAL CON LOS CAMBIOS REMOTOS (git pull) 55](#_Toc151803447)

[DESCARGA DE UN REPOSITORIO REMOTO (git clone) 58](#_Toc151803448)

[PROTEGIENDO NUESTRO PROYECTO FRENTE A CAMBIOS 59](#_Toc151803449)

[CONCEPTO DE FORK EN GITHUB 61](#_Toc151803450)

[FLUJO COLABORATIVO EN GITHUB 64](#_Toc151803451)

[PULL REQUEST EN GITHUB 65](#_Toc151803452)

[SINCRONIZACIÓN DE UN FORK EN GITHUB 69](#_Toc151803453)

[USO DE HERRAMIENTAS GRÁFICAS PARA GIT Y GITHUB 72](#_Toc151803454)

[A. GITHUB DESKTOP 72](#_Toc151803455)

[B. GITKRAKEN 72](#_Toc151803456)

[C. SOURCETREE 73](#_Toc151803457)

[D. FORK 73](#_Toc151803458)

[GLOSARIO DE COMANDOS - GIT/GITHUB CHEATSHEET 75](#_Toc151803459)

# INTRODUCCIÓN

## ¿QUÉ ES GIT?

Logotipo, Icono

Descripción generada automáticamenteGit es un sistema de control de versiones distribuido ampliamente utilizado para el seguimiento de cambios en proyectos de software. Fue creado por Linus Torvalds en 2005 y se ha convertido en una herramienta fundamental en el desarrollo de software colaborativo. Algunas de las características y funcionalidades clave de Git incluyen:

1. **Control de versiones**: Git permite mantener un historial detallado de todos los cambios realizados en un proyecto. Cada modificación se registra, lo que facilita la revisión de versiones anteriores, la identificación de errores y la gestión de cambios.
2. **Distribuido**: Git es un sistema de control de versiones distribuido, lo que significa que cada colaborador de un proyecto tiene una copia completa del repositorio, incluido su historial de cambios. Esto permite trabajar de manera independiente, sin necesidad de una conexión constante a un servidor central.
3. **Ramificación (*branching*) y Fusión (*merging*)**: Git facilita la creación de ramas (*branches*) para trabajar en funcionalidades o correcciones de errores de forma aislada. Una vez que se completa el trabajo en una rama, se puede fusionar (*merge*) con la rama principal o con otra rama, combinando los cambios de manera controlada.
4. **Gestión eficiente de conflictos**: Cuando se fusionan ramas con cambios concurrentes, Git ayuda a manejar conflictos que pueden surgir entre diferentes versiones del mismo archivo, permitiendo al usuario resolverlos de manera manual.
5. **Historial detallado**: Cada modificación en un proyecto se registra con un hash único que permite identificarla fácilmente. Esto incluye quién hizo el cambio, cuándo se realizó y un mensaje descriptivo que explica la naturaleza del cambio.
6. **Repositorios remotos**: Git facilita la colaboración al permitir la conexión y sincronización con repositorios remotos, como GitHub, GitLab o Bitbucket, lo que posibilita compartir el trabajo con otros colaboradores y mantener una copia centralizada del proyecto.
7. **Herramientas y comandos flexibles**: Git ofrece una amplia gama de comandos para realizar diferentes tareas, como commits, revertir cambios, etiquetar versiones, entre otros. También se integra con numerosas herramientas y entornos de desarrollo.

En resumen, Git se ha convertido en una herramienta esencial para el desarrollo de software moderno debido a su capacidad para mantener un historial detallado de cambios, facilitar la colaboración entre equipos y permitir un manejo eficiente de versiones en proyectos de aplicaciones web, de escritorio y otros tipos de software.

## INSTALACIÓN DE GIT EN NUESTROS EQUIPOS

Para instalar Git en nuestros equipos (en este caso lo haremos sobre un equipo Windows) debemos dirigirnos a la [web oficial](https://git-scm.com/) y descargarnos la versión que queramos utilizar. Actualmente la última versión disponible es la 2.42.1 y la última versión estable la 2.42.0 (esta será la versión que utilicemos a lo largo de la guía).

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Una vez descargada la versión que queramos simplemente tendremos que instalarla en nuestro equipo a través del instalador .exe que hayamos descargado.

En esta web vamos a encontrar también documentación muy útil para conocer el funcionamiento de Git, incluido el libro [Pro Git](https://git-scm.com/book/es/v2) el cual es considerado una muy buena guía de inicio.

Una vez instalado tenemos varias opciones para utilizarlo:

* Podemos utilizar la aplicación Git Bash que se nos habrá instalado y que consiste en una terminal de comandos personalizada para Git.
* También podemos utilizar la aplicación Git Gui que consiste en una aplicación gráfica para la gestión de repositorios.
* Por último, podremos utilizar la propia CMD o PowerShell para utilizar Git. Esta será la opción que utilizaremos en esta guía.

## COMPROBACIÓN DE INSTALACIÓN CORRECTA

Para comprobar que la instalación ha sido correcta y que nuestro sistema detecta que hemos instalado Git podemos utilizar el comando:

git --version (o git -v)

Imagen que contiene firmar, calle, medidor

Descripción generada automáticamente

Imagen de la pantalla de un celular de un mensaje en letras blancas

Descripción generada automáticamente con confianza baja

Si todo se ha instalado de forma correcta deberemos recibir el mensaje que vemos en las imágenes, el cual nos informa de que tenemos instalada la versión 2.42.0 de git.

En el caso de que no nos lo detectara podría deberse a un problema de variables de entorno en cuyo caso deberíamos ir al editor de variables de entorno (buscando “variables de entorno” en nuestro menú de Windows y haciendo clic en “Editor de variables de entorno”). En la ventana de propiedades del sistema, debemos hacer clic en “Variables de entorno…”:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Y en la nueva la nueva ventana que se abre buscar en las variables del sistema, la variable Path:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Hacemos clic en “Editar…” y vemos si tenemos la variable de entorno de Git, en caso de no tenerla, debemos añadirla (lo que escribiremos será la ruta hacia el directorio \cmd de la carpeta donde hayamos instalado Git) con el botón “Nuevo”:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Otro comando que podemos utilizar es simplemente el comando git:

git

Texto

Descripción generada automáticamente

Aquí podremos ver una especie de mini-manual de los principales comandos con los que contamos en git.

Otros comandos útiles son:

* git help -a: Nos proporciona un listado completo de todas las opciones disponibles en git y una pequeña descripción de su función.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

* git help -g: Se trata de un listado de guías de uso como un FAQ, un glosario, una guía de inicio, etc.

Texto

Descripción generada automáticamente

* git help <opción> o git help <guía>: Con este comando desplegaremos un manual sobre documento .html acerca del comando o la guía de uso que queramos ver (similar a lo que sería el man <comando> de Linux). Por ejemplo, esta es la web para el comando git add:



Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

# CREANDO NUESTRO DIRECTORIO DE TRABAJO

Vamos a crearnos un directorio donde crearemos los diferentes archivos que utilizaremos para crear nuestro repositorio, en este caso lo vamos a crear en el Escritorio con el nombre “Hello Git”:

mkdir “nombre del directorio”

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Con esto habremos creado un directorio que no contiene nada, vamos a desplazarnos dentro de la carpeta (comando cd ‘.\Hello Git\’) y a lanzar Visual Studio Code el cual vamos a poder lanzar con el comando code . si lo tenemos instalado:

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

Pantalla de computadora con fondo negro

Descripción generada automáticamente con confianza baja

# CONFIGURACIÓN DE GIT (git config)

Antes de continuar con nuestro repositorio vamos a configurar nuestro Git para que lo podamos utilizar con nuestro proyecto. En Git todo lo que hagamos debe estar asociado a alguien, tenemos que pensar que un proyecto puede que no sea el trabajo de una única persona sino de un equipo de trabajo, de forma que si queremos tener un verdadero control de versiones, debemos saber en todo momento no sólo que cambios se han hecho de versión a versión sino quién los ha ido realizando.

Además de esta forma se consigue un extra de seguridad ya que evitaremos que gente externa al proyecto pueda realizar cambios en el proyecto.

Por tanto, siempre va a haber que haber una asociación Git – Usuario y lo mínimo que necesitamos es el nombre de usuario y el email sin esto Git no nos va a permitir hacer absolutamente nada.

Para crear el usuario usaremos el siguiente comando:

git config --global user.name “Nombre del usuario”



La opción --global nos permite hacer que esta configuración nos sirva para cualquier directorio o proyecto en el que vayamos a trabajar y no únicamente en un proyecto concreto.

Esto nos debería haber creado un archivo .gitconfig en el directorio del usuario (por lo general, C:\Users\NombreUsuario), si abrimos el fichero veremos que tiene el siguiente aspecto:

Una captura de pantalla de un celular

Descripción generada automáticamente

Para crear el mail usamos un comando similar:

git config --global user.email “email del usuario”



Esto habrá modificado el archivo .gitconfig y ahora tendremos la información del email asociada:

Una captura de pantalla de un celular

Descripción generada automáticamente

Con esto ya tendremos configurado nuestro git y podremos empezar a trabajar con Git.

# INICIALIZANDO NUESTRO REPOSITORIO (git init)

Dentro de nuestro editor de código (en nuestro caso Visual Studio Code) vamos a crear un nuevo fichero dentro del directorio “Hello Git” que creamos en pasos previos. En este caso vamos a crear un fichero hellogit.py con un simple print:

Imagen de la pantalla de un celular

Descripción generada automáticamente con confianza baja

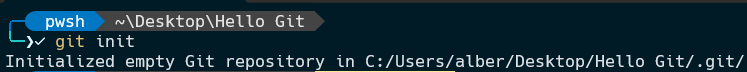
Si ahora hacemos un ls de nuestro directorio veremos el archivo creado:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Para inicializar el contexto de un control de versiones en este directorio debemos usar el comando:

git init



Esto nos habrá creado un directorio oculto de nombre .git:

Texto

Descripción generada automáticamente

Dentro de este directorio se han creado una serie de archivos y directorios que van a ser los responsables de que tengamos un control de versionado de nuestro proyecto:

Texto

Descripción generada automáticamente

# CONCEPTO DE RAMA EN GIT

Un concepto importante de Git son las ramas (*branches*) que no dejan de ser divisiones de un proyecto. Por ejemplo, si hablamos de un proyecto web podríamos tener una rama donde trabajarían los encargados del front-end y una rama donde trabajarían los encargados del back-end.

En realidad, esto es más complejo y en el día a día veremos situaciones donde a partir de una rama podremos tener nuevas ramas las cuales podrán dar lugar a nuevas ramas.

La finalidad de estas ramificaciones es conseguir separar los flujos del proyecto y que cada profesional del proyecto pueda trabajar en su rama.

De base siempre se creará una rama, la rama Master, que será el origen de todo el proyecto:

Gráfico, Gráfico de burbujas

Descripción generada automáticamente

Esta rama también puede ser llamada como Main o Trunk según el sistema, por ejemplo, si decidimos realizar un repositorio en GitHub y trabajar en todo momento allí, veremos que la rama se llama main.

Aun así, si queremos cambiar el nombre de la rama lo podremos hacer con el comando:

git branch -m nuevoNombre

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

También podemos configurar nuestro .gitconfig para que siempre que creemos un repositorio la rama inicial tenga un determinado nombre, para ello usaremos el comando:

git config --global init.defaultBranch nombreDeLaRamaInicial



Esto realizará una modificación de nuestro archivo .giconfig añadiendo este nuevo parámetro:



# AÑADIR ARCHIVOS (git add) Y GUARDAR CAMBIOS (git commit)

Ahora vamos a comenzar con el verdadero punto fuerte de Git, la generación de versiones en base a los cambios realizados en el proyecto. Lo que va a hacer Git cada vez que realizamos un cambio y queremos almacenarlo como una nueva versión es una “foto” del proyecto (similar a lo que ocurre cuando se hacen *snapshots* en una virtualización) que reciben el nombre de commits. De esa forma almacena el estado del proyecto en un determinado momento y bajo un determinado nombre.

En apartados anteriores hemos realizado cambios en nuestro proyecto, hemos generado un archivo .py, vamos ahora a guardar estos cambios generando con ello un nuevo commit.

Pero primero vamos a ver un nuevo comando que nos va a servir para conocer el estado de nuestro repositorio:

git status

Texto

Descripción generada automáticamente

Con este comando vamos a poder ver en qué rama nos encontramos, qué commits se han realizado en ella y qué archivos se encuentran sin seguimiento (*untracked*). Para que estos archivos pasen de este estado a uno de seguimiento se deben añadir y para ello debemos utilizar el siguiente comando:

git add nombreFichero[[1]](#footnote-2)



Si ahora realizamos una nueva inspección del estado veremos que se han producido cambios:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

El archivo hellogit.py ha pasado de estar sin seguimiento a tenerlo (ahora está en estado *staged*), es momento por tanto de hacer nuestro primer comando del proyecto:

git commit

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

Esto hará que se nos habrá una ventana de vim:

Texto

Descripción generada automáticamente

Tenemos que crear un comentario que describa a este commit. De nuevo, esto es una medida de seguridad extra de forma que para realizar ese cambio en la rama en la que estamos trabajando, debemos explicitar qué es lo que hemos cambiado.

Pantalla negra con letras blancas

Descripción generada automáticamente

Si guardamos cambios se realizará el commit, aun así en este caso no vamos a guardarlo y volvemos a la terminal porque vamos a explicar la otra forma que tenemos de denominar a los commits:

git commit -m “Texto del commit”

Imagen que contiene golpear, objeto, pelota, reloj

Descripción generada automáticamente

Una vez que lancemos el comando veremos el siguiente mensaje en nuestra terminal:

Texto

Descripción generada automáticamente

Que significa lo siguiente que en la rama main se ha generado un commit con un hash único 28d9310 y el nombre “Este es mi primer commit”. Además, nos informa de que este commit se produce porque se ha producido porque se ha cambiado 1 fichero y se le ha añadido 1 línea de código.

Vamos a lanzar un nuevo status:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Como podemos ver el mensaje de que no hay commits en este proyecto ya no nos aparece y, además, se nos informa de que no queda nada por hacer commit, es decir, que no quedan cambios por registrar.

# VISUALIZACIÓN DEL HISTORIAL DE COMMITS (git log)

Hemos realizado un commit sobre nuestro proyecto y ahora vamos a poder ver ya el historial de commits, para ello debemos usar el comando:

git log

Texto

Descripción generada automáticamente

Lo que nos informa este log es que tenemos un commit con un hash y una clave únicos, realizada en la rama main por X autor, en Y fecha y con N nombre.

Por esto debíamos hacer la configuración de git, sin esta configuración no habríamos podido realizar ningún commit al no podernos identificar.

De esta forma podemos tener un registro completo de todos los cambios que se vayan realizando en nuestro proyecto.

Vamos a crear un nuevo fichero en nuestro proyecto, por ejemplo, hellogit2.py:

Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media

Si ahora comprobamos el estado del proyecto veremos que tenemos este nuevo archivo sin ningún seguimiento activo:

Texto

Descripción generada automáticamente

Vamos a realizar un commit para guardar el estado del proyecto tras la creación de este fichero:

Imagen que contiene texto, firmar, calle, diferente

Descripción generada automáticamente

Si ahora hacemos un nuevo git log veremos los 2 commits:

Texto

Descripción generada automáticamente

Es decir, lo que tenemos ahora mismo es esto:

Diagrama

Descripción generada automáticamente

Sobre la rama main tenemos 2 commits uno primero que tiene el nombre de “Este es mi primer commit” que tiene el archivo hellogit.py y otro que tiene el nombre de “Este es mi segundo commit” que además de ese archivo tiene el archivo hellogit2.py.

# CANCELAR CAMBIOS SOBRE UN FICHERO (git checkout) O SOBRE TODOS LOS FICHEROS (git reset)

Vamos a editar el archivo hellogit.py, por ejemplo, vamos a cambiar el mensaje que imprime por pantalla:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Si ahora hacemos un status del proyecto veremos que este cambio se ha registrado:

Texto

Descripción generada automáticamente

Ahora este git status no nos informa de un nuevo fichero sino de que se han realizado cambios sobre un fichero el cual no ha sido almacenado para realizar un commit.

Vamos a modificar también el segundo fichero:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Al hacer un nuevo status veremos que este cambio registrado:

Texto

Descripción generada automáticamente

Supongamos que tras hacer estos cambios vemos que nuestro código no funciona como antes o que no nos gusta el resultado final y queremos volver a como lo teníamos antes.

Vamos a volver atrás en el fichero hellogit2.py, para ello deberemos usar la opción checkout de git. Esta opción nos va a permitirnos mover por estados de un fichero:

git checkout nombreFichero

Texto

Descripción generada automáticamente

Esto ha hecho que nuestro fichero haya vuelto a estar en su estado previo, el estado en el que se guardó en el último commit:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Vamos a hacer lo mismo con el otro fichero:

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Sitio web

Descripción generada automáticamente

También tenemos la opción de hacer un git reset con ello ambos ficheros habrían vuelto al estado previo al mismo tiempo ya que con este comando se cancelan todos los cambios realizados desde el último commit.

# PERSONALIZACIÓN DEL LOG Y GENERACIÓN DE ALIAS (git alias)

Llegados a este punto vamos a realizar un tercer commit y para ello vamos a modificar el primer archivo:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Imagen que contiene texto, firmar, calle, ciudad

Descripción generada automáticamente

Si ahora hacemos un log veremos los 3 commits:

Texto

Descripción generada automáticamente

Ahora vamos a ver cómo podríamos ver este log de una forma algo más gráfica por ejemplo con puntos que unan una rama:

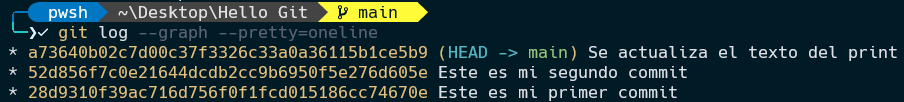
git log --graph

Texto

Descripción generada automáticamente

De esta forma tenemos un mejor feedback de lo que ha ido pasando en la rama. También podemos hacer que se vea la información principal de cada log en una línea:

git log --graph --pretty=oneline



En este caso vemos simplemente información única de cada commit obviando el usuario que lo ha realizado y la fecha.

Podemos incluso abreviarlo más:

git log --graph --decorate --all --oneline

Texto

Descripción generada automáticamente

Así solo veremos la información del hash único sin la clave única.

Pero estamos empezando a manejar comandos que son largos y “cansados” de escribir, Git para esto nos permite generar alias en el archivo de configuración:

git config --global alias.nombreAlias “comando a ejecutar”

Por ejemplo, en nuestro caso el comando se va a llamar tree y va a ejecutar el comando anterior:



Esto ha modificado nuestro archivo .gitconfig generando una nueva sección:



Si ahora lanzamos el comando git nombreAlias (en nuestro caso tree) veremos la ejecución asignada:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

# ARCHIVO .gitignore

Este fichero nos va a servir para evitar que ciertos archivos (ya sean ficheros o directorios) se añadan al stage.

Para entender cómo funciona esto vamos a crear un archivo de workspace de Visual Studio Code[[2]](#footnote-3):

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Si ahora hacemos un status del proyecto veremos que este archivo no tiene seguimiento:

Texto, Sitio web

Descripción generada automáticamente

Pero a diferencia de lo que ocurre con los ficheros hellogit.py y hellogit2.py, este fichero no tiene ningún sentido que se guarde en un commit porque es un fichero que guarda datos acerca de cómo queremos que se visualice este proyecto en nuestro Visual Studio Code por ello nunca lo guardaremos en el stage y siempre nos aparecerá como “sin seguimiento” cuando hagamos un status.

Para evitar esto vamos a crear un archivo con el nombre .gitignore (siempre se debe llamar así y siempre se debe añadir en la raíz del proyecto) en nuestro proyecto:

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

Ahora los ficheros o rutas que añadamos dentro de este fichero nunca se tomarán como archivos sin seguimiento, aunque sufran cambios:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Aquí también podemos usar expresiones como por ejemplo \*.class esto haría que todos los archivos .class de un proyecto basado en Java nunca se contaran como ficheros sin seguimiento.

Si ahora hacemos un git status veremos que nos aparece el archivo .gitignore como archivo sin seguimiento pero el archivo del workspace ya no aparecerá:

Texto

Descripción generada automáticamente

Vamos a hacer un commit para este .gitignore:

Imagen que contiene texto, firmar, calle, diferente

Descripción generada automáticamente

Si ahora volvemos a hacer un status veremos que no hay nada para meter en el stage:

Interfaz de usuario gráfica, Texto, Correo electrónico

Descripción generada automáticamente

# COMPROBAR QUÉ SE HA CAMBIADO DESDE EL ÚLTIMO COMMIT (git diff)

Hasta el momento hemos generado un total de 4 commits y nos encontramos en el último el commit donde hemos añadido el .gitignore.

Vamos a cambiar el archivo hellogit.py una vez más:

Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media

Imaginemos el supuesto de que hemos realizado muchos más cambios y queremos saber realmente qué hemos cambiado porque, por ejemplo, nuestra aplicación haya dejado de funcionar.

Para ver qué líneas y archivos se han visto modificados desde el último commit tenemos el comando:

git diff

Texto

Descripción generada automáticamente

Lo que nos dice este mensaje es que en el archivo hellogit.py se han producido cambios ha desaparecido una línea (-print(‘New Hello Git!’)) y ha aparecido una nueva línea (+print(‘New Hello Git with changes!’)).

De esta manera sin hacer un commit podríamos ver qué cambios se han realizado sobre los ficheros de nuestro proyecto desde el último commit.

# DESPLAZAMIENTO DENTRO DE UNA RAMA (git checkout)

Si pudiéramos el estado de nuestro proyecto a nivel de commits de forma gráfica veríamos algo similar a esto:

Diagrama

Descripción generada automáticamente

O lo que es lo mismo:

Texto

Descripción generada automáticamente

Supongamos el caso de que queremos volver al estado en el que se encontraba nuestro proyecto al hacer el primer commit para hacer esto debemos hacer un git checkout similar al que usamos para volver atrás en un fichero, pero en este caso en lugar de movernos en un fichero, nos vamos a mover en la rama y para ello utilizaremos el hash único del commit:

git checkout hashÚnico

Texto

Descripción generada automáticamente

Pero Git no nos ha dejado cambiar en este momento porque existe una modificación de hellogit.py que a la que no se le ha hecho un commit. Para evitar esto y como los cambios realizados no nos van a servir de nada, vamos a hacer un git checkout del propio fichero:

Texto

Descripción generada automáticamente

Y ahora sí que podremos movernos con el comando anterior:

Texto

Descripción generada automáticamente

Si ahora nos vamos a nuestro proyecto veremos que los archivos hellogit2.py y .gitignore han desaparecido:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Esto se debe a que hemos vuelto atrás en el tiempo, tenemos el proyecto en el mismo estado en el que se encontraba cuando hicimos el primer commit (el archivo de workspace aunque se creó después, como nunca ha sido asociado a una “foto” en concreto, permanece).

Es decir, de forma gráfica sería algo similar a esto, al principio antes del git checkout hash nos encontrábamos en el último commit:

Diagrama

Descripción generada automáticamente

Tras hacer el checkout nos encontramos aquí:

Diagrama

Descripción generada automáticamente

Hemos regresado a un punto anterior del proyecto.

Si ahora hacemos un git log veremos que sólo tenemos el primero de los commits:

Texto

Descripción generada automáticamente

Con esto podríamos indicar que esta situación es la nueva cabeza (HEAD) del proyecto de forma que borramos todo lo que hicimos posteriormente, una situación donde esto es útil es por ejemplo que tras hacer varios añadidos a nuestro código vemos que la aplicación ya no funciona como antes, va más lenta o ha dejado de funcionar, en lugar de ir cambiando modificaciones podemos volver a un estado anterior y reiniciar el proyecto desde ahí.

Si hacemos un git tree ahora podremos ver dónde nos encontramos (HEAD) con respecto a la situación real de la rama (main):

Texto

Descripción generada automáticamente

Para volver a la situación del final de la rama debemos volver a ejecutar el git checkout pero con el hashÚnico del último commit:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

En comparación al comando anterior donde hemos utilizado el hash con la clave única, ahora hemos utilizado únicamente el hash, como nuestro proyecto es tan pequeño lo podemos hacer sin problemas ya que cada commit es reconocible sólo con el hash.

Si ahora volvemos a hacer un git tree veremos que el puntero que apunta al final de la rama (main) y nosotros (HEAD) nos encontramos en el mismo punto:

Texto

Descripción generada automáticamente

Y en nuestro proyecto volvemos a tener esos archivos que se habían “perdido”:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Es decir, en cada commit lo que ha hecho Git es guardar una “foto” del estado del proyecto como hemos venido comentando, pero esa foto no es únicamente un estado general de los ficheros sino los ficheros en sí. Estas situaciones se van a almacenar en ese directorio .git que generamos al inicio de nuestro proyecto al utilizar el comando git init.

Por último, vamos a ver como nos podríamos colocar no sólo en el commit final sino el volver a “adjuntar” el HEAD al main ya que actualmente se encuentra separada (*detached*) para ello debemos usar otra variante del comando git checkout:

git checkout nombreRama



De esta forma nos situamos no sólo en el commit final, sino que nos colocamos al mismo nivel y enlazados con el main (HEAD 🡪 main).

Imagen que contiene texto, firmar, calle, grande

Descripción generada automáticamente

# VUELTA A UN ESTADO ANTERIOR (git reset --hard) Y LOG AMPLIADO (git reflog)

En anteriores apartados vimos el comando git reset que nos permitía eliminar los cambios realizados desde el último commit, tenemos un modificador para este comando (--hard) que nos va a servir para no sólo volver a un estado anterior sino también para “eliminar” todo lo que hemos hecho posteriormente.

Supongamos una situación donde después de programar hasta el cuarto commit decidimos que lo que hemos hecho en los dos últimos commits está mal y queremos volver al estado que teníamos en el segundo commit:

Imagen que contiene texto, firmar, calle, grande

Descripción generada automáticamente

Para hacer esto debemos usar el comando:

git reset --hard hashÚnico

Imagen que contiene firmar, calle, cerca, ciudad

Descripción generada automáticamente

Es decir, ahora el HEAD del proyecto se encuentra a la altura del segundo commit y, por ejemplo, ya no tenemos el archivo .gitignore en nuestro proyecto:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Ahora supongamos la situación donde después de hacer esto nos damos cuenta de que nos hemos equivocado realizando esta vuelta a un estado anterior porque al final nuestro desarrollo no era tan malo como preveíamos. Podríamos pensar en hacer lo mismo que hicimos en el apartado anterior de hacer un checkout al log final pero si hacemos un git log o un git tree veremos lo siguiente:

Texto

Descripción generada automáticamente

No tenemos, aparentemente, posibilidad de ver ese commit final que nos permitiera volver al estado donde teníamos 4 commits y en esta situación se puede pensar que lo hemos perdido todo y no queda otra que volver a realizar el desarrollo, pero en Git realmente todo aquello que ha tenido un commit queda registrado de una u otra manera.

En este caso debemos usar el comando del log ampliado:

git reflog

Texto

Descripción generada automáticamente

Con este comando sí que podemos ver cual es ese commit final. Sabemos que es el que tiene el hash 9ba6c52 porque es el que está asociado al commit “Se añade el .gitignore.

Podríamos pensar entonces que con hacer un git checkout hacia ese hash nos podría servir:

Texto

Descripción generada automáticamente

La realidad es que si hacemos un git tree podremos ver que, aunque el HEAD si se encuentra en el último commit, el final de la rama main no lo está:

Texto

Descripción generada automáticamente

Llegados a este caso se podría deducir que haciendo un git checkout sobre la rama se volverían a unir al final, pero en realidad lo que haremos es llevar el HEAD junto al main:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Haciendo un git tree lo podremos ver:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

La solución real a esto es volver a hacer un reset --hard pero sobre el commit final:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente con confianza media

De esta manera si accedemos al log o al tree veremos que todo ha vuelto a la normalidad:

Texto

Descripción generada automáticamente

Imagen que contiene texto, firmar, calle, ciudad

Descripción generada automáticamente

Por tanto, git reset --hard no solo nos va a servir para volver atrás borrando lo que hayamos hecho sino también para volver adelante y rehacer aquellos cambios que creíamos perdidos.

# GENERACIÓN DE ETIQUETAS (git tag)

Vamos a crear tags que nos sirvan para etiquetar commits. Esto lo vamos a realizar cuando tengamos un commit importante que nos sirva para identificar rápidamente un commit.

Estos tags también nos van a servir para ver visualmente dónde hemos creado ciertas cosas como por ejemplo las versiones de nuestros programas, de hecho, las etiquetas por definición a nivel de producción son las versiones de forma que cada vez que lanzamos una versión utilizable de nuestra aplicación, esta lleva aparejada una etiqueta.

Para crear un tag vamos a utilizar el siguiente comando

git tag nombre\_del\_tag[[3]](#footnote-4)



Si ahora hacemos un log veremos lo siguiente:

Texto

Descripción generada automáticamente

Junto al último commit ha aparecido un tag con el nombre primeros\_pasos. De esta forma, este punto del desarrollo tiene un nombre identificativo.

Si queremos ver todo el listado de tags que tenemos en nuestro proyecto usaremos el comando git tag sin ningún nombre:

git tag

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

Para ver para qué nos sirven realmente los tags, vamos a crear un nuevo fichero en el proyecto:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Vamos a generar un nuevo commit, para ello al igual hicimos anteriormente vamos a hacer una comprobación del estado del proyecto:

Texto, Sitio web

Descripción generada automáticamente

Como era de esperar nos informa de que hay un nuevo fichero que no ha tenido nunca seguimiento y que debemos añadirlo usando el comando git add.

Vamos a hacerlo pero en este caso con un pequeño cambio, en lugar de hacer un git add nombreFichero, vamos a hacer un git add ., la diferencia es que con este comando haremos un paso a stage de todos los ficheros nuevos y/o que hayan sufrido modificaciones desde el último commit. De esta manera si, por ejemplo, tras una sesión de programación tenemos 20 archivos nuevos y/o modificados, no tendremos que hacer 20 git add sino uno único.

git add .



Si ahora hacemos un git status veremos que el archivo ya se ha añadido al stage:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente con confianza media

Una vez hecho esto hacemos un commit:

Texto

Descripción generada automáticamente

Si ahora vemos un git tree veremos que tenemos un quinto commit en nuestra rama main:

Texto

Descripción generada automáticamente

Como podemos ver el HEAD y el main se han movido a este quinto commit, pero el tag de primeros\_pasos se mantiene sobre el mismo cuarto commit.

Supongamos que nos queremos mover a ese cuarto commit, ahora tenemos una forma más sencilla de hacerlo porque no requerimos de ese hash único del commit, sino que podemos hacerlo con el nombre identificativo del commit, su tag asociada primeros\_pasos:

git checkout tags/primeros\_pasos

Texto

Descripción generada automáticamente

Si hacemos un git tree veremos que el HEAD ahora se encuentra a la altura de primeros\_pasos:

Texto

Descripción generada automáticamente

Por tanto, los tags nos van a servir para simplificar el cómo nos movemos por una rama ya que lo haremos directamente a través de un nombre identificativo y mucho más sencillo de recordar.

Obviamente al hacer esto hemos perdido el archivo hellogit3.py:

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

Para terminar esta sección vamos a volver al final de la rama con un checkout:

Texto

Descripción generada automáticamente

# CREACIÓN DE NUEVAS RAMAS (git branch) Y MOVIMIENTO ENTRE RAMAS (git switch)

Si volvemos a ver una representación gráfica de nuestro proyecto en este punto sería algo así:

Diagrama

Descripción generada automáticamente

Tenemos un total de 5 commits y al cuarto de ellos le hemos colocado un tag con nombre primeros\_pasos, además la situación actualmente del HEAD y el main es que se encuentran en el último commit (el punto donde nos encontremos lo identificaremos siempre en los gráficos como el círculo que tenga el color rojo).

Hasta ahora hemos trabajado con una única rama y si estamos trabajando solos puede ser una manera correcta de realizar un desarrollo, pero supongamos que estamos en un equipo de trabajo de dos personas y cada una se debe encargar de cierta parte del desarrollo, si bien sería posible trabajar en una única rama, hacerlo sería muy tedioso y poco productivo, en este caso debemos crear una segunda rama para esta persona.

Antes de realizar comandos vamos a ver en representación gráfica lo que vamos a realizar, supongamos que esta segunda persona va a encargarse de generar una nueva funcionalidad, por ejemplo, un login para la aplicación que realmente se separe ce la rama main.

Gráfico

Descripción generada automáticamente

Para hacer esto en comando debemos usar el comando:

git branch nombreNuevaRama



Si ahora hacemos un git tree:

Texto

Descripción generada automáticamente

Vemos que nuestro HEAD sigue enlazado al main pero tenemos una nueva rama llamada login.

Para cambiarnos de rama vamos a utilizar otro comando:

git switch nombreRama

Texto

Descripción generada automáticamente

Si ahora volvemos a ver nuestro git tree veremos que el HEAD ahora se encuentra enlazado a la rama login:

Texto

Descripción generada automáticamente

Es decir, ahora nuestra situación es esta:

Diagrama

Descripción generada automáticamente

Hemos generado una rama, que contiene todo lo que contenía el proyecto hasta el momento, y con ello hemos creado un nuevo flujo de trabajo.

Vamos a crear un nuevo fichero en nuestro directorio de trabajo:

Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media

Texto

Descripción generada automáticamente

Si ahora hacemos un git status de la situación del proyecto:

Texto

Descripción generada automáticamente

Vamos a hacer el commit:

Imagen que contiene texto, firmar, calle, diferente

Descripción generada automáticamente

Nuestra situación actual por tanto es la siguiente:

Diagrama

Descripción generada automáticamente

Si lo vemos en log sería esto:

Texto

Descripción generada automáticamente

Ahora vamos a cambiar otra vez de rama y volver a la rama main:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Si ahora vamos al directorio de trabajo veremos como el archivo login.py no existe:

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

Esto se debe a que nuestro HEAD ahora se encuentra al final de la rama main:

Imagen que contiene Gráfico

Descripción generada automáticamente

Por tanto, como podemos ver tenemos 2 flujos completamente diferenciados.

Vamos a realizar ahora un cambio sobre el archivo hellogit3.py:

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

Y hacemos el correspondiente commit:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

La situación por tanto es esta:

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza baja

# FUSIÓN DE RAMAS (git merge)

Ahora volvamos a la rama login:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Supongamos que han pasado días y queremos comprobar si lo que se ha seguido realizando en la rama main es compatible todavía con lo que tenemos nosotros en nuestra rama login. Es decir, queremos tener los datos actualizados que tenga la rama main.

Para esto vamos a usar el “merge”, es decir, vamos a combinar los cambios:

git merge ramaAFusionar

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Si ahora vamos a nuestro proyecto veremos que en el archivo helloGit3.py tenemos los cambios que se realizaron en la rama main posteriormente a la división de ramas:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Es decir, nos hemos traído los datos de la rama main a la rama login.

En un git tree podemos hacernos una idea más gráfica de lo que ha ocurrido:

Texto

Descripción generada automáticamente

Y si vemos el log veremos que se ha generado un commit automático al hacer el merge:

Texto

Descripción generada automáticamente

En la representación gráfica que estamos haciendo podríamos decir que ha ocurrido esto:

Gráfico, Diagrama, Gráfico de burbujas

Descripción generada automáticamente

## CONFLICTOS EN LA FUSIÓN DE RAMAS

El ejemplo que hemos hecho para explicar un merge que funciona sin problema ya que no hay conflictos entre ramas, pero podemos tener una situación donde exista un conflicto entre ambas ramas, por ejemplo, porque dos programadores hayan tocado el mismo archivo realizando dos codificaciones diferentes.

Estando en la rama login vamos a modificar el archivo hellogit3.py:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Y hacemos un commit:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Es decir, estamos simulando una situación donde el equipo de login ha modificado un archivo que, en principio, es exclusivo de la rama main y que no debería ser modificado.

Ahora volvemos a la rama main:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Y si vamos al proyecto veremos que no tenemos el archigo login.py y que en el archivo hellogit3.py tenemos el texto original y no el modificado en login:

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamenteInterfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Esto ocurre porque cuando hicimos el “merge” fue en el login trayéndonos los cambios de main pero main no ha recibido ningún “merge” por lo que no puede tener los cambios que se han realizado en login.

Modificamos el archivo hellogit3.py:

Interfaz de usuario gráfica, Texto, Chat o mensaje de texto

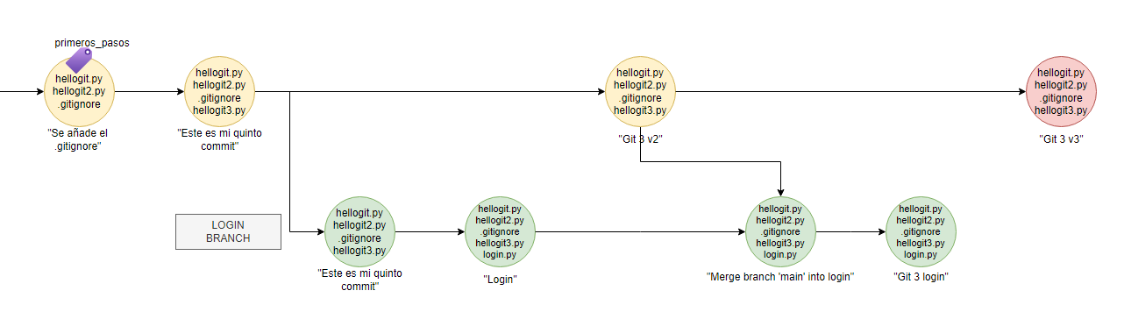
Descripción generada automáticamente

Y hacemos un commit en esta rama:

Texto, Aplicación

Descripción generada automáticamente

Ahora mismo la situación es esta:



Tenemos en la rama main un commit con unos cambios sobre el archivo hellogit3.py y otro commit en la rama login con cambios sobre este mismo archivo.

Vamos a volver a la rama login:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Y volvemos a hacer un git merge main para traernos los cambios de main:

Texto

Descripción generada automáticamente

Pero en este caso, al contrario que antes, surge un conflicto sobre el contenido del archivo hellogit3.py por lo que el merge automático falla y no se realiza.

Esto se debe a que ambas ramas han tocado el mismo fichero y la misma línea (en el caso de tocar el mismo fichero, pero diferentes líneas no habría problema y el merge seguiría siendo automático ya que no existiría conflicto alguno) por lo que git no sabe si debe quedarse con los cambios realizados por una rama o con los cambios realizados por la otra rama.

A nivel gráfico lo que ha ocurrido es esto:

Diagrama, Esquemático

Descripción generada automáticamente

Si ahora nos vamos al proyecto y, en concreto, al fichero hellogit3.py, veremos que el editor nos dice lo siguiente:

Patrón de fondo

Descripción generada automáticamente con confianza media

Nos colorea de verde el cambio realizado por login (la rama que ha solicitado el merge) y de azul el cambio realizado por main.

Suponemos que el equipo de login decide no conservar los cambios realizados por ellos y borra las líneas que causan conflicto, quedando el archivo así:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Esto no quiere decir que hayamos tocado el fichero que se encuentra en main (desde una rama no podemos afectar a los ficheros de otra rama de forma directa) sino que modificamos nuestro archivo hellogit3.py de login para que tenga los mismos cambios que tiene el archivo hellogi3.py de main.

Para finalizar el proceso de forma manual no podemos hacer un commit directamente ya que ocurriría lo siguiente:

Texto

Descripción generada automáticamente

Si vemos el status veremos que existe un cambio en un fichero así que demos añadirlo primero al stage y posteriormente hacer el commit:

Texto

Descripción generada automáticamente

Con esto hemos conseguido completar el merge de forma manual.

# ALMACENAR TEMPORALMENTE CAMBIOS (git stash)

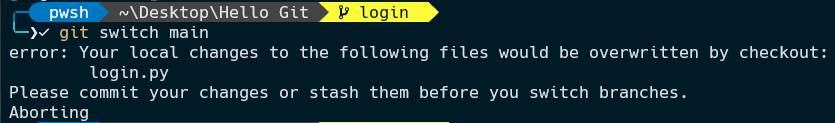
Supongamos que el equipo de login ha seguido trabajando sobre el archivo de login y ha realizado los siguientes cambios:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

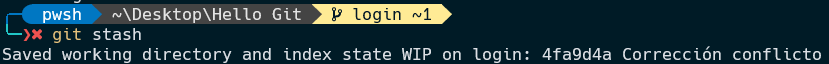
Y durante este proceso de cambios el equipo de main requiere de la ayuda del equipo de login para trabajar en determinada función de main.

Si ahora nos intentamos cambiar a la rama main ocurriría lo siguiente:



Git impide que nos podamos mover de rama porque tenemos cambios sin guardar y si nos cambiamos de rama los archivos que han sufrido alguna modificación desde el último commit se sobrescribirían. Para solucionar esta situación tenemos dos opciones: hacer un commit con los cambios realizados y guardar temporalmente los cambios, en este caso vamos a hacer lo segundo y para ello utilizamos el comando:

git stash



Esto lo que ha hecho es una especie de commit temporal que sólo afecta de forma local y no al árbol, no es un commit como tal es un almacenado de los cambios no guardados para poder seguir trabajando más tarde.

Podemos ver el estado de los stash con el comando:

git stash list

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Este listado de stash es visible desde cualquier rama por lo que desde la rama main también podríamos ver esta entrada del listado.

Y ahora podremos cambiar de rama sin problemas:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Terminamos de trabajar en esa funcionalidad que requería de nuestra ayuda y volvemos a la rama login:

Texto

Descripción generada automáticamente

Si ahora nos vamos al proyecto veremos lo siguiente:

Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media

En el archivo login.py tenemos el estado en el que se encontraba tras el último commit y no en el estado en el que se encontraba cuando hicimos stash.

Para recuperar los cambios temporales debemos usar el comando:

git stash pop

Texto

Descripción generada automáticamente

Ahora en el fichero veremos que sí que tenemos los cambios realizados y almacenados en stash:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Vamos a hacer commit de estos cambios:

Texto, Aplicación

Descripción generada automáticamente

Seguimos haciendo cambios en nuestro fichero de login:

Pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza media

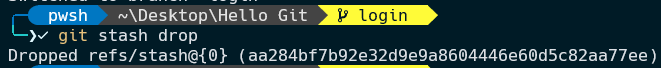
Y nos vuelven a pedir ayuda desde main por lo que tenemos que guardar los cambios temporalmente y volver a la rama main:

Escala de tiempo

Descripción generada automáticamente con confianza media

Supongamos que al volver de main a login decidimos que esos cambios que teníamos guardados temporalmente ya no los queremos y queremos eliminarlos, para ello usamos el comando:

git stash drop



Y ahora en el archivo login.py tendremos el estado que había tras el último commit:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

# REINTEGRACIÓN DE RAMAS[[4]](#footnote-5)

Hemos llegado al punto del desarrollo donde hemos acabado el login así que avisamos al equipo de main para que sepan que hemos terminado el desarrollo de la funcionalidad.

Vamos a cambiarnos a la rama main:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Y vamos a traernos los cambios de la rama login a main para poder integrar el trabajo del equipo.

Lo primero que vamos a hacer es comprobar si existen posibles conflictos para ello utilizaremos el comando git diff pero con un nuevo parámetro:

git diff nombreRama

Texto

Descripción generada automáticamente

Hay ciertas diferencias entre ramas pero no generan conflictos por lo que podemos hacer un merge sin problemas:

Texto

Descripción generada automáticamente

Y ahora en la rama main tendremos el archivo login.py:

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

Ahora podemos hacer cambios en este archivo de login para darle retoques finales desde la rama main:

Un reloj digital

Descripción generada automáticamente con confianza baja

Y le podemos hacer commit:

Texto

Descripción generada automáticamente

# ELIMINACIÓN DE RAMAS EN GIT

Hemos llegado al punto donde login es una rama de trabajo que ya no es necesaria porque el equipo que estaba asociada a ella ha terminado su trabajo y tenemos que eliminar su rama.

Siempre debemos eliminar las ramas que hayan terminado su función. Dejar ramas sin eliminar puede darnos problemas en proyectos más grandes donde veamos demasiadas ramas que ya no tienen ninguna función real y que dificulten la lectura del flujo.

Para eliminar ramas debemos utilizar el comando:

git branch -d nombreRama

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Esta rama que hemos eliminado si bien ya no va a aparecer en el listado de ramas:

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

Si que va a seguir referenciada en el reflog:

Un conjunto de letras blancas en un fondo verde

Descripción generada automáticamente con confianza baja

Vamos a movernos por ejemplo al commit cuando hicimos el primer merge:

Texto

Descripción generada automáticamente

Ahora en un git tree veremos que la rama existe:

Texto

Descripción generada automáticamente

Esto ocurre porque, como hemos visto en otros apartados, todo en Git permanece, aunque sea eliminado para que pueda ser rescatado siempre que queramos.

Vamos a volver al final del main para terminar con esta sección:

Interfaz de usuario gráfica, Texto, Sitio web

Descripción generada automáticamente

# GITHUB

## ¿QUÉ ES GITHUB?

Logotipo

Descripción generada automáticamenteGitHub se ha convertido en mucho más que una plataforma para alojar repositorios de código. Es un ecosistema completo y colaborativo para desarrolladores, permitiendo el almacenamiento, seguimiento de cambios, colaboración y gestión de proyectos de software de manera eficiente.

Esta plataforma basada en la nube se apoya en el sistema de control de versiones Git, lo que significa que hereda las potentes capacidades de control de versiones de Git y las complementa con una serie de herramientas y funcionalidades adicionales.

Uno de los aspectos más destacados de GitHub es su interfaz amigable y su facilidad de uso, lo que la convierte en un espacio accesible para desarrolladores de todos los niveles de experiencia. Desde el control detallado de versiones hasta la gestión de problemas, la revisión de código, la colaboración en equipo y la automatización de flujos de trabajo, GitHub proporciona un entorno completo para el ciclo de vida del desarrollo de software.

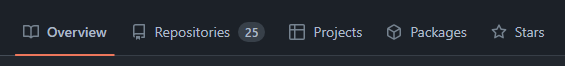
La función de “solicitudes de incorporación de cambios” *(pull requests*) es fundamental en GitHub. Permite a los desarrolladores proponer cambios en un repositorio y facilita el proceso de revisión y fusión de esos cambios de manera organizada y controlada.

Además, GitHub ofrece una gama de características adicionales, como la integración continua (CI), que permite automatizar pruebas y despliegues, y la posibilidad de crear páginas web estáticas a partir de repositorios, lo que lo convierte en un lugar versátil para alojar proyectos de diversa índole.

En los siguientes apartados vamos a ver diversas características y funcionalidades que ofrece GitHub, explorando cómo utilizar esta plataforma de manera efectiva para gestionar proyectos, colaborar con equipos distribuidos en todo el mundo y optimizar el desarrollo de software en entornos colaborativos.

## PRIMEROS PASOS EN GITHUB. CONOCIENDO LA INTERFAZ

En nuestro home vamos a tener los siguientes apartados:



* **Overview**: La vista general en GitHub ofrece un resumen detallado de un repositorio, mostrando estadísticas clave, actividades recientes y contribuciones. Es un punto central para comprender la salud y la actividad de un proyecto.
* **Repositories**: Los repositorios en GitHub son contenedores donde se almacena y gestiona el código fuente de un proyecto. Permiten controlar versiones, colaborar, mantener un historial de cambios y gestionar problemas.
* **Projects**: Los proyectos en GitHub son tableros para organizar y gestionar tareas, problemas o contribuciones. Ofrecen un flujo de trabajo visual para planificar, priorizar y monitorear el progreso del trabajo en un proyecto.
* **Packages**: GitHub Packages es un servicio para publicar y gestionar paquetes de software dentro de la plataforma. Permite almacenar y compartir librerías, contenedores, módulos y otros recursos de desarrollo.
* **Stars**: La función “stars” permite a los usuarios marcar o “star” repositorios que les interesan o encuentran útiles. Sirve como un sistema de marcadores para seguir proyectos, mostrar aprecio o indicar interés en un código en particular.

## CREANDO UN REPOSITORIO

Dentro de la sección Repositories veremos la siguiente barra de búsqueda y un botón “New” que nos servirá para crear un repositorio nuevo:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Cuando hagamos clic en dicho botón nos aparecerá la posibilidad de crear un nuevo repositorio:

Texto

Descripción generada automáticamente

En este formulario tenemos varias opciones. En la parte superior tenemos un primer formulario que nos solicita el nombre que tendrá el repositorio y la descripción del mismo:

Interfaz de usuario gráfica, Texto, Sitio web

Descripción generada automáticamente

Existe un nombre especial de repositorio que es el del propietario de la cuenta. Este repositorio sirve para formatear el perfil principal de GitHub usando un archivo MarkDown (.md):

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

En este caso utilizaremos un repositorio con el nombre HelloGit.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

En la siguiente parte del formulario podremos optar por un repositorio público (accesible por todo el mundo) o privado (accesible sólo por nosotros y por personas a las que le demos permiso):

Texto

Descripción generada automáticamente

También podemos elegir si queremos crear un archivo README o no, este archivo nos va a servir para hacer un resumen de lo que hace la aplicación que hemos subido al repositorio:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Además podemos decidir si queremos añadir o no el archivo .gitignore:

Texto

Descripción generada automáticamente

Por último, podemos elegir la licencia que tendrá asociada nuestro código:

Captura de pantalla de un celular

Descripción generada automáticamente

Terminamos pulsando en el botón “Create repository” y ya habremos creado nuestro repositorio:

Captura de pantalla de un celular

Descripción generada automáticamente

## LOCAL Y REMOTO

Hasta ahora en esta guía hemos trabajado a nivel local con nuestro proyecto, lo que nos va a permitir GitHub es poder seguir trabajando de esa manera a la vez que tenemos un soporte en remoto en el que se guarda el proyecto y sobre el que se hacen todos los cambios.

Esto permite a su vez que no solo nosotros podamos hacer cambios en el proyecto, sino que la integración de terceras personas en el proyecto sea mucho más sencilla. De esta forma los usuarios pueden subir sus cambios al proyecto de GitHub y recuperar los cambios realizados por otros:

Diagrama

Descripción generada automáticamente

# ENLAZAR UN REPOSITORIO LOCAL CON UN REPOSITORIO REMOTO (git remote)

Ahora que hemos avanzado en nuestro repositorio local y hemos creado un repositorio remoto es el momento de enlazar ese repositorio local con el repositorio remoto, para ello vamos a utilizar el siguiente comando:

git remote add origin repositorioRemoto

Este comando lo podemos encontrar en nuestro repositorio remoto[[5]](#footnote-6):

Texto

Descripción generada automáticamente

Vamos a lanzar este comando:



Lo que hemos hecho con este comando es enlazar el repositorio local con el repositorio remoto.

# SUBIR EL REPOSITORIO LOCAL AL REPOSITORIO REMOTO (git push)

Una vez que hemos enlazado ambos repositorios el siguiente paso es subir el contenido de nuestro repositorio local al repositorio remoto, para ello usaremos el siguiente comando:

git push origin ramaRemota

Texto

Descripción generada automáticamente

Si ahora vamos al repositorio remoto veremos que tenemos el mismo contenido que en el repositorio local:

Captura de pantalla de un celular

Descripción generada automáticamente

# SUBIDA DE CAMBIOS A GITHUB (git push)

Vamos a suponer que una nueva persona empieza a trabajar en el repositorio y va a encargarse de crear el README. Pulsamos en el botón “Add a README”

Interfaz de usuario gráfica, Aplicación, Teams

Descripción generada automáticamente

Hacemos clic en “Commit changes…” y se nos desplegará un modal:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

En él se nos solicita el nombre que tendrá el commit y si queremos hacer el commit directamente sobre la rama main o crear una nueva rama para este commit e iniciar con ello una pull request. Por ahora vamos a hacer un commit sobre la rama main.

Esto habrá cambiado algunas cosas de nuestro repositorio:

Captura de pantalla de un celular

Descripción generada automáticamente

Tenemos un archivo README.md entre nuestros ficheros del proyecto, además tenemos una página de bienvenida abajo y si hacemos clic en “History” veremos que tenemos un nuevo commit en nuestro proyecto:

Captura de pantalla de un celular

Descripción generada automáticamente

Volvamos ahora a nuestro repositorio local, en él, por ejemplo, no tenemos el archivo README.md:

Texto

Descripción generada automáticamente

Vamos a hacer algunos cambios, por ejemplo, sobre el primer hellogit:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Una vez que hemos hecho esto vamos a hacer un commit de nuestro proyecto:

Texto

Descripción generada automáticamente

Si ahora vamos al archivo hellogit.py del repositorio de GitHub veremos que no se ha producido ningún cambio:

Captura de pantalla de un celular

Descripción generada automáticamente

Esto se debe a que el commit se ha hecho en local y por tanto sólo existe en este ámbito, para que estos cambios se puedan ver en remoto hay que subir el commit para ello usamos el siguiente comando:

git push

Texto

Descripción generada automáticamente

Nuestro intento de subida ha sido rechazado porque existen cambios en remoto que no han sido traídos al local (el archivo README.md que generamos al inicio de este apartado).

# COMPROBACIÓN DE EXISTENCIA DE CAMBIOS EN REMOTO (git fetch) Y ACTUALIZACIÓN DEL REPOSITORIO LOCAL CON LOS CAMBIOS REMOTOS (git pull)

Lo primero que vamos a ver en este apartado es cómo comprobar si existen cambios en remoto que sea necesario traerlos a local para poder trabajar:

git fetch

Texto

Descripción generada automáticamente

Si ahora hacemos un tree veremos esto:

Texto

Descripción generada automáticamente

Lo que hace git fetch es descargarse los metadatos de los cambios que se hayan producido en remoto, es decir, información sobre la existencia de nuevos commits, pero no realiza ninguna descarga de los ficheros.

Para descargar estos cambios utilizamos el comando:

git pull

Imagen que contiene firmar, botella, foto, calle

Descripción generada automáticamente

Esto ha hecho que se descarguen los ficheros con cambios que teníamos en remoto:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Pero, sin embargo, nuestro archivo hellogit.py sí está como lo hemos dejado en local, no se ha visto afectado por el pull:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Esto se debe a que la fotografía que utilizamos es la local y no la remota. Ahora bien, en caso de que el archivo hellogit.py hubiera sufrido cambios en el remoto en ese caso sí que veríamos añadidos los cambios remotos, es decir, pull no sobrescribe los ficheros locales con los remotos, sino que descarga los cambios que existan. En el fondo hacer un pull es hacer un merge con la rama remota.

Obviamente si existiera conflicto entre archivos (como vimos en el caso del merge) Git nos avisaría y tendríamos que solucionar ese conflicto de forma manual.

Ya nos hemos sincronizado en local y si hacemos un git status veremos lo siguiente:

Texto

Descripción generada automáticamente

Git nos informa de que aparentemente todo está correcto, pero nos falta por publicar 2 commits estos commits son estos:

Texto

Descripción generada automáticamente

El primero de ellos corresponde al merge que realizamos entre el local y el remoto y el segundo al cambio que hicimos sobre el fichero hellogit.py. Para subir estos commits (y los cambios aparejados) debemos hacer un git push de nuevo:

Texto

Descripción generada automáticamente

En este caso el push ha sido correcto y hemos podido subir nuestros cambios al repositorio remoto. De hecho, para comprobar esto podemos ver el estado en GitHub del fichero hellogit.py:

Captura de pantalla de un teléfono celular

Descripción generada automáticamente

# DESCARGA DE UN REPOSITORIO REMOTO (git clone)

Supongamos que somos un nuevo trabajador del equipo y que queremos descargarnos el proyecto en local para poder trabajar.

Para descargar el código tenemos varias opciones, por ejemplo, podemos descargarlo a mano haciendo clic en Code 🡪 Dowload ZIP:

Captura de pantalla de un celular

Descripción generada automáticamente

Pero en este caso vamos a hacer una clonación utilizando la URL HTTPS, para ello vamos a crearnos un directorio:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Nos movemos a este directorio y utilizamos el comando:

git clone URLRepositorioRemoto

Imagen que contiene firmar, hombre, parado

Descripción generada automáticamente

Si ahora comprobamos el contenido del directorio veremos que tenemos un directorio con el nombre del repositorio remoto:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Y dentro de este repositorio tendremos todos los archivos del proyecto:

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

# PROTEGIENDO NUESTRO PROYECTO FRENTE A CAMBIOS

Estamos trabajando con un repositorio remoto público y eso, obviamente, implica que cualquier persona puede ver el repositorio y puede descargárselo. El problema que tenemos ahora mismo es que cualquier persona puede hacer cambios sobre nuestro repositorio porque no hemos generado ninguna regla que lo impida.

Para realizar esta protección frente a cambios tenemos varias opciones, establecer unas reglas sobre una determinada rama, establecer un conjunto de reglas para varias o todas las ramas de nuestro proyecto (*ruleset*) o el uso de la autenticación vía SSH para realizar cambios. En este caso como estamos trabajando con una única rama y hemos comenzado todo como autenticación HTTPS vamos a utilizar la primera de las formas.

Para proteger nuestra rama principal debemos ir al apartado de configuración de nuestro repositorio:



Y en el menú lateral elegir la opción “branches” de la sección “code and automation”:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Una vez aquí veremos la siguiente sección de la web:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Hacemos clic en añadir una nueva regla de protección y se nos desplegará un formulario donde nos solicita el nombre (o el patrón) que tiene la rama a la que queremos afectar y las reglas que vamos a aplicar en este caso vamos a elegir las opciones:

1. *Require a pull request before merging*: Más adelante vamos a ver el concepto de *pull request* y cómo realizarlas. Estas *pull requests* son commits que se realizan a ramas no protegidas y que no se pueden adjuntar (*merge*) a la rama protegida a la que afectan pero a través de una solicitud y no de forma directa.
2. *Requiere approvals*: Relacionado con lo anterior las PR necesitarán de una aprobación para que se puedan unir a la rama afectada.
3. *Do not allow bypassing the above settings*: Con ello conseguiremos que las reglas afecten a todos los roles del proyecto incluido el administrador de la rama.

Hacemos clic en crear y con ello ya tendremos nuestra rama main con protecciones:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Ahora nuestro nuevo miembro del equipo va a hacer cambios en su proyecto local (el que hemos clonado en un apartado anterior) y genera commits:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Si ahora intenta hacer un push al repositorio remoto ocurrirá lo siguiente:

Texto

Descripción generada automáticamente

El repositorio remoto ha impedido que se hagan cambios directos sobre la rama a la que se está intentando acceder y nos informa de que estos cambios se deben hacer a través de una *pull request.*

# CONCEPTO DE FORK EN GITHUB

Antes de entrar de lleno con el concepto de *pull request* vamos a ver el concepto de *fork* (bifurcación) en GitHub.

Para realizar esto vamos a trabajar con una cuenta secundaria en GitHub que es diferente a la original y a la que estamos usando hasta ahora.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

Desde esta cuenta podemos acceder al repositorio que hemos creado en la cuenta principal:

Captura de pantalla de un celular

Descripción generada automáticamente

Y una de las opciones que nos aparece arriba a la derecha es *fork*:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

El hacer *fork* consiste en hacer una copia del repositorio ajeno y llevárnoslo a nuestra lista de repositorios de forma que podamos realizar todos los cambios que queramos sin afectar al repositorio de origen. Cuando hacemos click en Fork nos aparece el siguiente formulario:

Captura de pantalla de un celular

Descripción generada automáticamente

Generamos el fork en nuestro repositorio y ya podemos trabajar con él sin problemas:

Captura de pantalla de un celular

Descripción generada automáticamente

Ahora el siguiente paso sería hacer un clone de este repositorio y trabajar sobre él. Por tanto, vamos a irnos al directorio padre donde tenemos todas las clonaciones y clonamos este repositorio utilizando la URL asociada:

Texto

Descripción generada automáticamente

De esta forma tenemos el repositorio del fork:

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

# FLUJO COLABORATIVO EN GITHUB

Vamos ahora a trabajar sobre el repositorio fork y vamos a crear un nuevo archivo MarkDown (hello.md):

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

Y vamos a hacer cambio sobre él:

Interfaz de usuario gráfica, Texto, Sitio web

Descripción generada automáticamente

Vamos a hacer el correspondiente commit para guardarlo:

Texto, Escala de tiempo

Descripción generada automáticamente

Y ahora vamos a hacer el push (este push se hará sobre el repositorio remoto de la cuenta secundaria y no sobre el de la principal)[[6]](#footnote-7):

Texto

Descripción generada automáticamente

Si ahora vamos a nuestro repositorio remoto veremos que se han almacenado los cambios:

Interfaz de usuario gráfica, Texto, Aplicación, Sitio web

Descripción generada automáticamente

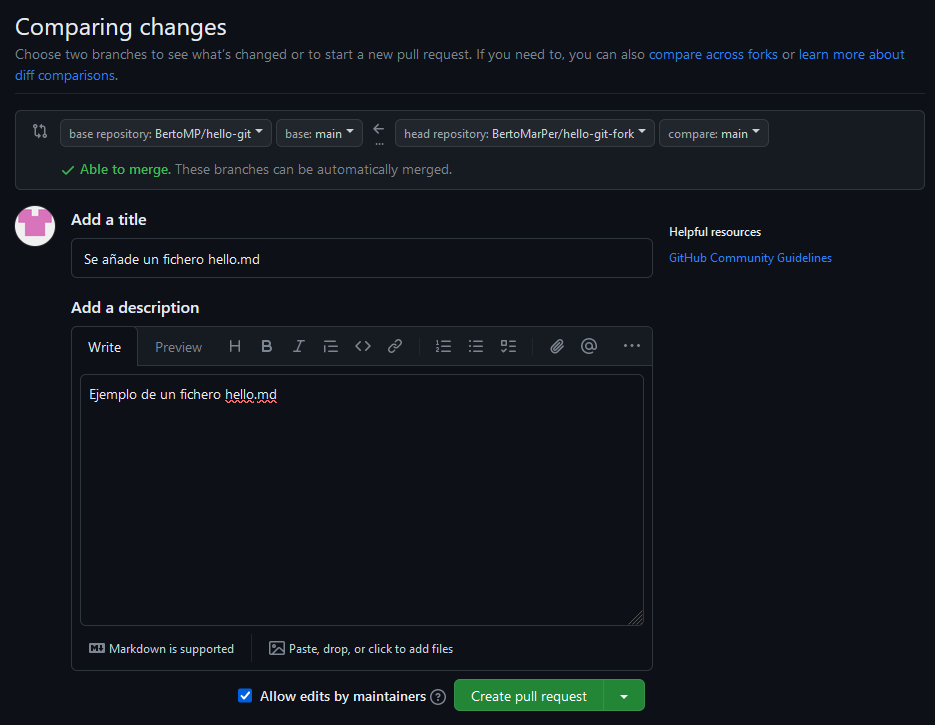
# PULL REQUEST EN GITHUB

El siguiente paso que debemos hacer es la *pull request* al repositorio original donde vamos a contribuir con los cambios que hemos realizado en el apartado anterior, para ello desde nuestro repositorio secundario vamos a hacer clic en “contribute” y, a continuación, en “Open pull request”:

Captura de pantalla de un celular

Descripción generada automáticamente

Esto nos abrirá un formulario:



En él podemos añadir un título a la *pull request* y una descripción de la misma. Hacemos clic en “Create pull request” y veremos lo siguiente:

Captura de pantalla de un celular

Descripción generada automáticamente

Con esto hemos creado una *pull request* pero esta, por las normas que establecimos en la protección del repositorio, debe ser revisada por el administrador del repositorio original.

Ahora en el repositorio del administrador aparecerá esto:

Interfaz de usuario gráfica, Aplicación, Sitio web

Descripción generada automáticamente

Ha recibido una *pull request* sobre el proyecto:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Si entramos en ella podremos ver los cambios que se quieren realizar sobre nuestro repositorio:

Captura de pantalla de computadora

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Captura de pantalla de un celular

Descripción generada automáticamente

Podemos hacer clic en “review changes” para dar feedback sobre lo que nos parecen los cambios que quiere realizar el colaborador:

Captura de pantalla de un celular

Descripción generada automáticamente

Podemos hacer un comentario (*comment*) para dar feedback simplemente, aprobarlo (*approve*) para añadirlo a nuestro repositorio o solicitar cambios (*request changes*) para decirle al colaborador que si cambia ciertas cosas lo podríamos terminar aceptado. En este caso lo vamos a aprobar:

Captura de pantalla de un celular

Descripción generada automáticamente

Ahora en la sección de conversación nos aparecerá esto, una primera parte con los comentarios que se han hecho:

Captura de pantalla de un celular

Descripción generada automáticamente

Y una segunda parte donde nos aparece la opción de hacer *merge* de la *pull request*:

Captura de pantalla de un celular

Descripción generada automáticamente

Hacemos clic en “Merge pull request” y con esto actualizaremos nuestro repositorio remoto principal con los cambios realizados por el colaborador:

Interfaz de usuario gráfica, Sitio web

Descripción generada automáticamente

Si ahora vamos a nuestra sección “code” veremos que tenemos el archivo hello.md que creó el colaborador:

Captura de pantalla de un celular

Descripción generada automáticamente

Por último, apuntar que al igual que vimos en los merge locales en el caso de que existan conflictos entre diferentes *pull requests* o entre una de estas y nuestro repositorio remoto, deberemos hacer un estudio de estos conflictos y decidir qué cambios son con los que nos quedaremos al final.

# SINCRONIZACIÓN DE UN FORK EN GITHUB

Siguiendo con lo que sería el flujo de trabajo normal si utilizamos GitHub podemos tener la situación donde tras un tiempo la diferencia entre nuestro repositorio fork y el repositorio original ha cambiado, por ejemplo, vamos a crear un nuevo archivo en nuestro repositorio original, pero para ello primero vamos a traernos los archivos del repositorio remoto:

Texto

Descripción generada automáticamente

Y ahora creamos un nuevo archivo, un hellogit4.py:

Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media

Vamos a hacerle un commit:

Imagen que contiene texto, firmar, calle, diferente

Descripción generada automáticamente

Y lo vamos a subir al remoto[[7]](#footnote-8):

Texto

Descripción generada automáticamente

Repetimos el proceso generando un nuevo fichero al cual vamos a hacer commit y push:

Un reloj digital en la pantalla

Descripción generada automáticamente con confianza baja

Texto

Descripción generada automáticamente

Si ahora nos vamos a nuestro repositorio remoto con la cuenta de propietario veremos que los archivos hellogit4.py y hellogit5.py se han subido:

Captura de pantalla de un celular

Descripción generada automáticamente

Sin embargo, desde la cuenta secundaria seguimos teniendo el repositorio en el estado donde lo dejamos tras hacer nuestra *pull request* para actualizarlo debemos sincronizar el fork con el repositorio original, para ello debemos hacer clic en “Sync fork”:

Captura de pantalla de un celular

Descripción generada automáticamente

Y, a continuación, en “update branch”:

Captura de pantalla de computadora

Descripción generada automáticamente

De esta forma tendremos el repositorio fork actualizado con los cambios del original.

# USO DE HERRAMIENTAS GRÁFICAS PARA GIT Y GITHUB

Durante esta guía hemos visto lo que es utilizar Git y su integración con GitHub a través de la línea de comandos, pero también tenemos disponibles herramientas gráficas (GUI) que hacen esto mismo destinados para aquellos usuarios que prefieren utilizar una GUI para trabajar. Utilizar una u otra opción depende de cada persona lo importante es entender y conocer en todo momento lo que se está haciendo.

Entre las GUI para Git que tenemos en el mercado vamos a destacar las siguientes:

## GITHUB DESKTOP

Desarrollado por GitHub, es una interfaz gráfica intuitiva y fácil de usar para Git.

Ofrece una forma sencilla de clonar repositorios, gestionar ramas, realizar commits, fusiones y resolver conflictos.

Proporciona una visualización clara de la historia del repositorio y permite colaborar en proyectos alojados en GitHub.

Captura de pantalla de computadora

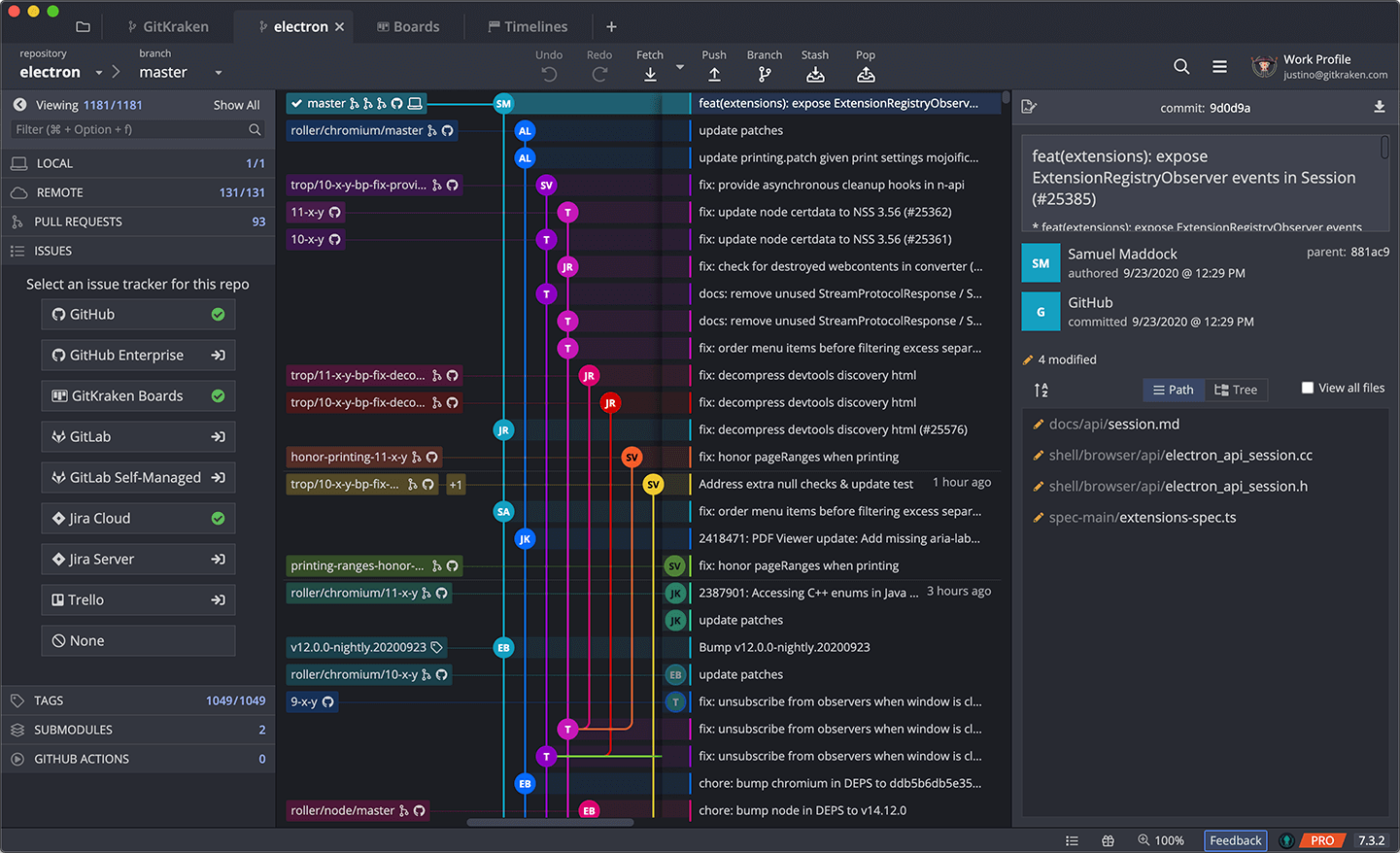
Descripción generada automáticamente

## GITKRAKEN

Es una herramienta multiplataforma con una interfaz visual atractiva.

Permite gestionar repositorios Git, realizar acciones como commits, fusiones, cambios de ramas y resolución de conflictos.

Ofrece integración con servicios de alojamiento como GitHub, GitLab y Bitbucket.

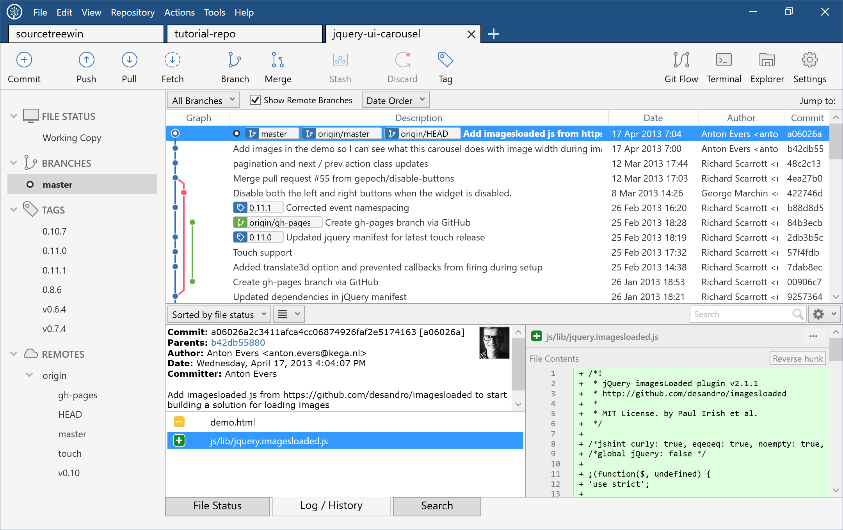


## SOURCETREE

Desarrollado por Atlassian, es una GUI para Git y Mercurial.

Ofrece una interfaz intuitiva para gestionar repositorios, realizar acciones como commits, cambios de ramas, fusiones y resolver conflictos.

Proporciona una visión clara del historial del repositorio y es útil para proyectos alojados en Bitbucket y otros servicios.

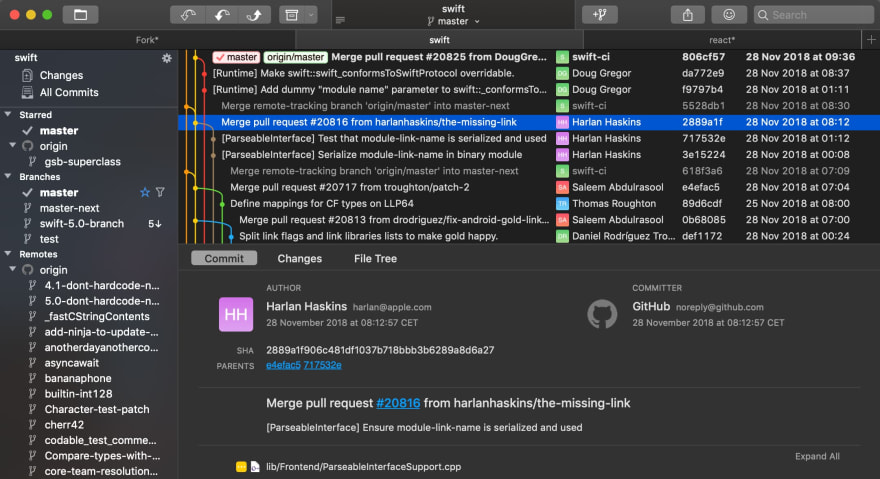


## FORK

Es una interfaz gráfica potente y fácil de usar para Git.

Permite la gestión de ramas, fusiones, resolución de conflictos y proporciona una visión detallada del historial del repositorio.

Ofrece integración con GitHub y está disponible en Windows, macOS y Linux.



El uso de estas aplicaciones depende del usuario y puede ser que nos faciliten la tarea del control de versionado al ver de forma gráfica (como en el git tree que creamos como comando alias) los cambios y las ramas que se han ido generando en nuestra rama.

Aun así, no es recomendable su uso si no se ha entendido Git, si bien nos facilitan el trabajo, como se dijo más arriba lo importante es saber lo que estamos haciendo y que si hacemos clic en un botón “pull changes” sepamos lo que estamos realizando y no hacerlo de forma mecánica.

Por último, hay que mencionar que los principales IDEs actuales como Visual Studio Code, PyCharm, Webstorm, IntelliJ IDEA, etc. cuentan con integraciones de Git para realizar cambios directos bien sobre el repositorio local, bien sobre el repositorio remoto, así como visualizar gráficamente lo que está ocurriendo:

Captura de pantalla de computadora

Descripción generada automáticamente

Pantalla de computadora con fondo negro

Descripción generada automáticamente con confianza media

# GLOSARIO DE COMANDOS - GIT/GITHUB CHEATSHEET

|  |  |
| --- | --- |
| CONFIGURACIÓN DE GIT | |
| git config --global user.name “nombre” | Establece el nombre de usuario asociado a los commits. |
| git config --global user.email “email” | Establece el email asociado a los commits. |
| git config --global init.defaultBranch “nombre” | Establece el nombre que tendrá por defecto la rama principal del proyecto. |
| git config --global alias.Alias “comando” | Establece un alias para un comando en específico. |

|  |  |
| --- | --- |
| MANEJO PRINCIPAL DE UN REPOSITORIO | |
| git init | Crea un nuevo repositorio local. |
| git status | Enumera todos los archivos nuevos o modificados que se deben confirmar (pasar a stage). |
| git add <fichero| .> | Añade un fichero en específico (si se menciona el fichero) o de todos los que han sido creados o modificados (si se usa la opción ‘.’) al estado de stage. |
| git diff | Visualiza qué archivos han sufrido cambios y de qué tipo desde el último commit. |
| git reset fichero | Mueve el archivo del área de stage, pero preserva su contenido. |
| git checkout fichero | Devuelve el fichero a su estado anterior (el que tenía en la última instantánea del proyecto). |
| git commit -m “mensaje” | Genera una instantánea del proyecto y la registra en el historial de versiones del proyecto. |

|  |  |
| --- | --- |
| MODIFICAR Y REHACER COMMITS | |
| git reset “commitHash” | Deshace todos los commits realizados después del commit del parámetro, preservando los cambios de forma local. |
| git reset --hard “commitHash” | Desecha todo el historial de commits posteriores y regresa al commit especificado. |
| git tag “nombreTag” | Genera una etiqueta que identifica al commit asociado. |
| git tag | Enumera todas las etiquetas que han sido generadas en el repositorio actual. |

|  |  |
| --- | --- |
| MANEJO DE LAS RAMAS | |
| git branch nombre | Genera una nueva rama en el repositorio actual. |
| git branch | Enumera todas las ramas del repositorio actual. |
| git checkout nombreRama | Cambia al final de la rama especificada en el comando. |
| git switch nombreRama | Cambia a la rama especificada en el comando. |
| git merge nombreRama | Realizar una fusión de la rama actual con la rama del comando. |
| git branch -d nombreRama | Elimina la rama |

|  |  |
| --- | --- |
| MANEJO DEL HISTORIAL | |
| git log | Enumera el historial de versiones. |
| git reflog | Permite visualizar un historial ampliado. |

|  |  |
| --- | --- |
| CAMBIOS TEMPORALES | |
| git stash | Genera un guardado temporal de los cambios actuales de la rama. |
| git stash list | Listado de stash del proyecto. |
| git stash pop | Recupera los cambios almacenados. |
| git stash drop | Elimina los cambios almacenados. |

|  |  |
| --- | --- |
| USO DE GITHUB | |
| git remote add origin repositorioRemoto | Sincroniza el repositorio local con el repositorio remoto. |
| git push | Intenta actualizar el repositorio remoto con los cambios realizados en local. |
| git fetch | Actualiza los metadatos del repositorio local con la información de los cambios del repositorio remoto. |
| git pull | Descarga desde el repositorio remoto los cambios que no se encuentren en el repositorio local. |
| git clone URLRepositorioRemoto | Genera una clonación del repositorio remoto en un nuevo repositorio local. |

1. Más adelante empezaremos a utilizar el comando “git add .” que va a servir para añadir todos aquellos ficheros que se encuentren sin seguimiento, pero de momento vamos a trabajar con el comando con nombres específicos. [↑](#footnote-ref-2)
2. Para crear este fichero hay que hacer clic en “File” 🡪 “Save workspace as…” y elegir el nombre que tendrá el fichero. [↑](#footnote-ref-3)
3. Por definición y buena práctica los tags deben escribirse en minúsculas y respetando el estilo de escritura *snake\_case* de forma que cada palabra está separada de la siguiente con el carácter de subrayado (\_). [↑](#footnote-ref-4)
4. En realidad, y como veremos más adelante cuando veamos GitHub esta no es la forma real en la que se hacen estas acciones, pero de momento, vamos a hacerlo así ya que se supone que estamos trabajando con un repositorio completamente local. [↑](#footnote-ref-5)
5. En este caso lo vamos a hacer a través de cifrado HTTPS la opción más segura de hacer esto es a través de crear un par de claves pública-privada y usar un cifrado SSH. Visitar la [web oficial](https://docs.github.com/es/authentication/connecting-to-github-with-ssh/about-ssh) para más información sobre cómo hacer este proceso de autenticación. [↑](#footnote-ref-6)
6. A efectos de que haya una mayor comprensión de los cambios y de quién los hace se ha modificado el archivo .gitconfig para que haya un nombre de usuario y correo diferentes a los que tenía el usuario original. [↑](#footnote-ref-7)
7. Para hacer posible este cambio directo a main hemos deshabilitado la regla que impedía hacer commits directos a esta rama. En el día a día los commits siempre se harán a ramas diferentes dejando el main limpio para recibir commits únicamente a través de *pull requests*. [↑](#footnote-ref-8)