



GUÍAS DE USO

GUÍA INICIAL DE GIT Y GITHUB

ÍNDICE

INTRODUCCIÓN	4
A. ¿QUÉ ES GIT?.....	4
B. INSTALACIÓN DE GIT EN NUESTROS EQUIPOS.....	5
C. COMPROBACIÓN DE INSTALACIÓN CORRECTA	6
CREANDO NUESTRO DIRECTORIO DE TRABAJO.....	9
CONFIGURACIÓN DE GIT (<i>git config</i>)	9
INICIALIZANDO NUESTRO REPOSITORIO (<i>git init</i>)	11
CONCEPTO DE RAMA EN GIT.....	12
AÑADIR ARCHIVOS (<i>git add</i>) Y GUARDAR CAMBIOS (<i>git commit</i>)	13
VISUALIZACIÓN DEL HISTORIAL DE COMMITS (<i>git log</i>)	15
CANCELAR CAMBIOS SOBRE UN FICHERO (<i>git checkout</i>) O SOBRE TODOS LOS FICHEROS (<i>git reset</i>)	16
PERSONALIZACIÓN DEL LOG Y GENERACIÓN DE ALIAS (<i>git alias</i>)	18
ARCHIVO <i>.gitignore</i>	20
COMPROBAR QUÉ SE HA CAMBIADO DESDE EL ÚLTIMO COMMIT (<i>git diff</i>).....	21
DESPLAZAMIENTO DENTRO DE UNA RAMA (<i>git checkout</i>).....	22
VUELTA A UN ESTADO ANTERIOR (<i>git reset --hard</i>) Y LOG AMPLIADO (<i>git reflog</i>)	25
GENERACIÓN DE ETIQUETAS (<i>git tag</i>)	28
CREACIÓN DE NUEVAS RAMAS (<i>git branch</i>) Y MOVIMIENTO ENTRE RAMAS (<i>git switch</i>)	32
FUSIÓN DE RAMAS (<i>git merge</i>)	35
A. CONFLICTOS EN LA FUSIÓN DE RAMAS	37
ALMACENAR TEMPORALMENTE CAMBIOS (<i>git stash</i>).....	40
REINTEGRACIÓN DE RAMAS	43
ELIMINACIÓN DE RAMAS EN GIT.....	44
GITHUB	46
A. ¿QUÉ ES GITHUB?	46
B. PRIMEROS PASOS EN GITHUB. CONOCIENDO LA INTERFAZ	47
C. CREANDO UN REPOSITORIO.....	47
D. LOCAL Y REMOTO	50
ENLAZAR UN REPOSITORIO LOCAL CON UN REPOSITORIO REMOTO (<i>git remote</i>)	51
SUBIR EL REPOSITORIO LOCAL AL REPOSITORIO REMOTO (<i>git push</i>).....	51
SUBIDA DE CAMBIOS A GITHUB (<i>git push</i>).....	52

COMPROBACIÓN DE EXISTENCIA DE CAMBIOS EN REMOTO (<i>git fetch</i>) Y ACTUALIZACIÓN DEL REPOSITORIO LOCAL CON LOS CAMBIOS REMOTOS (<i>git pull</i>) ..	55
DESCARGA DE UN REPOSITORIO REMOTO (<i>git clone</i>) ..	58
PROTEGIENDO NUESTRO PROYECTO FRENTE A CAMBIOS.....	59
CONCEPTO DE FORK EN GITHUB ..	61
FLUJO COLABORATIVO EN GITHUB.....	64
PULL REQUEST EN GITHUB.....	65
SINCRONIZACIÓN DE UN FORK EN GITHUB.....	69
USO DE HERRAMIENTAS GRÁFICAS PARA GIT Y GITHUB ..	72
A. GITHUB DESKTOP	72
B. GITKRAKEN.....	72
C. SOURCETREE	73
D. FORK	73
GLOSARIO DE COMANDOS - GIT/GITHUB CHEATSHEET.....	75

INTRODUCCIÓN

A. ¿QUÉ ES GIT?

Git 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.

B. 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](#) 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).



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](#) el cual es considerado una muy buena guía de inicio.

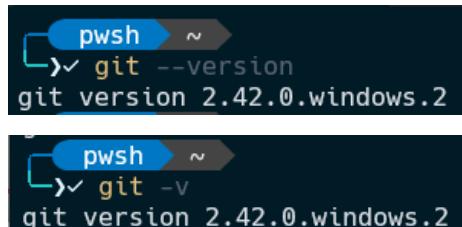
Una vez instalado tenemos varias opciones para utilizarlo:

- 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.

C. 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)
```

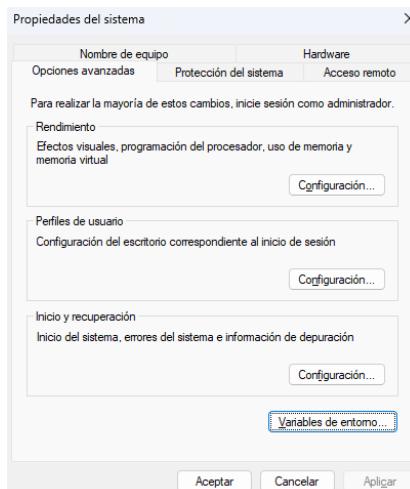


```
pwsh ~
> git --version
git version 2.42.0.windows.2

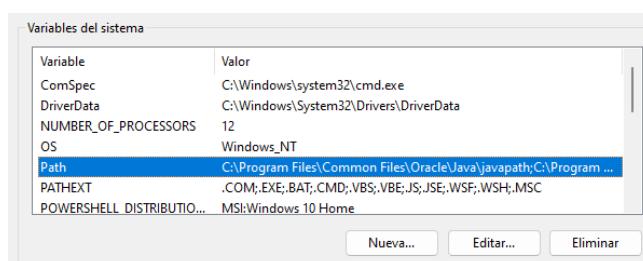
pwsh ~
> git -v
git version 2.42.0.windows.2
```

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...”:

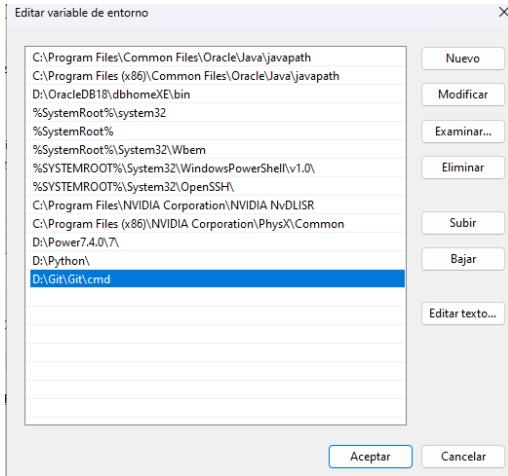


Y en la nueva la nueva ventana que se abre buscar en las variables del sistema, la variable Path:



Variable	Valor
ComSpec	C:\Windows\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
NUMBER_OF_PROCESSORS	12
OS	Windows_NT
Path	C:\Program Files\Common Files\Oracle\Java\javapath;C:\Program ...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
POWERSHELL DISTRIBUTIO...	MSI:Windows 10 Home

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”:



Otro comando que podemos utilizar es simplemente el comando git:

```
git
[> pwsh > ~]
>>> git
usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           [--config-env=<name>=<envvar>] <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
  clone      Clone a repository into a new directory
  init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
  add        Add file contents to the index
  mv        Move or rename a file, a directory, or a symlink
  restore    Restore working tree files
  rm        Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
  bisect    Use binary search to find the commit that introduced a bug
  diff      Show changes between commits, commit and working tree, etc
  grep      Print lines matching a pattern
  log       Show commit logs
  show      Show various types of objects
  status    Show the working tree status

grow, mark and tweak your common history
  branch   List, create, or delete branches
  commit   Record changes to the repository
  merge    Join two or more development histories together
  rebase   Reapply commits on top of another base tip
  reset   Reset current HEAD to the specified state
  switch  Switch branches
  tag     Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
  fetch   Download objects and refs from another repository
  pull    Fetch from and integrate with another repository or a local branch
  push    Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
See 'git help git' for an overview of the system.
```

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.

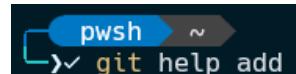
```
pwsh ~
git help -a
See 'git help <command>' to read about a specific subcommand

Main Porcelain Commands
add           Add file contents to the index
am            Apply a series of patches from a mailbox
archive       Create an archive of files from a named tree
bisect        Use binary search to find the commit that introduced a bug
branch       List, create, or delete branches
bundle        Move objects and refs by archive
checkout      Switch branches or restore working tree files
cherry-pick   Apply the changes introduced by some existing commits
citool        Graphical alternative to git-commit
clean         Remove untracked files from the working tree
clone         Clone a repository into a new directory
commit        Record changes to the repository
describe     Give an object a human readable name based on an available ref
diff          Show changes between commits, commit and working tree, etc
fetch         Download objects and refs from another repository
format-patch Prepare patches for e-mail submission
gc            Cleanup unnecessary files and optimize the local repository
gitk          The Git repository browser
grep          Print lines matching a pattern
gui           A portable graphical interface to Git
init          Create an empty Git repository or reinitialize an existing one
```

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

```
pwsh ~
git help -g
The Git concept guides are:
core-tutorial    A Git core tutorial for developers
credentials      Providing usernames and passwords to Git
cvs-migration    Git for CVS users
diffcore         Tweaking diff output
everyday         A useful minimum set of commands for Everyday Git
faq              Frequently asked questions about using Git
glossary         A Git Glossary
namespaces       Git namespaces
remote-helpers   Helper programs to interact with remote repositories
submodules       Mounting one repository inside another
tutorial         A tutorial introduction to Git
tutorial-2       A tutorial introduction to Git: part two
workflows        An overview of recommended workflows with Git
```

- `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:



git-add(1) Manual Page

NAME

git-add - Add file contents to the index

SYNOPSIS

```
git add [-verbose | -v] [-dry-run | -n] [-force | -f] [-interactive | -i] [-patch | -p]
[-edit | -e] [-(no-)all] [-(no-)ignore-removal | [-update | -u]] [-sparse]
[-intent-to-add | -I] [-(refresh)] [-(ignore-errors)] [-(ignore-missing)] [-(renormalize]
[-chmod(+|-)x] [-(pathspec-from-file=<file> | -pathspec-file-null)]
[-] [<pathspec>...]
```

DESCRIPTION

This command updates the index using the current content found in the working tree, to prepare the content staged for the next commit. It typically adds the current content of existing paths as a whole, but with some options it can also be used to add content with only part of the changes made to the working tree files applied, or remove paths that do not exist in the working tree anymore.

The "index" holds a snapshot of the content of the working tree, and it is this snapshot that is taken as the contents of the next commit. Thus after making any changes to the working tree, and before running the commit command, you must use the `git add` command to add any new or modified files to the index.

This command can be performed multiple times before a commit. It only adds the content of the specified file(s) at the time the `git add` command is run; if you want subsequent changes included in the next commit, then you must run `git add` again to add the new content to the index.

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”:

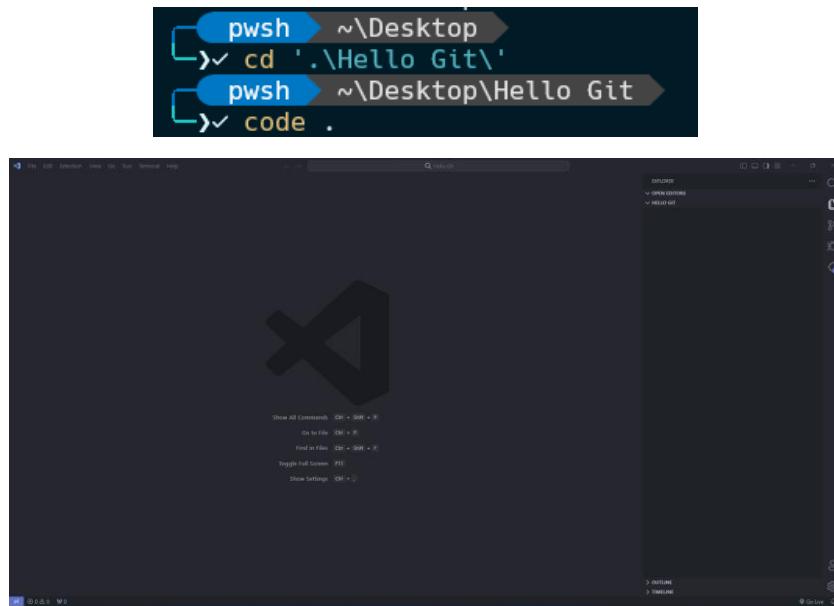
```
mkdir "nombre del directorio"
```

```
pwsh ~\Desktop
> mkdir "Hello Git"

Directory: C:\Users\alber\Desktop

Mode                LastWriteTime         Length Name
----                -              -           -
d---       20/11/2023     16:28          Hello Git
```

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:



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.

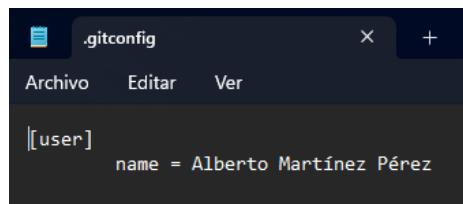
Para crear el usuario usaremos el siguiente comando:

```
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:

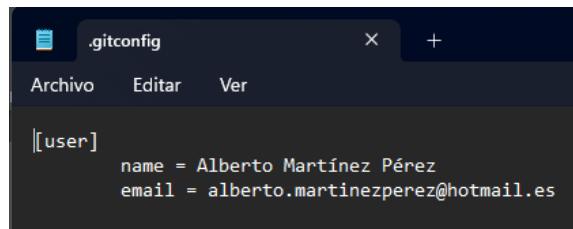


Para crear el mail usamos un comando similar:

```
git config --global user.email "email del usuario"
```



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



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:

```
  hellogit.py X  
  hellogit.py  
1   print('Hello Git!')
```

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

```
pwsh ~\Desktop\Hello Git
> ls

Directory: C:\Users\alber\Desktop\Hello Git

Mode          LastWriteTime      Length Name
----          -              -          -
-a---  20/11/2023 16:58          20  hellogit.py
```

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

git init

```
[pwsh ~\Desktop\Hello Git] > git init  
Initialized empty Git repository in C:/Users/alber/Desktop/Hello Git/.git/
```

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

```
pwsh ~\Desktop\Hello Git ➜ master ?1
❯ ls -force

Directory: C:\Users\alber\Desktop\Hello Git

Mode                LastWriteTime      Length Name
----                -----          ----  -
d--h-        20/11/2023    17:08          .git
-a---        20/11/2023    17:00           19  hellogit.py
```

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:

```
pwsh ~\Desktop\Hello Git >? master ?1
> ls .\.git

Directory: C:\Users\alber\Desktop\Hello Git\.git

Mode          LastWriteTime      Length Name
----          -----          ---- 
d---          20/11/2023    17:08      hooks
d---          20/11/2023    17:08      info
d---          20/11/2023    17:08      objects
d---          20/11/2023    17:08      refs
-a--          20/11/2023    17:08      130 config
-a--          20/11/2023    17:08      73  description
-a--          20/11/2023    17:08      23  HEAD
```

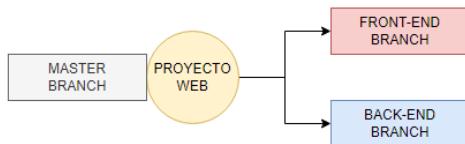
CONCEPTO DE RAMA EN GIT

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:



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
```

```
pwsh ~\Desktop\Hello Git > master ?1
└─> git branch -m main
pwsh ~\Desktop\Hello Git > main ?1
└─>
```

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

```
git config --global init.defaultBranch nombreDeLaRamaInicial
```

```
pwsh ~\Desktop\Hello Git > main ?1
└─> git config --global init.defaultBranch main
```

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

```
[init]
  defaultBranch = main
```

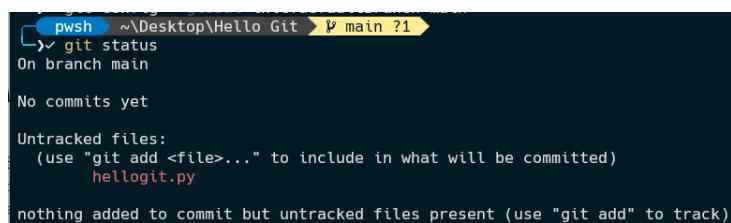
AÑADIR ARCHIVOS (*git add*) Y GUARDAR CAMBIOS (*git commit*)

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
```



```
pwsh ~\Desktop\Hello Git > main ?1
> git status
On branch main

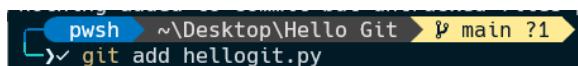
No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    hellogit.py

nothing added to commit but untracked files present (use "git add" to track)
```

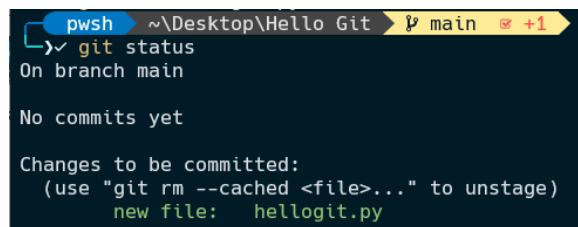
Con este comando vamos a poder ver en qué rama nos encontramos, qué commits se han realizado en ella y qué archivos se encuentran sin seguimiento (*untracked*). Para que estos archivos pasen de este estado a uno de seguimiento se deben añadir y para ello debemos utilizar el siguiente comando:

```
git add nombreFichero1
```



```
pwsh ~\Desktop\Hello Git > main ?1
> git add hellogit.py
```

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



```
pwsh ~\Desktop\Hello Git > main ?1
> git status
On branch main

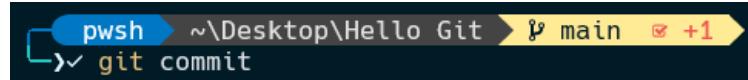
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   hellogit.py
```

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.

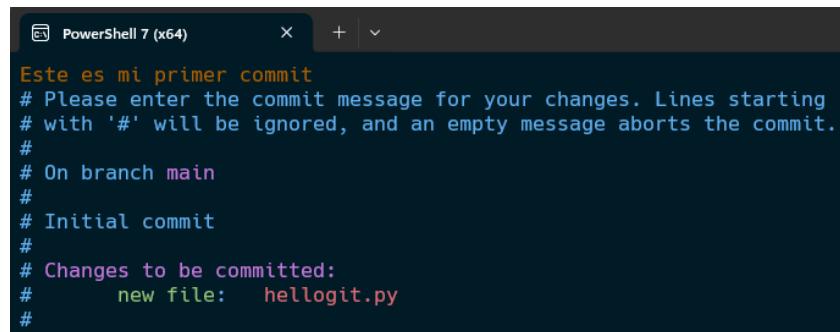
El archivo `hellogit.py` ha pasado de estar sin seguimiento a tenerlo (ahora está en estado *staged*), es momento por tanto de hacer nuestro primer comando del proyecto:

git commit



Esto hará que se nos habrá una ventana de vim:

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.



Si guardamos cambios se realizará el commit, aun así en este caso no vamos a guardarlos 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"
```



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

```
[main (root-commit) 28d9310] Este es mi primer commit  
 1 file changed, 1 insertion(+)  
 create mode 100644 hellogit.py
```

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:

```
pwsh ~\Desktop\Hello Git p main  
> git status  
On branch main  
nothing to commit, working tree clean
```

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
```

```
pwsh ~\Desktop\Hello Git p main  
> git log  
commit 28d9310f39ac716d756f0f1fc015186cc74670e (HEAD -> main)  
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>  
Date: Mon Nov 20 17:53:44 2023 +0100  
  
Este es mi primer commit
```

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:

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

```
pwsh ~\Desktop\Hello Git > p main
└─> git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    hellogit2.py

nothing added to commit but untracked files present (use "git add" to track)
```

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

```
pwsh ~\Desktop\Hello Git > p main ?1
└─> git add hellogit2.py
└─> pwsh ~\Desktop\Hello Git > p main ✘ +1
└─> git commit -m "Este es mi segundo commit"
[main 52d856f] Este es mi segundo commit
  1 file changed, 1 insertion(+)
  create mode 100644 hellogit2.py
```

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

```
pwsh ~\Desktop\Hello Git > p main
└─> git log
commit 52d856f7c0e21644dcdb2cc9b6950f5e276d605e (HEAD -> main)
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date: Mon Nov 20 18:07:35 2023 +0100

  Este es mi segundo commit

commit 28d9310f39ac716d756f0f1fcfd015186cc74670e
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date: Mon Nov 20 17:53:44 2023 +0100

  Este es mi primer commit
```

Es decir, lo que tenemos ahora mismo es esto:



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:

```
hellogit.py M X  hellogit2.py
hellogit.py
You, 1 second ago | 1 author (You)
1 | print('Ahora estoy editando el fichero')
```

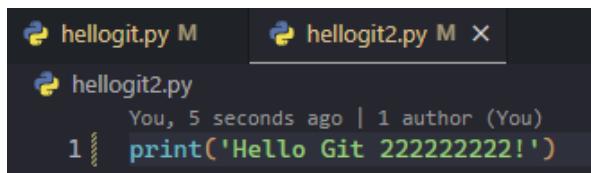
Si ahora hacemos un status del proyecto veremos que este cambio se ha registrado:

```
pwsh ~\Desktop\Hello Git p main
> git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   hellogit.py

no changes added to commit (use "git add" and/or "git commit -a")
```

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:



Al hacer un nuevo status veremos que este cambio registrado:

```
pwsh ~\Desktop\Hello Git p main ~1
> git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   hellogit.py
    modified:   hellogit2.py

no changes added to commit (use "git add" and/or "git commit -a")
```

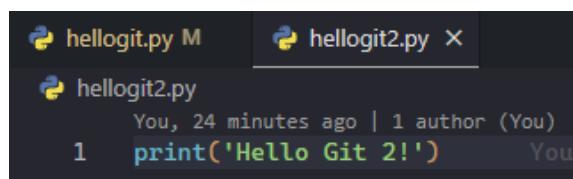
Supongamos que tras hacer estos cambios vemos que nuestro código no funciona como antes o que no nos gusta el resultado final y queremos volver a como lo teníamos antes.

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:

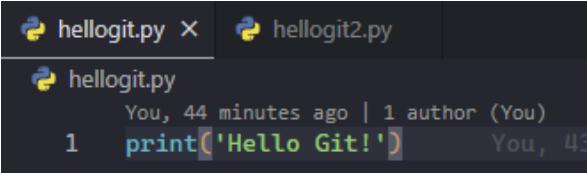
```
git checkout nombreFichero
```

```
pwsh ~\Desktop\Hello Git p main ~2
> git checkout hellogit2.py
Updated 1 path from the index
```

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:



Vamos a hacer lo mismo con el otro fichero:



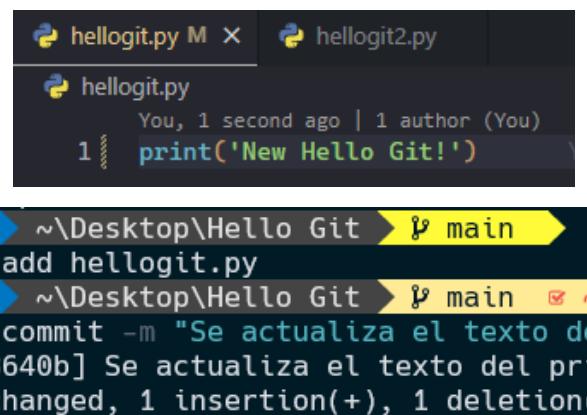
```
pwsh ~\Desktop\Hello Git > main ~1
> git checkout hellogit.py
Updated 1 path from the index
```

```
hellogit.py X hellogit2.py
hellogit.py
You, 44 minutes ago | 1 author (You)
1 print('Hello Git!')
You, 44 minutes ago | 1 author (You)
```

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:



```
hellogit.py M X hellogit2.py
hellogit.py
You, 1 second ago | 1 author (You)
1 print('New Hello Git!')
```

```
pwsh ~\Desktop\Hello Git > main
> git add hellogit.py
pwsh ~\Desktop\Hello Git > main ~1
> git commit -m "Se actualiza el texto del print"
[main a73640b] Se actualiza el texto del print
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Si ahora hacemos un log veremos los 3 commits:



```
git log
commit a73640b02c7d00c37f3326c33a0a36115b1ce5b9 (HEAD -> main)
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date:   Mon Nov 20 18:56:23 2023 +0100

  Se actualiza el texto del print

commit 52d856f7c0e21644dcdb2cc9b6950f5e276d605e
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date:   Mon Nov 20 18:07:35 2023 +0100

  Este es mi segundo commit

commit 28d9310f39ac716d756f0f1fc015186cc74670e
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date:   Mon Nov 20 17:53:44 2023 +0100

  Este es mi primer commit
```

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
```

```
pwsh ~\Desktop\Hello Git ➜ main
git log --graph
* commit a73640b02c7d00c37f3326c33a0a36115b1ce5b9 (HEAD -> main)
| Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
| Date: Mon Nov 20 18:56:23 2023 +0100
|
|     Se actualiza el texto del print
|
* commit 52d856f7c0e21644dcdb2cc9b6950f5e276d605e
| Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
| Date: Mon Nov 20 18:07:35 2023 +0100
|
|     Este es mi segundo commit
|
* commit 28d9310f39ac716d756f0f1fc0d015186cc74670e
| Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
| Date: Mon Nov 20 17:53:44 2023 +0100
|
Este es mi primer commit
```

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
```

```
pwsh ~\Desktop\Hello Git ➜ main
git log --graph --pretty=oneline
* a73640b02c7d00c37f3326c33a0a36115b1ce5b9 (HEAD -> main) Se actualiza el texto del print
* 52d856f7c0e21644dcdb2cc9b6950f5e276d605e Este es mi segundo commit
* 28d9310f39ac716d756f0f1fc0d015186cc74670e Este es mi primer commit
```

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
```

```
pwsh ~\Desktop\Hello Git ➜ main
git log --graph --decorate --all --oneline
* a73640b (HEAD -> main) Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

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:

```
pwsh ~\Desktop\Hello Git ➜ main
git config --global alias.tree "log --graph --decorate --all --oneline"
```

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

```
[alias]
    tree = log --graph --decorate --all --oneline
```

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

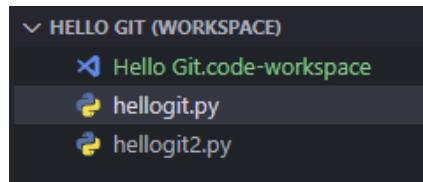


```
pwsh ~\Desktop\Hello Git main
> git tree
* a73640b (HEAD -> main) Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

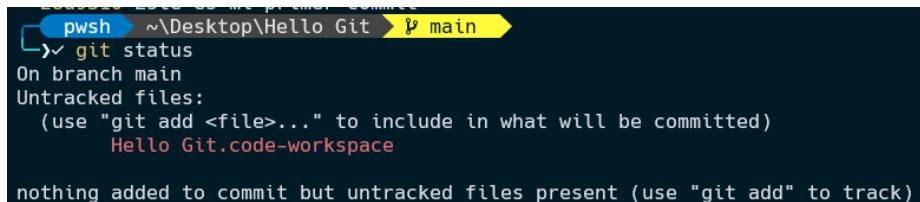
ARCHIVO .gitignore

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 Code²:



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



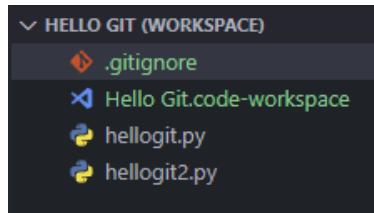
```
pwsh ~\Desktop\Hello Git main
> git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Hello Git.code-workspace

nothing added to commit but untracked files present (use "git add" to track)
```

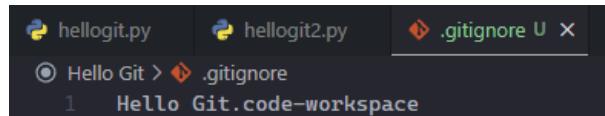
Pero a diferencia de lo que ocurre con los ficheros hellogit.py y hellogit2.py, este fichero no tiene ningún sentido que se guarde en un commit porque es un fichero que guarda datos acerca de cómo queremos que se visualice este proyecto en nuestro Visual Studio Code por ello nunca lo guardaremos en el 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:

² Para crear este fichero hay que hacer clic en “File” → “Save workspace as...” y elegir el nombre que tendrá el fichero.

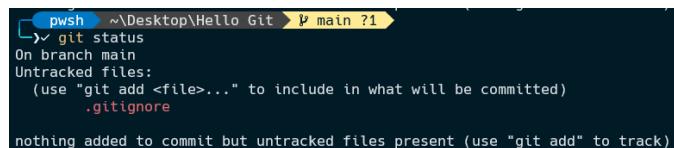


Ahora los ficheros o rutas que añadamos dentro de este fichero nunca se tomarán como archivos sin seguimiento, aunque sufran cambios:

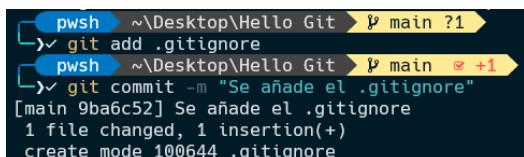


Aquí también podemos usar expresiones como por ejemplo *.class esto haría que todos los archivos .class de un proyecto basado en Java nunca se contaran como ficheros sin seguimiento.

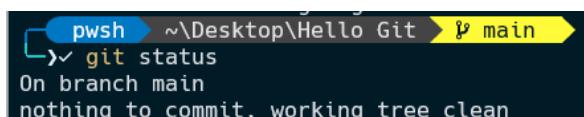
Si ahora hacemos un git status veremos que nos aparece el archivo .gitignore como archivo sin seguimiento pero el archivo del workspace ya no aparecerá:



Vamos a hacer un commit para este .gitignore:



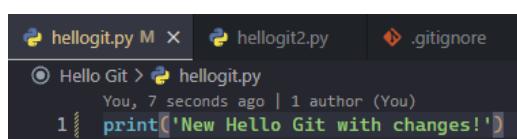
Si ahora volvemos a hacer un status veremos que no hay nada para meter en el stage:



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:



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
```

```
pwsh ~\Desktop\Hello Git p main
git diff
diff --git a/hellogit.py b/hellogit.py
index 9267b05..4580af4 100644
--- a/hellogit.py
+++ b/hellogit.py
@@ -1 +1 @@
-print('New Hello Git!')
\ No newline at end of file
+print('New Hello Git with changes!')
\ No newline at end of file
```

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:



O lo que es lo mismo:

```
pwsh ~\Desktop\Hello Git p main ~1
git log
commit 9ba6c52f704c87afeaa68f1a6e2661a08122e378 (HEAD -> main)
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date: Mon Nov 20 19:30:47 2023 +0100

    Se añade el .gitignore

commit a73640b02c7d00c37f3326c33a0a36115b1ce5b9
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date: Mon Nov 20 18:56:23 2023 +0100

    Se actualiza el texto del print

commit 52d856f7c0e21644dcdb2cc9b6950f5e276d605e
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date: Mon Nov 20 18:07:35 2023 +0100

    Este es mi segundo commit

commit 28d9310f39ac716d756f0f1fc015186cc74670e
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date: Mon Nov 20 17:53:44 2023 +0100

    Este es mi primer commit
```

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

```
pwsh ~\Desktop\Hello Git main ~1
> git checkout 28d9310f39ac716d756f0f1fc015186cc74670e
error: Your local changes to the following files would be overwritten by checkout:
    hellogit.py
Please commit your changes or stash them before you switch branches.
Aborting
```

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:

```
pwsh ~\Desktop\Hello Git main ~1
> git checkout hellogit.py
Updated 1 path from the index
```

Y ahora sí que podremos movernos con el comando anterior:

```
pwsh ~\Desktop\Hello Git main
> git checkout 28d9310f39ac716d756f0f1fc015186cc74670e
Note: switching to '28d9310f39ac716d756f0f1fc015186cc74670e'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

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

    git switch -c <new-branch-name>

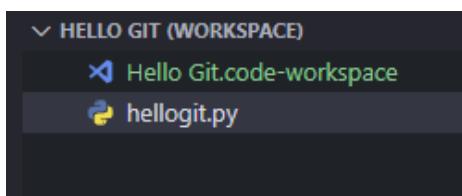
Or undo this operation with:

    git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 28d9310 Este es mi primer commit
```

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

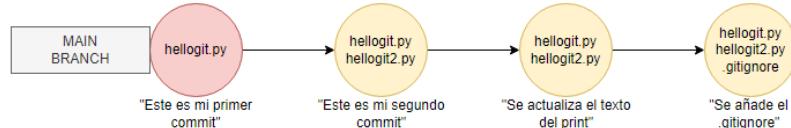


Esto se debe a que hemos vuelto atrás en el tiempo, tenemos el proyecto en el mismo estado en el que se encontraba cuando hicimos el primer commit (el archivo de workspace aunque se creó después, como nunca ha sido asociado a una "foto" en concreto, permanece).

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



Tras hacer el checkout nos encontramos aquí:



Hemos regresado a un punto anterior del proyecto.

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

```

pwsh ~\Desktop\Hello Git > git log
commit 28d9310f39ac716d756f0f1fcfd015186cc74670e (HEAD)
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date:   Mon Nov 20 17:53:44 2023 +0100

Este es mi primer commit

```

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):

```

pwsh ~\Desktop\Hello Git > git tree
* 9ba6c52 (main) Se añade el .gitignore
* a73640b Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 (HEAD) Este es mi primer commit

```

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

```

pwsh ~\Desktop\Hello Git > git checkout 9ba6c52
Previous HEAD position was 28d9310 Este es mi primer commit
HEAD is now at 9ba6c52 Se añade el .gitignore

```

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:

```
pwsh ~\Desktop\Hello Git > git tree
* 9ba6c52 (HEAD, main) Se añade el .gitignore
* a73640b Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

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



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
```

```
pwsh ~\Desktop\Hello Git > git checkout main
Switched to branch 'main'
```

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).

```
pwsh ~\Desktop\Hello Git > git tree
* 9ba6c52 (HEAD -> main) Se añade el .gitignore
* a73640b Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

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:

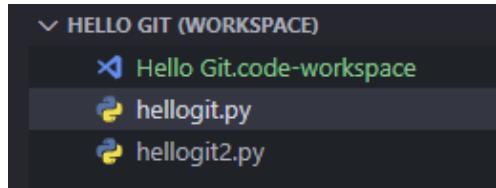
```
pwsh ~\Desktop\Hello Git > p main
└─> git tree
* 9ba6c52 (HEAD -> main) Se añade el .gitignore
* a73640b Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

Para hacer esto debemos usar el comando:

```
git reset --hard hashÚnico
```

```
pwsh ~\Desktop\Hello Git > p main
└─> git reset --hard 52d856f
HEAD is now at 52d856f Este es mi segundo commit
```

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:



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:

```
pwsh ~\Desktop\Hello Git > p main ?1
└─> git log
commit 52d856f7c0e21644dcdb2cc9b6950f5e276d605e (HEAD -> main)
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date: Mon Nov 20 18:07:35 2023 +0100

Este es mi segundo commit

commit 28d9310f39ac716d756f0f1fc0d015186cc74670e
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date: Mon Nov 20 17:53:44 2023 +0100

Este es mi primer commit
pwsh ~\Desktop\Hello Git > p main ?1
└─> git tree
* 52d856f (HEAD -> main) Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

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
```

```
pwsh > ~\Desktop\Hello Git > p main ?1
>>> git reflog
52d856f (HEAD -> main) HEAD@{0}: reset: moving to 52d856f
9ba6c52 HEAD@{1}: checkout: moving from 9ba6c52f704c87afeaa68f1a6e2661a08122e378 to main
9ba6c52 HEAD@{2}: checkout: moving from 28d9310f39ac716d756f0f1fc015186cc74670e to 9ba6c52
28d9310 HEAD@{3}: checkout: moving from main to 28d9310f39ac716d756f0f1fc015186cc74670e
9ba6c52 HEAD@{4}: commit: Se añade el .gitignore
a73640b HEAD@{5}: commit: Se actualiza el texto del print
52d856f (HEAD -> main) HEAD@{6}: reset: moving to HEAD
52d856f (HEAD -> main) HEAD@{7}: commit: Este es mi segundo commit
28d9310 HEAD@{8}: commit (initial): Este es mi primer commit
```

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:

```
pwsh > ~\Desktop\Hello Git > p main ?1
>>> git checkout 9ba6c52
Note: switching to '9ba6c52'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

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

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 9ba6c52 Se añade el .gitignore
```

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:

```
pwsh > ~\Desktop\Hello Git > p detached at ~9ba6c52
>>> git tree
* 9ba6c52 (HEAD) Se añade el .gitignore
* a73640b Se actualiza el texto del print
* 52d856f (main) Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

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:

```
pwsh > ~\Desktop\Hello Git > p detached at ~9ba6c52
>>> git checkout main
Warning: you are leaving 2 commits behind, not connected to
any of your branches:

  9ba6c52 Se añade el .gitignore
  a73640b Se actualiza el texto del print

If you want to keep them by creating a new branch, this may be a good time
to do so with:

  git branch <new-branch-name> 9ba6c52

Switched to branch 'main'
```

Haciendo un git tree lo podremos ver:

```
pwsh ~\Desktop\Hello Git ➜ main ?1
└─> git tree
* 52d856f (HEAD -> main) Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

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

```
pwsh ~\Desktop\Hello Git ➜ main ?1
└─> git reset --hard 9ba6c52
HEAD is now at 9ba6c52 Se añade el .gitignore
```

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

```
pwsh ~\Desktop\Hello Git ➜ main
└─> git log
commit 9ba6c52f704c87afeaa68f1a6e2661a08122e378 (HEAD -> main)
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date: Mon Nov 20 19:30:47 2023 +0100

    Se añade el .gitignore

commit a73640b02c7d00c37f3326c33a0a36115b1ce5b9
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date: Mon Nov 20 18:56:23 2023 +0100

    Se actualiza el texto del print

commit 52d856f7c0e21644dcdb2cc9b6950f5e276d605e
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date: Mon Nov 20 18:07:35 2023 +0100

    Este es mi segundo commit

commit 28d9310f39ac716d756f0f1fc015186cc74670e
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date: Mon Nov 20 17:53:44 2023 +0100

    Este es mi primer commit

└─> git tree
* 9ba6c52 (HEAD -> main) Se añade el .gitignore
* a73640b Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

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 creímos perdidos.

GENERACIÓN DE ETIQUETAS (*git tag*)

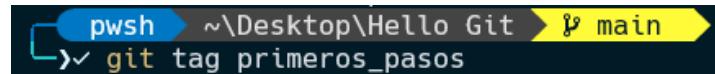
Vamos a crear tags que nos sirvan para etiquetar commits. Esto lo vamos a realizar cuando tengamos un commit importante que nos sirva para identificar rápidamente un commit.

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

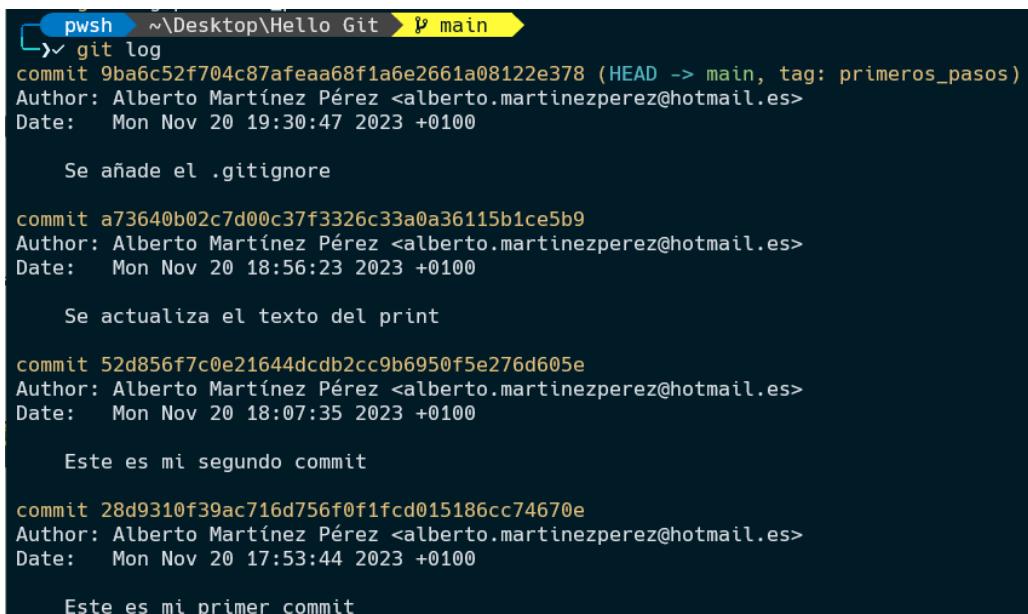
cada vez que lanzamos una versión utilizable de nuestra aplicación, esta lleva aparejada una etiqueta.

Para crear un tag vamos a utilizar el siguiente comando

```
git tag nombre_del_tag3
```



Si ahora hacemos un log veremos lo siguiente:



```
git log
commit 9ba6c52f704c87afeaa68f1a6e2661a08122e378 (HEAD -> main, tag: primeros_pasos)
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date:   Mon Nov 20 19:30:47 2023 +0100

    Se añade el .gitignore

commit a73640b02c7d00c37f3326c33a0a36115b1ce5b9
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date:   Mon Nov 20 18:56:23 2023 +0100

    Se actualiza el texto del print

commit 52d856f7c0e21644dcdb2cc9b6950f5e276d605e
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date:   Mon Nov 20 18:07:35 2023 +0100

    Este es mi segundo commit

commit 28d9310f39ac716d756f0f1fc015186cc74670e
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date:   Mon Nov 20 17:53:44 2023 +0100

    Este es mi primer commit
```

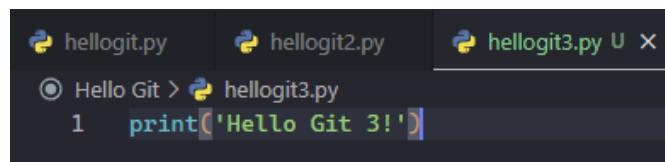
Junto al último commit ha aparecido un tag con el nombre `primeros_pasos`. De esta forma, este punto del desarrollo tiene un nombre identificativo.

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

```
git tag
```



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



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 (`_`).

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

```
pwsh ~\Desktop\Hello Git ➜ main
└─> git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    hellogit3.py

nothing added to commit but untracked files present (use "git add" to track)
```

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 .

```
pwsh ~\Desktop\Hello Git ➜ main ?1
└─> git add .
```

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

```
pwsh ~\Desktop\Hello Git ➜ main ✘ +1
└─> git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   hellogit3.py
```

Una vez hecho esto hacemos un commit:

```
pwsh ~\Desktop\Hello Git ➜ main ✘ +1
└─> git commit -m "Este es mi quinto commit"
[main b8b3068] Este es mi quinto commit
  1 file changed, 1 insertion(+)
  create mode 100644 hellogit3.py
```

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

```
pwsh ~\Desktop\Hello Git ➜ main
└─> git tree
* b8b3068 (HEAD -> main) Este es mi quinto commit
* 9ba6c52 (tag: primeros_pasos) Se añade el .gitignore
* a73640b Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

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
```

```

pwsh ~\Desktop\Hello Git > p main
>>> git checkout tags/primeros_pasos
Note: switching to 'tags/primeros_pasos'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

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

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 9ba6c52 Se añade el .gitignore

```

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

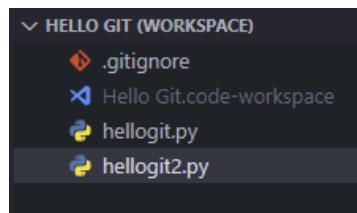
```

pwsh ~\Desktop\Hello Git > p detached at primeros_pasos
>>> git tree
* b8b3068 (main) Este es mi quinto commit
* 9ba6c52 (HEAD, tag: primeros_pasos) Se añade el .gitignore
* a73640b Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 Este es mi primer commit

```

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:



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

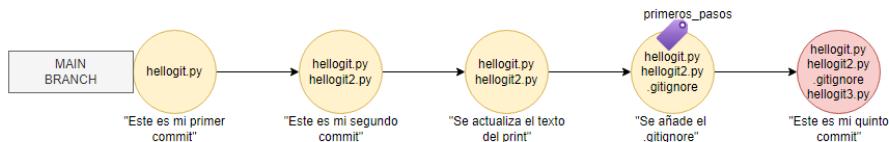
```

pwsh ~\Desktop\Hello Git > p detached at primeros_pasos
>>> git checkout main
Previous HEAD position was 9ba6c52 Se añade el .gitignore
Switched to branch 'main'
pwsh ~\Desktop\Hello Git > p main
>>> git tree
* b8b3068 (HEAD -> main) Este es mi quinto commit
* 9ba6c52 (tag: primeros_pasos) Se añade el .gitignore
* a73640b Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 Este es mi primer commit

```

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

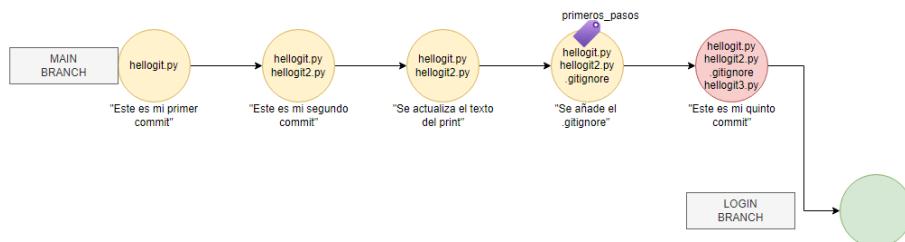
Si volvemos a ver una representación gráfica de nuestro proyecto en este punto sería algo así:



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 encontramos 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 de la rama `main`.



Para hacer esto en comando debemos usar el comando:

```
git branch nombreNuevaRama
```

```
pwsh ~\Desktop\Hello Git p main
> git branch login
```

Si ahora hacemos un `git tree`:

```
pwsh ~\Desktop\Hello Git p main
> git tree
* b8b3068 (HEAD -> main, login) Este es mi quinto commit
* 9ba6c52 (tag: primeros_pasos) Se añade el .gitignore
* a73640b Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

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

Para cambiarnos de rama vamos a utilizar otro comando:

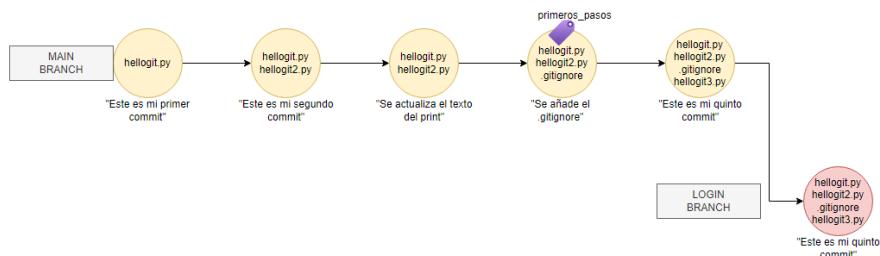
```
git switch nombreRama
```

```
pwsh ~\Desktop\Hello Git ➜ main
❯ git switch login
Switched to branch 'login'
```

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

```
pwsh ~\Desktop\Hello Git ➜ login
* b8b3068 (HEAD -> login, main) Este es mi quinto commit
* 9ba6c52 (tag: primeros_pasos) Se añade el .gitignore
* a73640b Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

Es decir, ahora nuestra situación es esta:



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:

```

login.py U X
Hello Git > login.py
1   print('Login')

▼ HELLO GIT (WORKSPACE)
  .gitignore
  Hello Git.code-workspace
  hellogit.py
  hellogit2.py
  hellogit3.py
  login.py

```

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

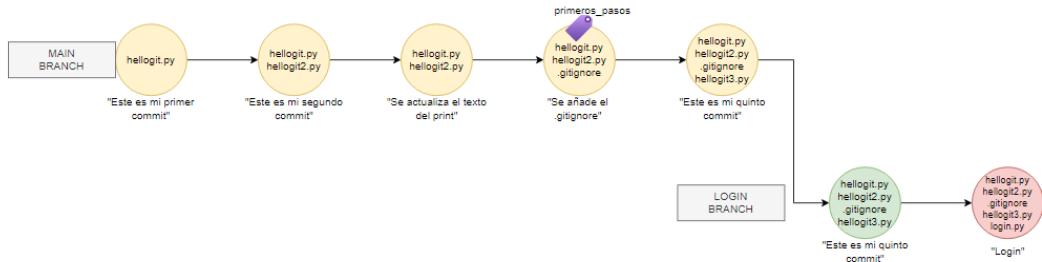
```
pwsh ~\Desktop\Hello Git ➜ login
❯ git status
On branch login
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    login.py

nothing added to commit but untracked files present (use "git add" to track)
```

Vamos a hacer el commit:

```
pwsh ~\Desktop\Hello Git > p login ?1  
>\v git add .  
pwsh ~\Desktop\Hello Git > p login ✘ +1  
>\v git commit -m "Login"  
[login bc3e1ac] Login  
1 file changed, 1 insertion(+)  
create mode 100644 login.py
```

Nuestra situación actual por tanto es la siguiente:



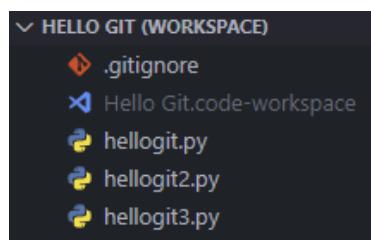
Si lo vemos en log sería esto:

```
pwsh ~\Desktop\Hello Git > p login  
>\v git log  
commit bc3e1acddc39b99af922dac3acd45fcfed86048 (HEAD -> login)  
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>  
Date: Mon Nov 20 23:41:11 2023 +0100  
  
    Login  
  
commit b8b3068f8bf7bf113c3a4231ad02ff226daa5169 (main)  
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>  
Date: Mon Nov 20 22:56:55 2023 +0100  
  
    Este es mi quinto commit  
  
commit 9ba6c52f704c87afeaa68f1a6e2661a08122e378 (tag: primeros_pasos)  
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>  
Date: Mon Nov 20 19:30:47 2023 +0100  
  
    Se añade el .gitignore  
  
commit a73640b02c7d00c37f3326c33a0a36115b1ce5b9  
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>  
Date: Mon Nov 20 18:56:23 2023 +0100  
  
    Se actualiza el texto del print  
  
commit 52d856f7c0e21644dcdb2cc9b6950f5e276d605e  
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>  
Date: Mon Nov 20 18:07:35 2023 +0100  
  
    Este es mi segundo commit  
  
commit 28d9310f39ac716d756f0f1fcfd015186cc74670e  
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>  
Date: Mon Nov 20 17:53:44 2023 +0100  
  
    Este es mi primer commit
```

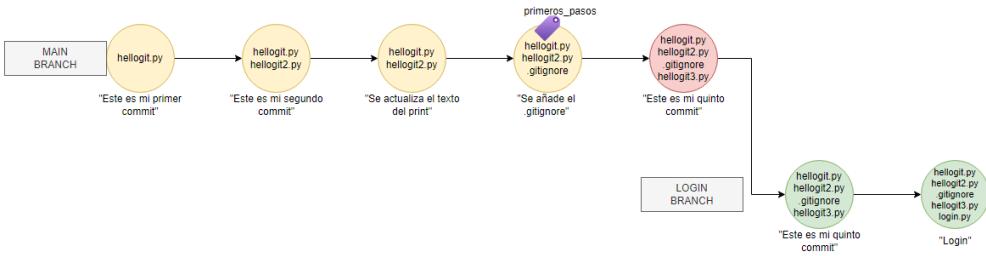
Ahora vamos a cambiar otra vez de rama y volver a la rama main:

```
pwsh ~\Desktop\Hello Git > p login  
>\v git switch main  
Switched to branch 'main'
```

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



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



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

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

```

git commit -m "Git 3 v2"
[main 1aeed13] Git 3 v2
 1 file changed, 1 insertion(+), 1 deletion(-)

```

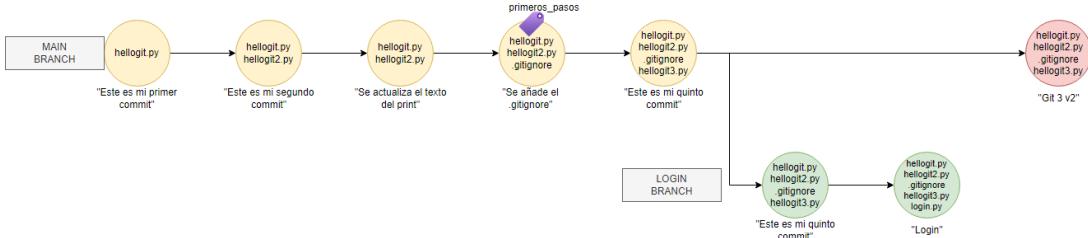
Y hacemos el correspondiente commit:

```

git add .
git commit -m "Git 3 v2"
[main 1aeed13] Git 3 v2
 1 file changed, 1 insertion(+), 1 deletion(-)

```

La situación por tanto es esta:



FUSIÓN DE RAMAS (git merge)

Ahora volvamos a la rama login:

```

git switch login
Switched to branch 'login'

```

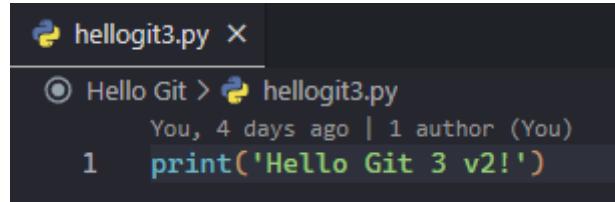
Supongamos que han pasado días y queremos comprobar si lo que se ha seguido realizando en la rama main es compatible todavía con lo que tenemos nosotros en nuestra rama login. Es decir, queremos tener los datos actualizados que tenga la rama main.

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

git merge ramaA Fusionar

```
pwsh ~\Desktop\Hello Git > login
> git merge main
Merge made by the 'ort' strategy.
  hellogit3.py | 2 +-
  1 file changed, 1 insertion(+), 1 deletion(-)
```

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:



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:

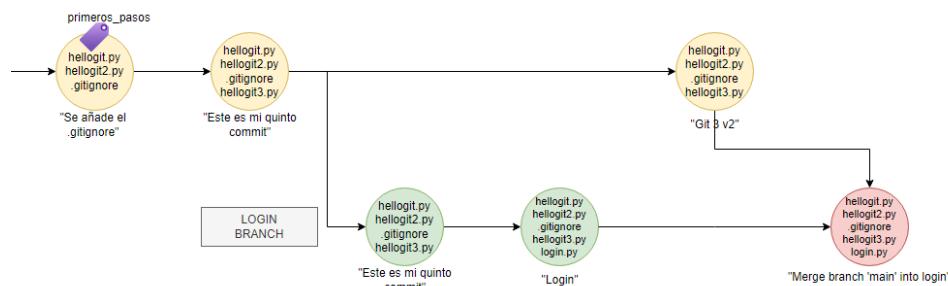
```
pwsh ~\Desktop\Hello Git > login
> git tree
* 38a836d (HEAD -> login) Merge branch 'main' into login
| \
| * 1aeed13 (main) Git 3 v2
* | bc3e1ac Login
| /
* b8b3068 Este es mi quinto commit
* 9ba6c52 (tag: primeros_pasos) Se añade el .gitignore
* a73640b Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

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

```
pwsh ~\Desktop\Hello Git > login
> git log
commit 38a836d8665c261988e0bf6d58120fb8912dc70f (HEAD -> login)
Merge: bc3e1ac 1aeed13
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date:   Fri Nov 24 18:42:05 2023 +0100

        Merge branch 'main' into login
```

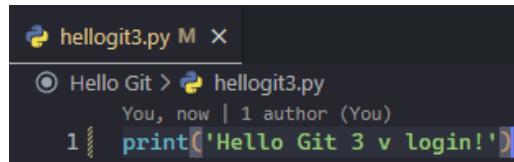
En la representación gráfica que estamos haciendo podríamos decir que ha ocurrido esto:



A. CONFLICTOS EN LA FUSIÓN DE RAMAS

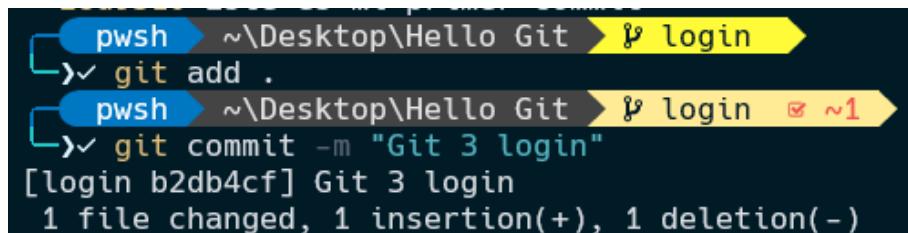
El ejemplo que hemos hecho para explicar un merge que funciona sin problema ya que no hay conflictos entre ramas, pero podemos tener una situación donde existe un conflicto entre ambas ramas, por ejemplo, porque dos programadores hayan tocado el mismo archivo realizando dos codificaciones diferentes.

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



```
git commit -m "Git 3 login"
[login b2db4cf] Git 3 login
 1 file changed, 1 insertion(+), 1 deletion(-)
```

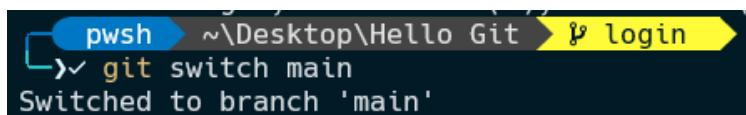
Y hacemos un commit:



```
git commit -m "Git 3 login"
[login b2db4cf] Git 3 login
 1 file changed, 1 insertion(+), 1 deletion(-)
```

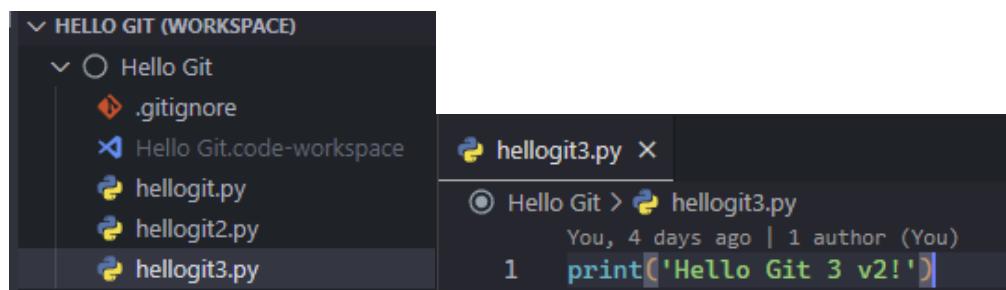
Es decir, estamos simulando una situación donde el equipo de login ha modificado un archivo que, en principio, es exclusivo de la rama main y que no debería ser modificado.

Ahora volvemos a la rama main:



```
git switch main
Switched to branch 'main'
```

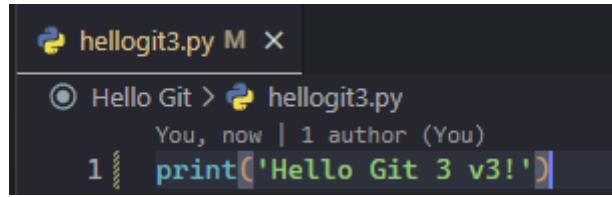
Y si vamos al proyecto veremos que no tenemos el archivo login.py y que en el archivo hellogit3.py tenemos el texto original y no el modificado en login:



```
git commit -m "Git 3 login"
[login b2db4cf] Git 3 login
 1 file changed, 1 insertion(+), 1 deletion(-)
```

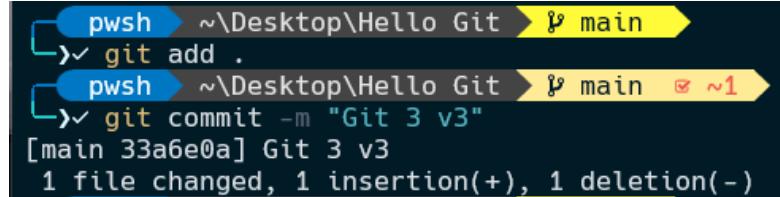
Esto ocurre porque cuando hicimos el “merge” fue en el login trayéndonos los cambios de main pero main no ha recibido ningún “merge” por lo que no puede tener los cambios que se han realizado en login.

Modificamos el archivo hellogit3.py:



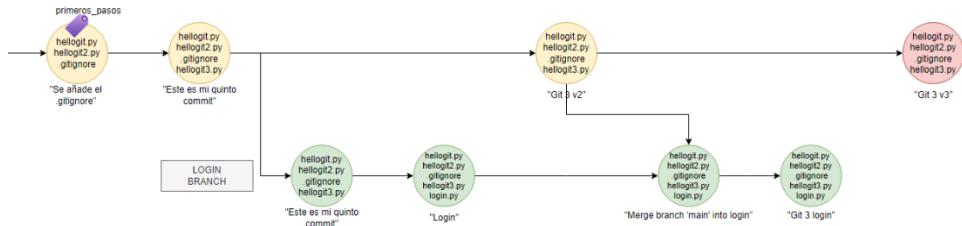
```
helloworld3.py M X
Hello Git > helloworld3.py
You, now | 1 author (You)
1 | print('Hello Git 3 v3!')
```

Y hacemos un commit en esta rama:



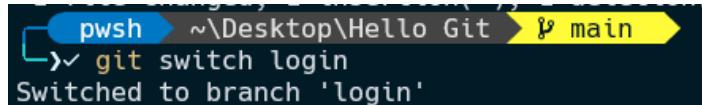
```
pwsh ~\Desktop\Hello Git > p main
> git add .
pwsh ~\Desktop\Hello Git > p main ✘ ~1
> git commit -m "Git 3 v3"
[main 33a6e0a] Git 3 v3
 1 file changed, 1 insertion(+), 1 deletion(-)
```

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:



```
pwsh ~\Desktop\Hello Git > p main
> git switch login
Switched to branch 'login'
```

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

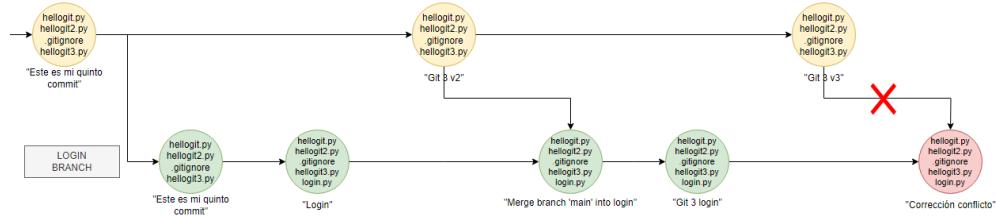


```
pwsh ~\Desktop\Hello Git > p login
> git merge main
Auto-merging hellogit3.py
CONFLICT (content): Merge conflict in hellogit3.py
Automatic merge failed; fix conflicts and then commit the result.
```

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:



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

hellogit3.py :

```

❶ Hello Git > 🐍 hellogit3.py
❷ You, 4 minutes ago | 1 author (You) | Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
1 <<<<< HEAD (Current Change)
2 print('Hello Git 3 v1!')
3 ==
4 print('Hello Git 3 v3!')
5 >>>>> main (Incoming Change)
6

```

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:

hellogit3.py :

```

❶ Hello Git > 🐍 hellogit3.py
❷ You, 12 minutes ago | 1 author (You)
1   print('Hello Git 3 v3!')

```

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 `hellogit3.py` de `main`.

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

```

pwsh ~\Desktop\Hello Git > 🐍 main into 🐍 login x1 | ✘ x1
git commit -m "Corrección conflicto"
error: Committing is not possible because you have unmerged files.
hint: Fix them up in the work tree, and then use 'git add/rm <file>'
hint: as appropriate to mark resolution and make a commit.
fatal: Exiting because of an unresolved conflict.
U      hellogit3.py

```

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

```

pwsh ~\Desktop\Hello Git > git main into login x1 | x1
git status
On branch login
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified: hellogit3.py

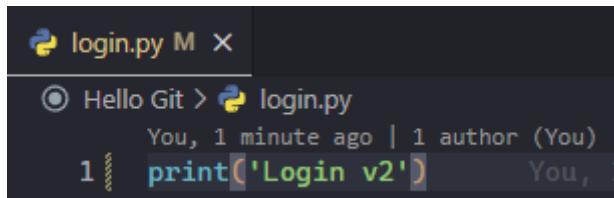
no changes added to commit (use "git add" and/or "git commit -a")
pwsh ~\Desktop\Hello Git > git main into login x1 | x1
git add hellogit3.py
pwsh ~\Desktop\Hello Git > git main into login | ~1
git commit -m "Corrección conflicto"
[login 4fa9d4a] Corrección conflicto

```

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:



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:

```

pwsh ~\Desktop\Hello Git > git login
git switch main
error: Your local changes to the following files would be overwritten by checkout:
  login.py
Please commit your changes or stash them before you switch branches.
Aborting

```

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

```

pwsh ~\Desktop\Hello Git > git login ~1
git stash
Saved working directory and index state WIP on login: 4fa9d4a Corrección conflicto

```

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
```

```
pwsh ~\Desktop\Hello Git > login
> git stash list
stash@{0}: WIP on login: 4fa9d4a Corrección conflicto
```

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

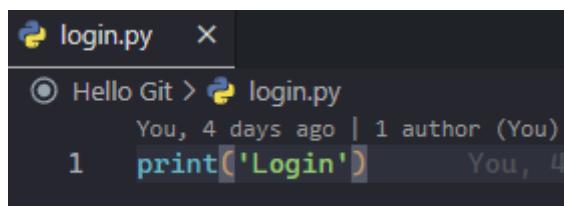
Y ahora podremos cambiar de rama sin problemas:

```
pwsh ~\Desktop\Hello Git > login
> git switch main
Switched to branch 'main'
```

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

```
pwsh ~\Desktop\Hello Git > main
> git switch login
Switched to branch 'login'
```

Si ahora nos vamos al proyecto veremos lo siguiente:



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.

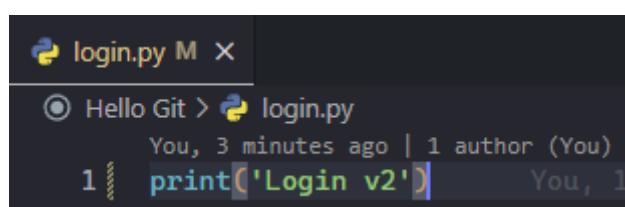
Para recuperar los cambios temporales debemos usar el comando:

```
git stash pop
```

```
pwsh ~\Desktop\Hello Git > login
> git stash pop
On branch login
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   login.py

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (558039de3917d513a69c21f913676514c57ab9ec)
```

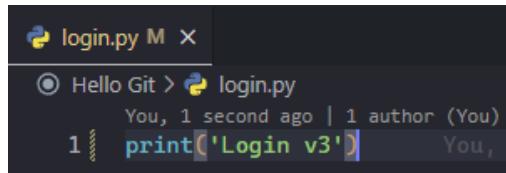
Ahora en el fichero veremos que sí que tenemos los cambios realizados y almacenados en stash:



Vamos a hacer commit de estos cambios:

```
pwsh ~\Desktop\Hello Git > p login ~1
> git add .
pwsh ~\Desktop\Hello Git > p login ✘ ~1
> git commit -m "Login v2"
[login 8ff7bef] Login v2
1 file changed, 1 insertion(+), 1 deletion(-)
```

Seguimos haciendo cambios en nuestro fichero de login:



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

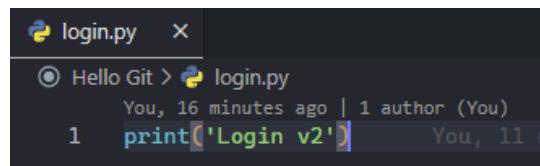
```
pwsh ~\Desktop\Hello Git > p login
> git stash
Saved working directory and index state WIP on login: 8ff7bef Login v2
pwsh ~\Desktop\Hello Git > p login
> git stash list
stash@{0}: WIP on login: 8ff7bef Login v2
pwsh ~\Desktop\Hello Git > p login
> git switch main
Switched to branch 'main'
```

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
```

```
pwsh ~\Desktop\Hello Git > p login
> git stash drop
Dropped refs/stash@{0} (aa284bf7b92e32d9e9a8604446e60d5c82aa77ee)
```

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



REINTEGRACIÓN DE RAMAS⁴

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:

```
pwsh ~\Desktop\Hello Git ➜ login
└─> git switch main
      Switched to branch 'main'
```

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:

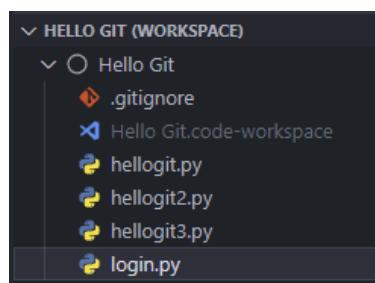
```
git diff nombreRama
```

```
pwsh ~\Desktop\Hello Git ➜ main
└─> git diff login
      diff --git a/login.py b/login.py
      deleted file mode 100644
      index e9ee38f..0000000
      --- a/login.py
      +++ /dev/null
      @@ -1 +0,0 @@
      -print('Login v2')
      \ No newline at end of file
```

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

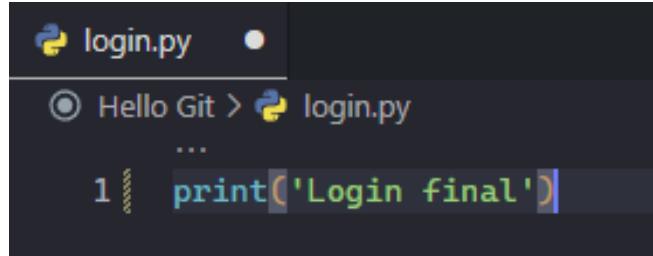
```
pwsh ~\Desktop\Hello Git ➜ main
└─> git merge login
      Updating 33a6e0a..8ff7bef
      Fast-forward
      login.py | 1 +
      1 file changed, 1 insertion(+)
      create mode 100644 login.py
```

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



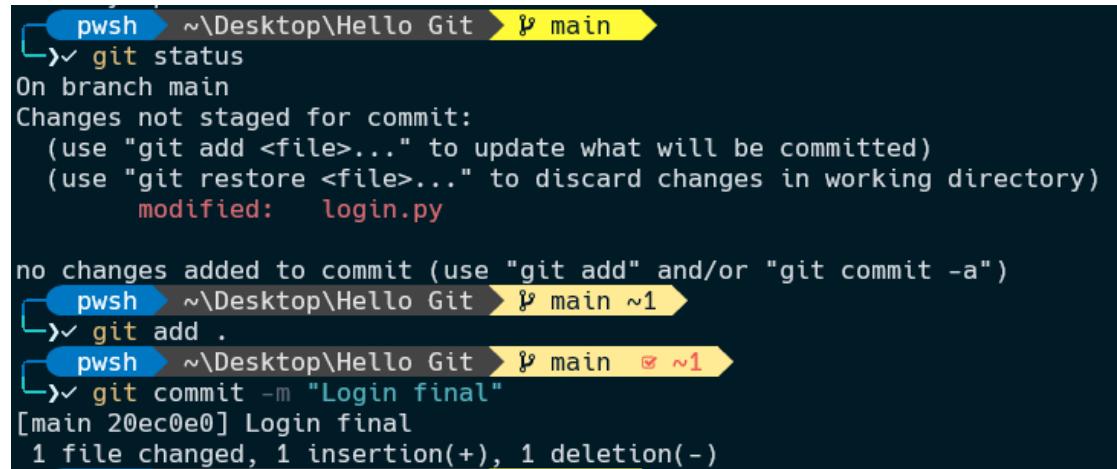
⁴ En realidad, y como veremos más adelante cuando veamos GitHub esta no es la forma real en la que se hacen estas acciones, pero de momento, vamos a hacerlo así ya que se supone que estamos trabajando con un repositorio completamente local.

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



```
login.py
Hello Git > login.py
...
1| print('Login final')
```

Y le podemos hacer commit:



```
git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   login.py

no changes added to commit (use "git add" and/or "git commit -a")
git add .
git commit -m "Login final"
[main 20ec0e0] Login final
 1 file changed, 1 insertion(+), 1 deletion(-)
```

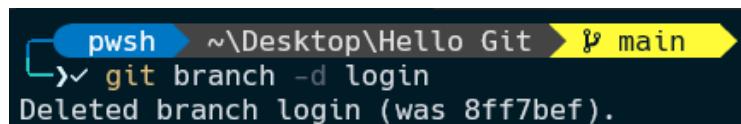
ELIMINACIÓN DE RAMAS EN GIT

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.

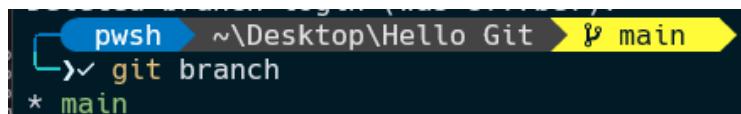
Para eliminar ramas debemos utilizar el comando:

```
git branch -d nombreRama
```



```
git branch -d login
Deleted branch login (was 8ff7bef).
```

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



```
git branch
* main
```

Si que va a seguir referenciada en el reflog:

```
pwsh ~\Desktop\Hello Git p main
└─> git reflog
20ec0e0 (HEAD -> main) HEAD@{0}: commit: Login final
8ff7bef HEAD@{1}: merge login: Fast-forward
33a6e0a HEAD@{2}: checkout: moving from login to main
8ff7bef HEAD@{3}: checkout: moving from main to login
33a6e0a HEAD@{4}: checkout: moving from login to main
8ff7bef HEAD@{5}: reset: moving to HEAD
8ff7bef HEAD@{6}: commit: Login v2
4fa9d4a HEAD@{7}: checkout: moving from main to login
33a6e0a HEAD@{8}: checkout: moving from login to main
4fa9d4a HEAD@{9}: reset: moving to HEAD
4fa9d4a HEAD@{10}: commit (merge): Corrección conflicto
b2db4cf HEAD@{11}: checkout: moving from main to login
33a6e0a HEAD@{12}: commit: Git 3 v3
1aeed13 HEAD@{13}: checkout: moving from login to main
b2db4cf HEAD@{14}: commit: Git 3 login
38a836d HEAD@{15}: merge main: Merge made by the 'ort' strategy.
bc3e1ac HEAD@{16}: checkout: moving from main to login
1aeed13 HEAD@{17}: commit: Git 3 v2
b8b3068 HEAD@{18}: checkout: moving from login to main
bc3e1ac HEAD@{19}: commit: Login
```

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

```
pwsh ~\Desktop\Hello Git p main
└─> git checkout 38a836d
Note: switching to '38a836d'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

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

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 38a836d Merge branch 'main' into login
```

Ahora en un git tree veremos que la rama existe:

```
pwsh ~\Desktop\Hello Git p detached at ~38a836d
└─> git tree
* 20ec0e0 (main) Login final
* 8ff7bef Login v2
* 4fa9d4a Corrección conflicto
| \
| * 33a6e0a Git 3 v3
| * b2db4cf Git 3 login
| * 38a836d (HEAD) Merge branch 'main' into login
| \
| * 1aeed13 Git 3 v2
| * bc3e1ac Login
| /
* b8b3068 Este es mi quinto commit
* 9ba6c52 (tag: primeros_pasos) Se añade el .gitignore
* a73640b Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

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:

```
pwsh ~\Desktop\Hello Git p detached at ~38a836d
└─> git checkout main
Previous HEAD position was 38a836d Merge branch 'main' into login
Switched to branch 'main'
```

GITHUB

A. ¿QUÉ ES GITHUB?

GitHub 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.

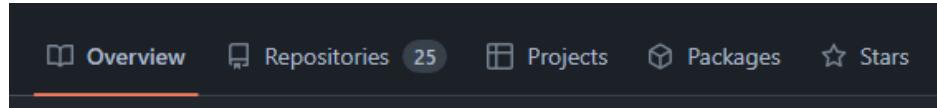
La función de “solicitudes de incorporación de cambios” (*pull requests*) es fundamental en GitHub. Permite a los desarrolladores proponer cambios en un repositorio y facilita el proceso de revisión y fusión de esos cambios de manera organizada y controlada.

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.

B. PRIMEROS PASOS EN GITHUB. CONOCIENDO LA INTERFAZ

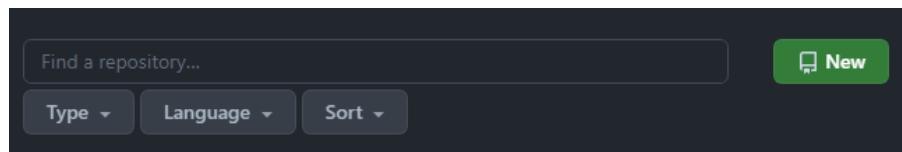
En nuestro home vamos a tener los siguientes apartados:



- **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.

C. 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:



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

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (*).

Owner *	Repository name *
 BertoMP	/ <input type="text"/>

Great repository names are short and memorable. Need inspiration? How about [turbo-meme](#) ?

Description (optional)

 **Public**
Anyone on the internet can see this repository. You choose who can commit.

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

Initialize this repository with:

Add a README file
This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: **None**

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: **None**

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

 You are creating a public repository in your personal account.

Create repository

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:

Required fields are marked with an asterisk (*).

Owner *	Repository name *
 BertoMP	/ <input type="text"/>

Great repository names are short and memorable. Need inspiration? How about [turbo-meme](#) ?

Description (optional)

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):

Owner *	Repository name *
 BertoMP	/ <input type="text" value="BertoMP"/>

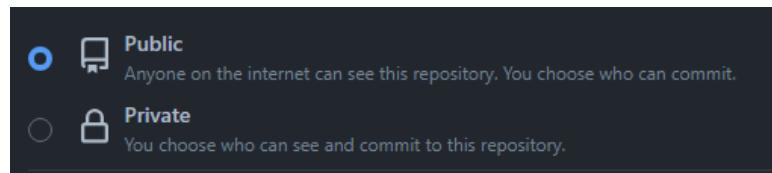
 BertoMP is available.

 BertoMP/BertoMP is a   special  repository that you can use to add a README.md to your GitHub profile. Make sure it's public and initialize it with a README to get started.

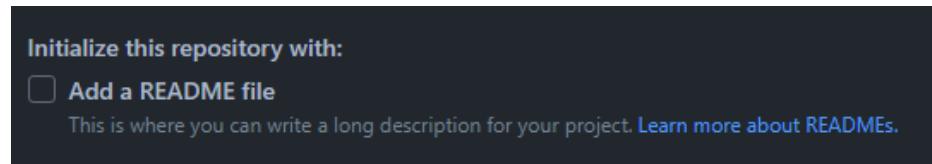
En este caso utilizaremos un repositorio con el nombre HelloGit.

The screenshot shows the GitHub repository creation interface. The 'Owner' field is set to 'BertoMP'. The 'Repository name' field contains 'hello-git', which is highlighted in green with the message 'hello-git is available.' Below the fields, there's a note suggesting repository names should be short and memorable, with a link to 'fantastic-tribble'.

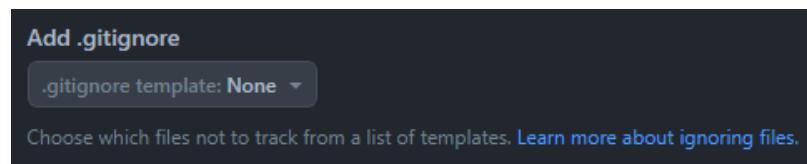
En la siguiente parte del formulario podremos optar por un repositorio público (accesible por todo el mundo) o privado (accesible sólo por nosotros y por personas a las que le demos permiso):



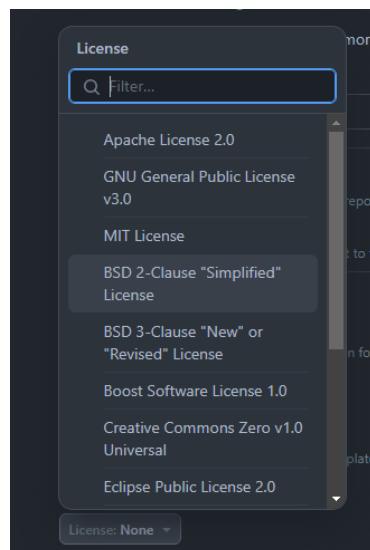
También podemos elegir si queremos crear un archivo README o no, este archivo nos va a servir para hacer un resumen de lo que hace la aplicación que hemos subido al repositorio:



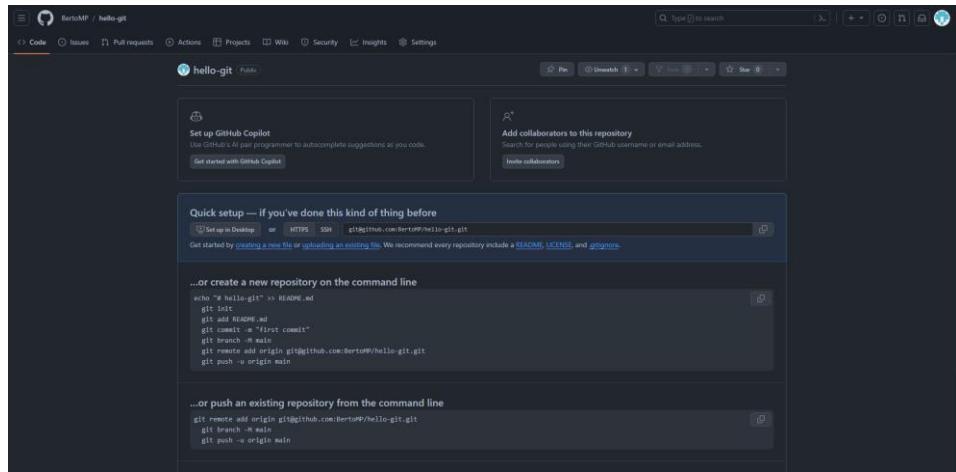
Además podemos decidir si queremos añadir o no el archivo .gitignore:



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



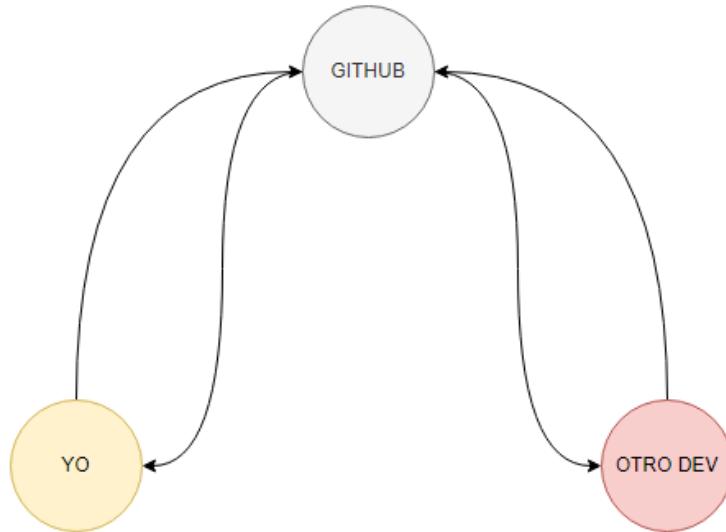
Terminamos pulsando en el botón “Create repository” y ya habremos creado nuestro repositorio:



D. 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:



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 remoto⁵:

```
...or create a new repository on the command line  
echo "# hello-git" >> README.md  
git init  
git add README.md  
git commit -m "first commit"  
git branch -M main  
git remote add origin https://github.com/BertoMP/hello-git.git  
git push -u origin main
```

Vamos a lanzar este comando:

```
pwsh ~\Desktop\Hello Git p main  
git remote add origin https://github.com/BertoMP/hello-git.git
```

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
```

```
pwsh ~\Desktop\Hello Git p main  
git push -u origin main  
Enumerating objects: 37, done.  
Counting objects: 100% (37/37), done.  
Delta compression using up to 12 threads  
Compressing objects: 100% (25/25), done.  
Writing objects: 100% (37/37), 3.19 KiB | 543.00 KiB/s, done.  
Total 37 (delta 11), reused 0 (delta 0), pack-reused 0  
remote: Resolving deltas: 100% (11/11), done.  
To https://github.com/BertoMP/hello-git.git  
 * [new branch] main -> main  
branch 'main' set up to track 'origin/main'.
```

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](#) para más información sobre cómo hacer este proceso de autenticación.

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

The screenshot shows a GitHub repository named "hello-git" with 13 commits. The commits are as follows:

- BertoMP Login final ... (1 hour ago)
- .gitignore (4 days ago)
- hellogit.py (4 days ago)
- hellogit2.py (4 days ago)
- hellogit3.py (3 hours ago)
- login.py (1 hour ago)

Below the commits, there is a message: "Help people interested in this repository understand your project by adding a README." followed by a green button labeled "Add a README".

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”

The screenshot shows the GitHub interface for editing a README file. The file content is "# hello-git". There are buttons for "Edit" and "Preview" at the top, and settings for "Spaces" (set to 2), "2", and "No wrap". At the top right, there are "Cancel changes" and "Commit changes..." buttons.

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

The screenshot shows a modal window titled "Commit changes". It contains fields for "Commit message" (with "Create README.md" typed in) and "Extended description" (with placeholder text "Add an optional extended description.."). At the bottom, there are two radio button options: " Commit directly to the main branch" and " Create a new branch for this commit and start a pull request". Below these options is a link "Learn more about pull requests". At the bottom right are "Cancel" and "Commit changes" buttons.

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:

The screenshot shows a GitHub repository interface. At the top, there's a navigation bar with links for Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. Below the navigation bar, the repository name 'BertoMP / hello-git' is displayed, along with a search bar and a 'Code' dropdown set to 'main'. A 'hello-git' folder icon is also present. To the right of the search bar are buttons for 'Go to file', 'Add file', and more. A 'History' button is located at the top right of the main content area. The main content area displays a table of commits:

Name	Last commit message	Last commit date
.gitignore	Se añade el .gitignore	4 days ago
README.md	Create README.md	now
hellogit.py	Se actualiza el texto del print	4 days ago
hellogit2.py	Este es mi segundo commit	4 days ago
hellogit3.py	Git 3 v3	3 hours ago
login.py	Login final	1 hour ago

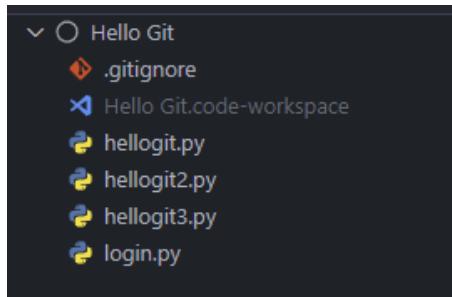
Below the commit table, there's a section for 'README.md' with a edit icon. At the bottom of the page, there's a large 'hello-git' heading.

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:

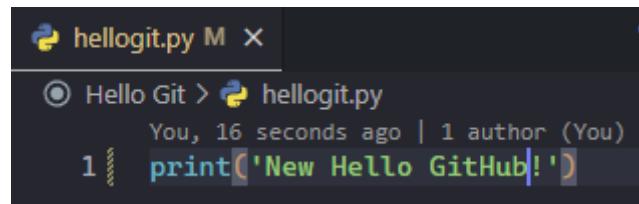
The screenshot shows the 'Commits' page for the 'main' branch of the 'hello-git' repository. The page title is 'Commits' and it shows a dropdown menu for the branch 'main'. Below the dropdown, it says 'Commits on Nov 24, 2023'. The main content area lists several commits:

- Create README.md by BertoMP committed 2 minutes ago. Status: Verified. Hash: 1a1beae.
- Login final by BertoMP committed 1 hour ago. Hash: 20ec0e0.
- Login v2 by BertoMP committed 2 hours ago. Hash: 8ff7bef.
- Corrección conflicto by BertoMP committed 3 hours ago. Hash: 4fa9d4a.
- Git 3 v3 by BertoMP committed 3 hours ago. Hash: 33a6e0a.
- Git 3 login by BertoMP committed 3 hours ago. Hash: b2db4cf.
- Merge branch 'main' into login by BertoMP committed 3 hours ago. Hash: 38a836d.

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



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



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

```
pwsh ~\Desktop\Hello Git p main
> git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   hellogit.py

no changes added to commit (use "git add" and/or "git commit -a")
pwsh ~\Desktop\Hello Git p main ~1
> git add .
pwsh ~\Desktop\Hello Git p main ~1
> git commit -m "Hello GitHub"
[main a48dc36] Hello GitHub
  1 file changed, 1 insertion(+), 1 deletion(-)
```

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



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
```

```
pwsh ~\Desktop\Hello Git p main
> git push
To https://github.com/BertoMP/hello-git.git
! [rejected]      main -> main (fetch first)
error: failed to push some refs to 'https://github.com/BertoMP/hello-git.git'
hint: Updates were rejected because the remote contains work that you do not
hint: have locally. This is usually caused by another repository pushing to
hint: the same ref. If you want to integrate the remote changes, use
hint: 'git pull' before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

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
```

```
pwsh ~\Desktop\Hello Git p main
> git fetch
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 678 bytes | 39.00 KiB/s, done.
From https://github.com/BertoMP/hello-git
  20ec0e0..1a1beae main      -> origin/main
```

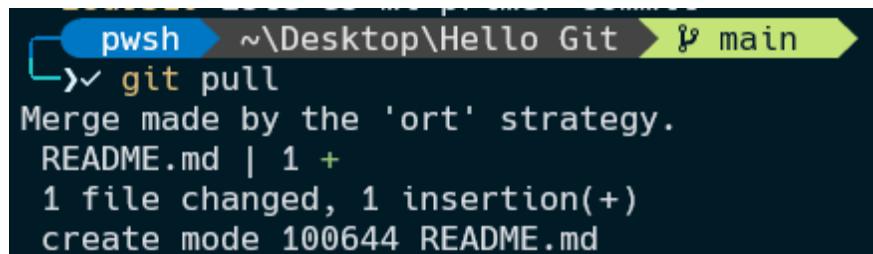
Si ahora hacemos un tree veremos esto:

```
pwsh ~\Desktop\Hello Git p main
> git tree
* a48dc36 (HEAD -> main) Hello GitHub
| * 1a1beae (origin/main) Create README.md
|/
* 20ec0e0 Login final
* 8ff7bef Login v2
* 4fa9d4a Corrección conflicto
| \
| * 33a6e0a Git 3 v3
| * b2db4cf Git 3 login
| * 38a836d Merge branch 'main' into login
| \
| * 1aeed13 Git 3 v2
| * bc3e1ac Login
|/
* b8b3068 Este es mi quinto commit
* 9ba6c52 (tag: primeros_pasos) Se añade el .gitignore
* a73640b Se actualiza el texto del print
* 52d856f Este es mi segundo commit
* 28d9310 Este es mi primer commit
```

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.

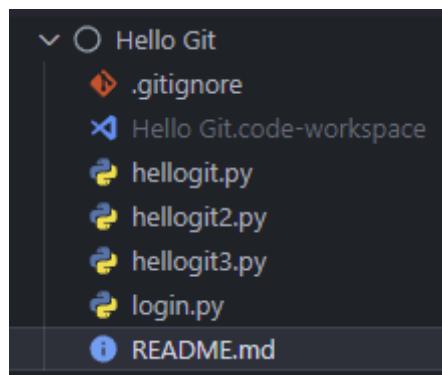
Para descargar estos cambios utilizamos el comando:

```
git pull
```

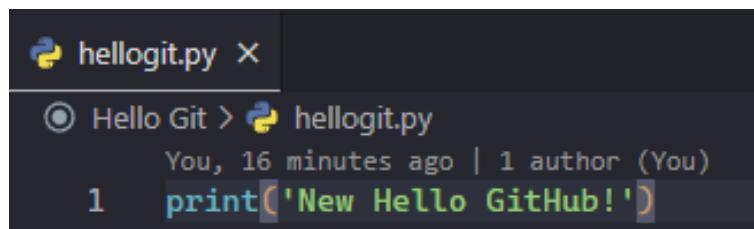


```
pwsh ~\Desktop\Hello Git main
> git pull
Merge made by the 'ort' strategy.
 README.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

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



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



```
hellogit.py X
Hello Git > hellogit.py
You, 16 minutes ago | 1 author (You)
1 print('New Hello GitHub!')
```

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:

```
pwsh ~\Desktop\Hello Git ➜ main
└─> git status
On branch main
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

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

```
nothing to commit, working tree clean
pwsh ~\Desktop\Hello Git ➜ main
└─> git log
commit fc6b76a0389460c40c95dd28016cf882faeda3b (HEAD -> main)
Merge: a48dc36 1a1beae
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date:   Fri Nov 24 22:36:58 2023 +0100

  Merge branch 'main' of https://github.com/BertoMP/hello-git

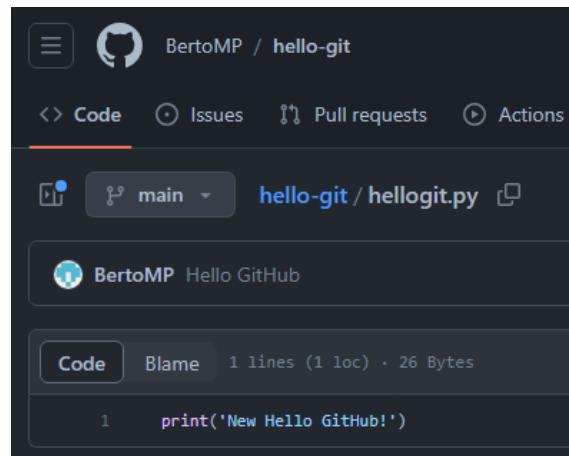
commit a48dc367a8bc1efbc0543895eb21a69fc901b715
Author: Alberto Martínez Pérez <alberto.martinezperez@hotmail.es>
Date:   Fri Nov 24 22:22:44 2023 +0100

  Hello GitHub
```

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:

```
pwsh ~\Desktop\Hello Git ➜ main
└─> git push
Enumerating objects: 9, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 12 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 571 bytes | 571.00 KiB/s, done.
Total 5 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/BertoMP/hello-git.git
  1a1beae..fc6b76a  main -> main
```

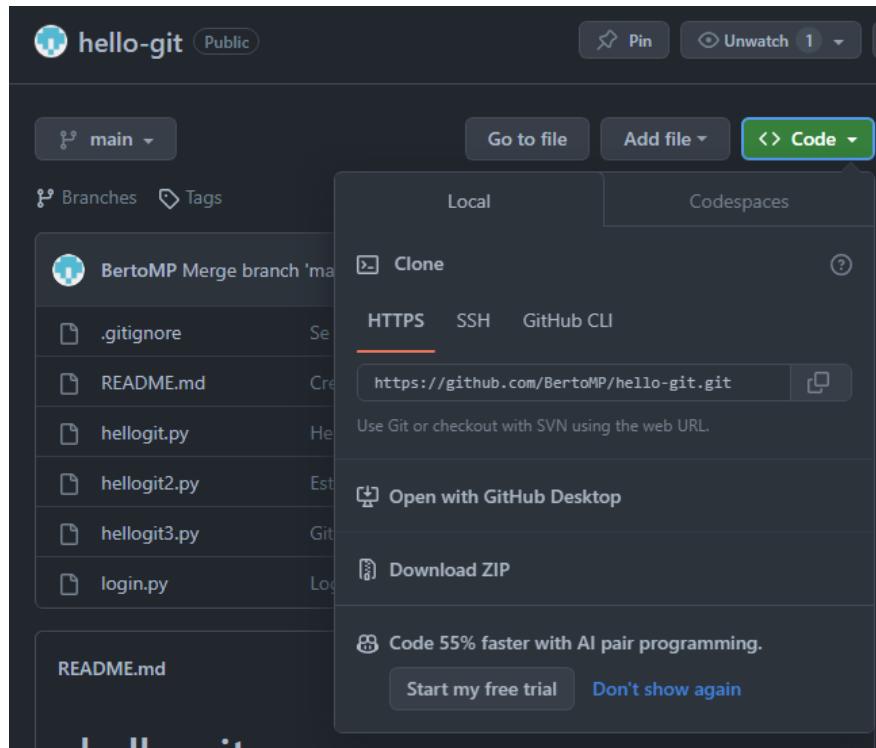
En este caso el push ha sido correcto y hemos podido subir nuestros cambios al repositorio remoto. De hecho, para comprobar esto podemos ver el estado en GitHub del fichero hellogit.py:



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 → Download ZIP:



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



Nos movemos a este directorio y utilizamos el comando:

```
git clone URLRepositorioRemoto
```

```
pwsh ~\Desktop
> cd '..\Mi clonacion\' 
pwsh ~\Desktop\Mi clonacion
> git clone https://github.com/BertoMP/hello-git.git
Cloning into 'hello-git'...
remote: Enumerating objects: 45, done.
remote: Counting objects: 100% (45/45), done.
remote: Compressing objects: 100% (18/18), done.
remote: Total 45 (delta 15), reused 41 (delta 13), pack-reused 0
Receiving objects: 100% (45/45), 4.28 KiB | 2.14 MiB/s, done.
Resolving deltas: 100% (15/15), done.
```

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

```
pwsh ~\Desktop\Mi clonacion
└─╼ ll

Directory: C:\Users\alber\Desktop\Mi clonacion

Mode                LastWriteTime      Length Name
----                -----          ----- 
d---        24/11/2023    23:06            hello-git
```

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

```
pwsh ~\Desktop\Mi clonacion
└─╼ ll .\hello-git\

Directory: C:\Users\alber\Desktop\Mi clonacion\hello-git

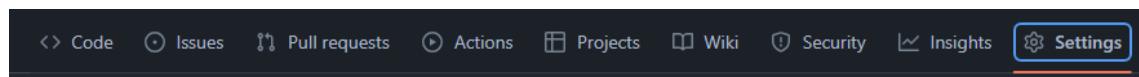
Mode                LastWriteTime      Length Name
----                -----          ----- 
-a---    24/11/2023    23:06           26 ◊ .gitignore
-a---    24/11/2023    23:06           26 ⚡ hellogit.py
-a---    24/11/2023    23:06           21 ⚡ hellogit2.py
-a---    24/11/2023    23:06           24 ⚡ hellogit3.py
-a---    24/11/2023    23:06           20 ⚡ login.py
-a---    24/11/2023    23:06           13 📄 README.md
```

PROTEGIENDO NUESTRO PROYECTO FRENTE A CAMBIOS

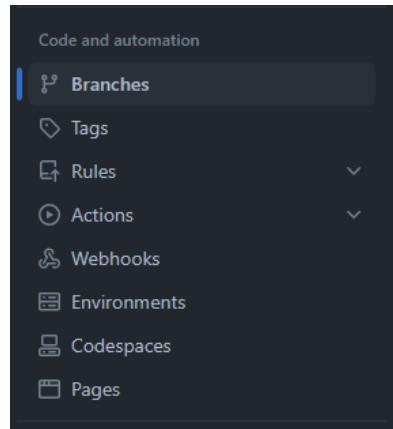
Estamos trabajando con un repositorio remoto público y eso, obviamente, implica que cualquier persona puede ver el repositorio y puede descargárselo. El problema que tenemos ahora mismo es que cualquier persona puede hacer cambios sobre nuestro repositorio porque no hemos generado ninguna regla que lo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

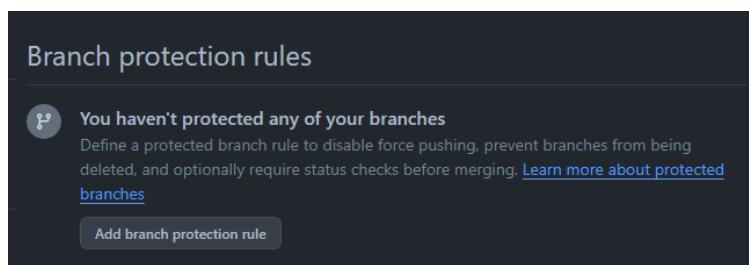
Para proteger nuestra rama principal debemos ir al apartado de configuración de nuestro repositorio:



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



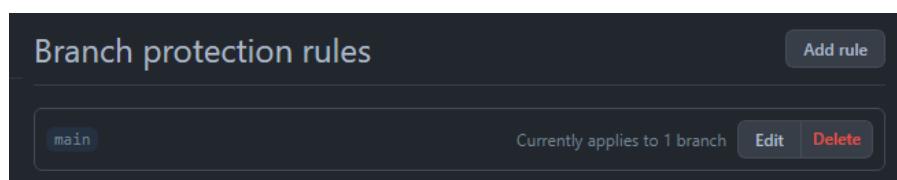
Una vez aquí veremos la siguiente sección de la web:



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:

- 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.
- Requiere approvals:* Relacionado con lo anterior las PR necesitarán de una aprobación para que se puedan unir a la rama afectada.
- 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:



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:

```
C:\> Users > alber > Desktop > Mi clonacion > hello-git > README.md > # cambios en el archivo README.
1 # hello-git
2
3 # cambios en el archivo README.

pwsh ~\Desktop\Mi clonacion\hello-git > main
git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
pwsh ~\Desktop\Mi clonacion\hello-git > main ~1
git add .
pwsh ~\Desktop\Mi clonacion\hello-git > main ~1
git commit -m "Cambios en el README"
[main 0eeb716] Cambios en el README
  1 file changed, 2 insertions(+)
```

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

```
pwsh ~\Desktop\Mi clonacion\hello-git > main
git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 307 bytes | 307.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote: error: GH006: Protected branch update failed for refs/heads/main.
remote: error: Changes must be made through a pull request.
To https://github.com/BertoMP/hello-git.git
 ! [remote rejected] main -> main (protected branch hook declined)
error: failed to push some refs to 'https://github.com/BertoMP/hello-git.git'
```

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.



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

The screenshot shows a GitHub repository page for 'hello-git'. At the top, there's a navigation bar with links for Code, Issues, Pull requests, Actions, Projects, Security, and Insights. Below the navigation bar, the repository name 'hello-git' is displayed, along with its status as 'Public'. There are buttons for Watch, Fork (which shows 0 forks), and Star (which shows 0 stars). On the left, there are tabs for 'main' (selected), 'Branches', and 'Tags'. The main content area shows a list of commits, including a merge from 'BertoMP' and several commits from the user. To the right of the commit list, there's an 'About' section with a note: 'No description, website, or topics provided.' It also lists 'Readme', 'Activity', '0 stars', '1 watching', '0 forks', and a 'Report repository' link. Below that is a 'Releases' section stating 'No releases published'. Under 'Packages', it says 'No packages published'. The 'Languages' section shows 'Python 100.0%'. At the bottom left, there's a preview of the 'README.md' file which contains the text 'hello-git'.

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



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:

The screenshot shows the 'Create a new fork' form on GitHub. The title is 'Create a new fork'. A note below it says: 'A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.' It also states: 'Required fields are marked with an asterisk (*).'

The form fields include:

- Owner ***: A dropdown menu showing 'BertoMarPer'.
- Repository name ***: An input field containing 'hello-git-fork'. A note next to it says: 'hello-git-fork is available.'
- Description (optional)**: A text input field with a placeholder.
- Copy the `main` branch only**: A checked checkbox with a note below it: 'Contribute back to BertoMP/hello-git by adding your own branch. [Learn more](#)'.
- Information**: A note: '(i) You are creating a fork in your personal account.'
- Create fork**: A green button at the bottom right.

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

 hello-git-fork Public

forked from [BertoMP/hello-git](#)

[Pin](#) [Watch 0](#) [Fork](#) [Star 0](#)

[main](#) [Go to file](#) [Add file](#) [Code](#)

[Branches](#) [Tags](#)

This branch is up to date with BertoMP/hello-git:main. [Contribute](#) [Sync fork](#)

 BertoMP Merge branch 'main' of <https://github.com/BertoMP/> ... [...](#) 9 hours ago [16](#)

	.gitignore	Se añade el .gitignore
	README.md	Create README.md
	hellogit.py	Hello GitHub
	hellogit2.py	Este es mi segundo commit
	hellogit3.py	Git 3 v3
	login.py	Login final

[README.md](#) 

hello-git

About

No description, website, or topics provided.

[Readme](#)
[Activity](#)
[0 stars](#)
[0 watching](#)
[1 fork](#)

Releases

No releases published
[Create a new release](#)

Packages

No packages published
[Publish your first package](#)

Languages

Python 100.0%

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:

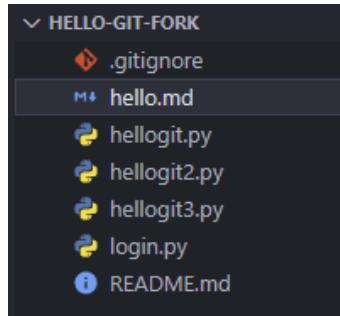
```
pwsh ~\Desktop\Mi clonacion\hello-git main
> cd ..
pwsh ~\Desktop\Mi clonacion
> git clone https://github.com/BertoMarPer/hello-git-fork.git
Cloning into 'hello-git-fork'...
remote: Enumerating objects: 45, done.
remote: Counting objects: 100% (45/45), done.
remote: Compressing objects: 100% (18/18), done.
remote: Total 45 (delta 15), reused 41 (delta 13), pack-reused 0
Receiving objects: 100% (45/45), 4.28 KiB | 2.14 MiB/s, done.
Resolving deltas: 100% (15/15), done.
```

De esta forma tenemos el repositorio del fork:

```
pwsh ~\Desktop\Mi clonacion
> ll .\hello-git-fork\  
  
Directory: C:\Users\alber\Desktop\Mi clonacion\hello-git-fork  
  
Mode LastWriteTime Length Name
---- - - - - -  
-a--- 25/11/2023 8:13 26 ◊ .gitignore  
-a--- 25/11/2023 8:13 26 ⚡ hellogit.py  
-a--- 25/11/2023 8:13 21 ⚡ hellogit2.py  
-a--- 25/11/2023 8:13 24 ⚡ hellogit3.py  
-a--- 25/11/2023 8:13 20 ⚡ login.py  
-a--- 25/11/2023 8:13 13 📄 README.md
```

FLUJO COLABORATIVO EN GITHUB

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



Y vamos a hacer cambio sobre él:

```
diff hello.md U X
*** hello.md > ⚡ # Esto es un archivo .md con el nombre 'hello'
1   # Esto es un archivo .md con el nombre 'hello'
2   |
```

Vamos a hacer el correspondiente commit para guardarlo:

```
pwsh ~\Desktop\Mi clonacion\hello-git-fork ➜ main ?1
└─> git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    hello.md

nothing added to commit but untracked files present (use "git add" to track)
└─> git add .
└─> pwsh ~\Desktop\Mi clonacion\hello-git-fork ➜ main ✘ +1
└─> git commit -m "Se añade un fichero hello.md"
[main de18c72] Se añade un fichero hello.md
 1 file changed, 1 insertion(+)
 create mode 100644 hello.md
```

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:

```
pwsh ~\Desktop\Mi clonacion\hello-git-fork ➜ main
└─> git push
info: please complete authentication in your browser...
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 325 bytes | 325.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/BertoMarPer/hello-git-fork.git
  fc6b76a..de18c72  main -> main
```

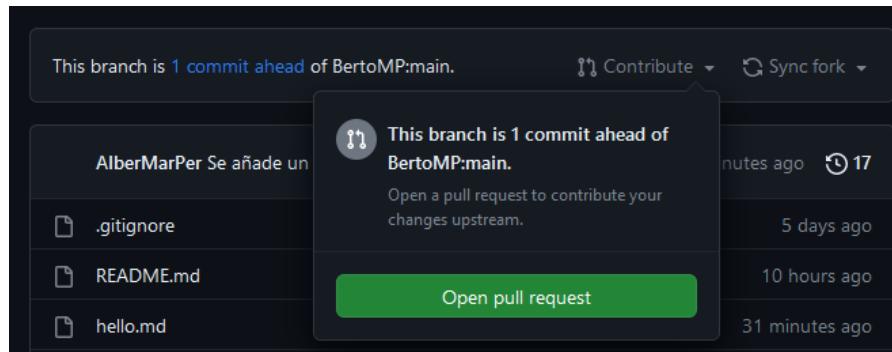
Si ahora vamos a nuestro repositorio remoto veremos que se han almacenado los cambios:

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.

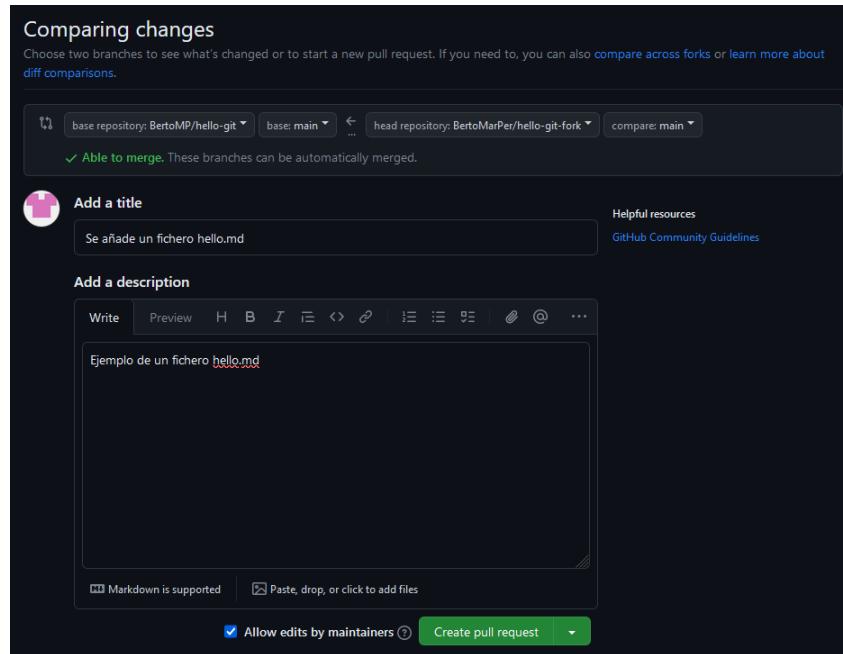


PULL REQUEST EN GITHUB

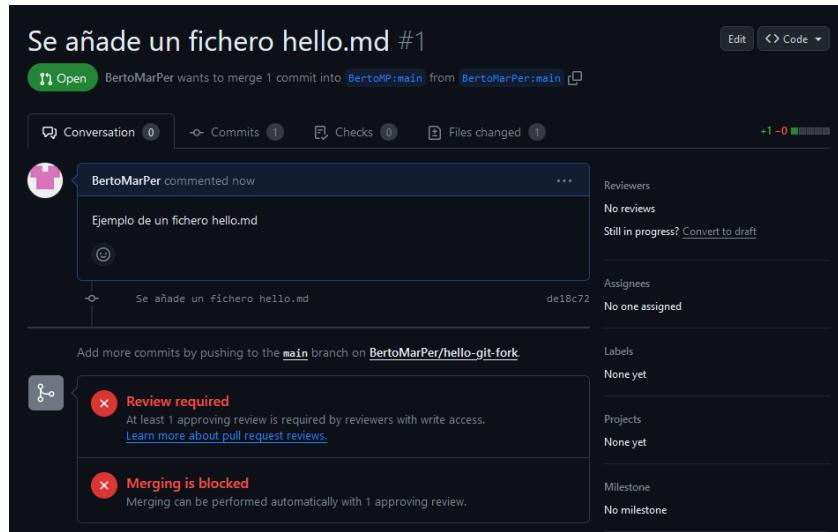
El siguiente paso que debemos hacer es la *pull request* al repositorio original donde vamos a contribuir con los cambios que hemos realizado en el apartado anterior, para ello desde nuestro repositorio secundario vamos a hacer clic en “contribute” y, a continuación, en “Open pull request”:



Esto nos abrirá un formulario:

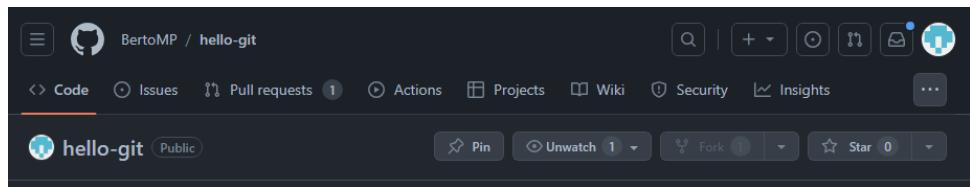


En él podemos añadir un título a la *pull request* y una descripción de la misma. Hacemos clic en “Create pull request” y veremos lo siguiente:

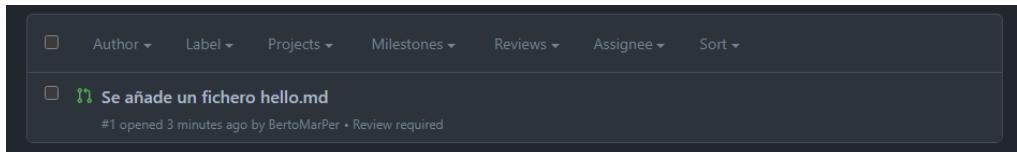


Con esto hemos creado una *pull request* pero esta, por las normas que establecimos en la protección del repositorio, debe ser revisada por el administrador del repositorio original.

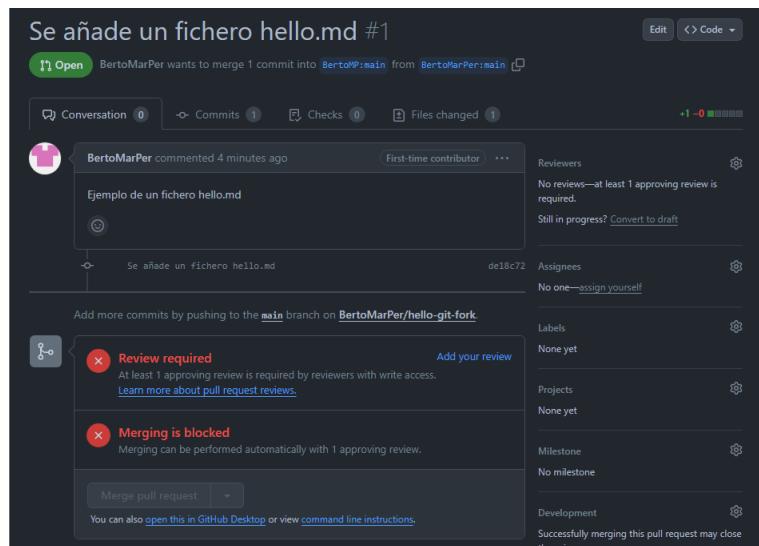
Ahora en el repositorio del administrador aparecerá esto:



Ha recibido una *pull request* sobre el proyecto:



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



The top image shows a GitHub commit history for a repository. It displays a single commit from 'AlberMarPer' dated November 25, 2023, titled 'Se añade un fichero hello.md'. The bottom image shows a detailed view of the 'hello.md' file's content, which contains the text '# Esto es un archivo .md con el nombre 'hello''. Both images illustrate the initial state of the code review process.

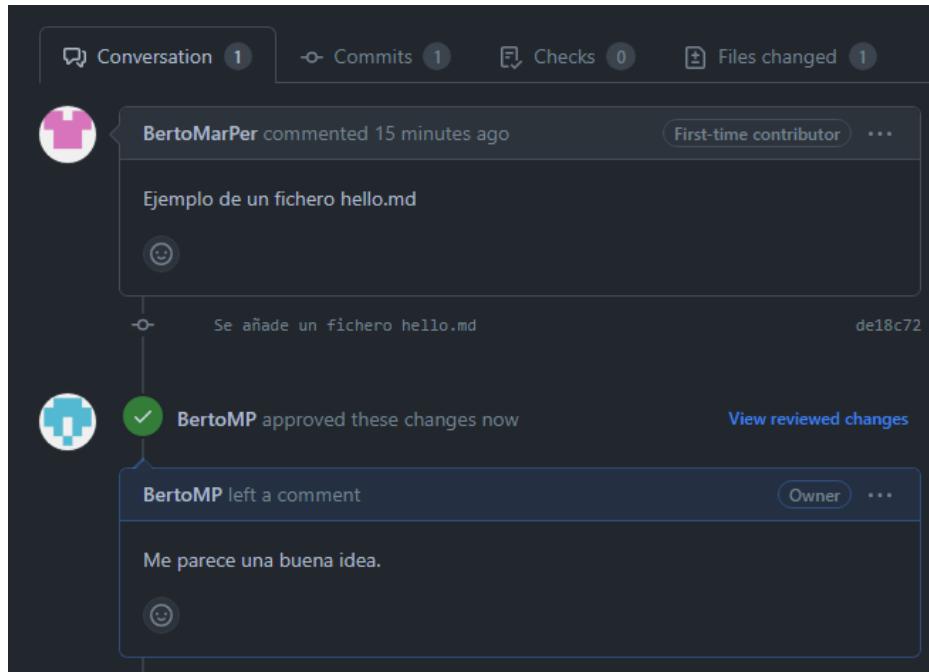
Podemos hacer clic en “review changes” para dar feedback sobre lo que nos parecen los cambios que quiere realizar el colaborador:

This screenshot shows the 'Finish your review' dialog box overlaid on the GitHub interface. The dialog includes a rich text editor for writing comments, a preview section, and a 'Review changes' button. Below the editor, there are three radio button options: 'Comment', 'Approve', and 'Request changes'. The 'Approve' option is selected, indicating the reviewer's intent to merge the changes.

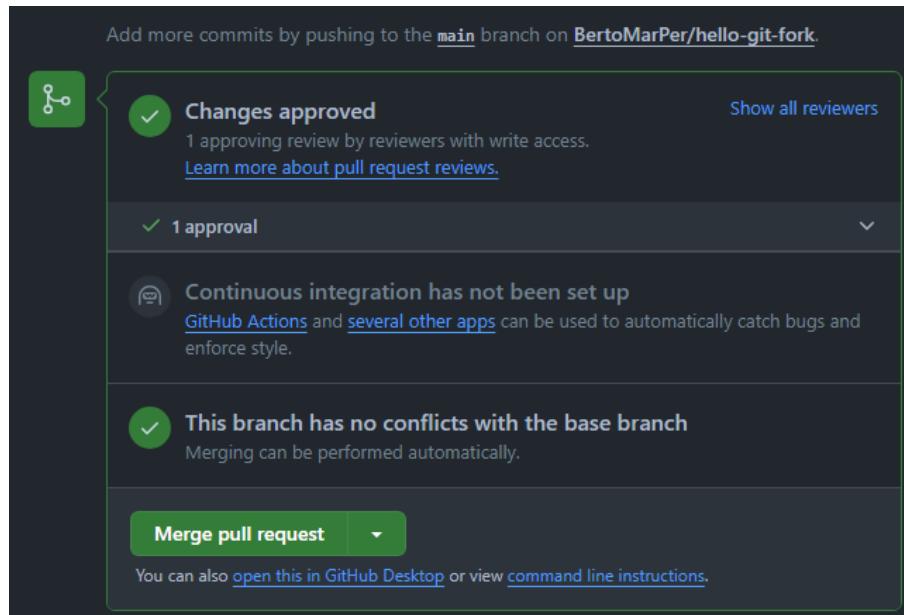
Podemos hacer un comentario (*comment*) para dar feedback simplemente, aprobarlo (*approve*) para añadirlo a nuestro repositorio o solicitar cambios (*request changes*) para decirle al colaborador que si cambia ciertas cosas lo podríamos terminar aceptado. En este caso lo vamos a aprobar:

This screenshot shows the 'Finish your review' dialog with a comment entered: 'Me parece una buena idea.' The 'Approve' radio button is selected. The 'Comment' and 'Request changes' options are also visible below it. The 'Submit review' button is at the bottom right of the dialog.

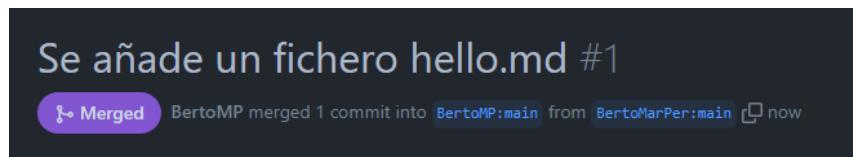
Ahora en la sección de conversación nos aparecerá esto, una primera parte con los comentarios que se han hecho:



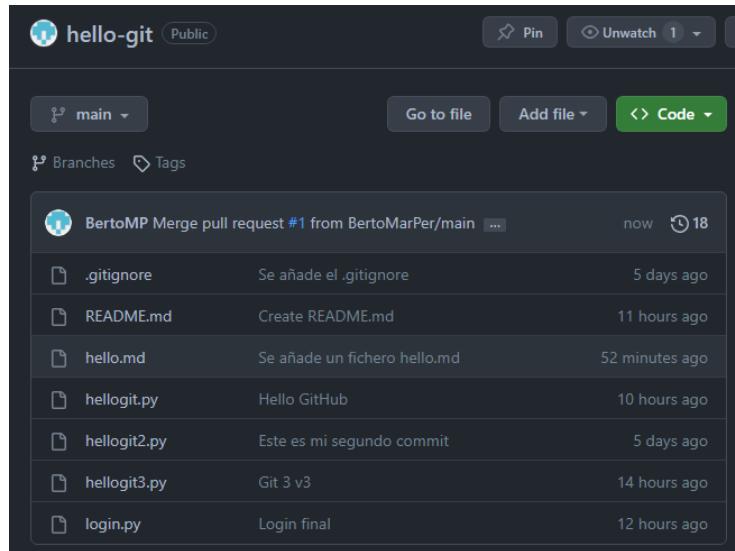
Y una segunda parte donde nos aparece la opción de hacer *merge* de la *pull request*:



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



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



Commit	Message	Time Ago
.gitignore	Se añade el .gitignore	5 days ago
README.md	Create README.md	11 hours ago
hello.md	Se añade un fichero hello.md	52 minutes ago
hellogit.py	Hello GitHub	10 hours ago
hellogit2.py	Este es mi segundo commit	5 days ago
hellogit3.py	Git 3 v3	14 hours ago
login.py	Login final	12 hours ago

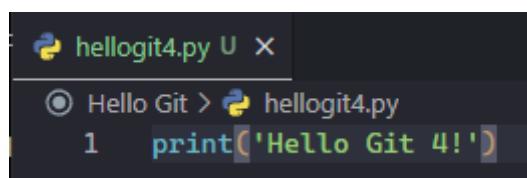
Por último, apuntar que al igual que vimos en los merge locales en el caso de que existan conflictos entre diferentes *pull requests* o entre una de estas y nuestro repositorio remoto, deberemos hacer un estudio de estos conflictos y decidir qué cambios son con los que nos quedaremos al final.

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:

```
pwsh ~\Desktop\Hello Git p main
-> git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (4/4), 951 bytes | 43.00 KiB/s, done.
From https://github.com/BertoMP/hello-git
  fc6b76a..4088f70 main      -> origin/main
Updating fc6b76a..4088f70
Fast-forward
 hello.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 hello.md
```

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



```
hellogit4.py U X
Hello Git > hellogit4.py
1 print('Hello Git 4!')
```

Vamos a hacerle un commit:

```
pwsh ~\Desktop\Hello Git > p main ?1
>✓ git add .
pwsh ~\Desktop\Hello Git > p main ✘ +1
>✓ git commit -m "Hello git 4"
[main 0b5ad4e] Hello git 4
 1 file changed, 1 insertion(+)
 create mode 100644 hellogit4.py
```

Y lo vamos a subir al remoto⁷:

```
pwsh ~\Desktop\Hello Git > p main
>✗ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 302 bytes | 302.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/BertoMP/hello-git.git
 4088f70..0b5ad4e  main -> main
```

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

```
hellogit5.py U X
Hello Git > hellogit5.py
1 print('Hello Git 5!')
```

```
pwsh ~\Desktop\Hello Git > p main
>✓ git add .
pwsh ~\Desktop\Hello Git > p main ✘ +1
>✓ git commit -m "Hello git 5"
[main 902df44] Hello git 5
 1 file changed, 1 insertion(+)
 create mode 100644 hellogit5.py
pwsh ~\Desktop\Hello Git > p main
>✓ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 321 bytes | 321.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/BertoMP/hello-git.git
 0b5ad4e..902df44  main -> main
```

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*.

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:

Commit	Message	Time Ago
.gitignore	Se añade el .gitignore	5 days ago
README.md	Create README.md	11 hours ago
hello.md	Se añade un fichero hello.md	1 hour ago
hellogit.py	Hello GitHub	11 hours ago
hellogit2.py	Este es mi segundo commit	5 days ago
hellogit3.py	Git 3 v3	14 hours ago
hellogit4.py	Hello git 4	8 minutes ago
hellogit5.py	Hello git 5	1 minute ago
login.py	Login final	13 hours ago

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”:

This branch is 3 commits behind BertoMP:main.

This branch is out-of-date

Update branch to keep this branch up-to-date by syncing 3 commits from the upstream repository.

[Learn more about syncing forks](#)

[Compare](#) [Update branch](#)

Y, a continuación, en “update branch”:

Commit	Message	Time Ago
.gitignore	Se añade el .gitignore	5 days ago
README.md	Create README.md	11 hours ago
hello.md	Se añade un fichero hello.md	1 hour ago
hellogit.py	Hello GitHub	11 hours ago
hellogit2.py	Este es mi segundo commit	5 days ago
hellogit3.py	Git 3 v3	14 hours ago
hellogit4.py	Hello git 4	11 minutes ago
hellogit5.py	Hello git 5	3 minutes ago
login.py	Login final	13 hours ago

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.

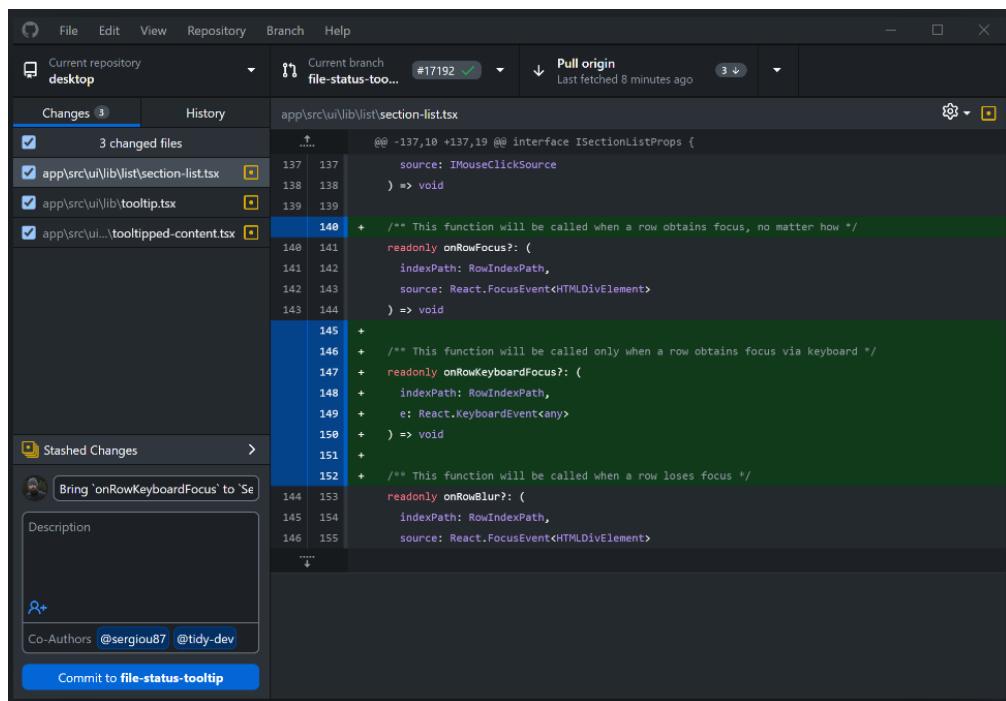
Entre las GUI para Git que tenemos en el mercado vamos a destacar las siguientes:

A. 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.

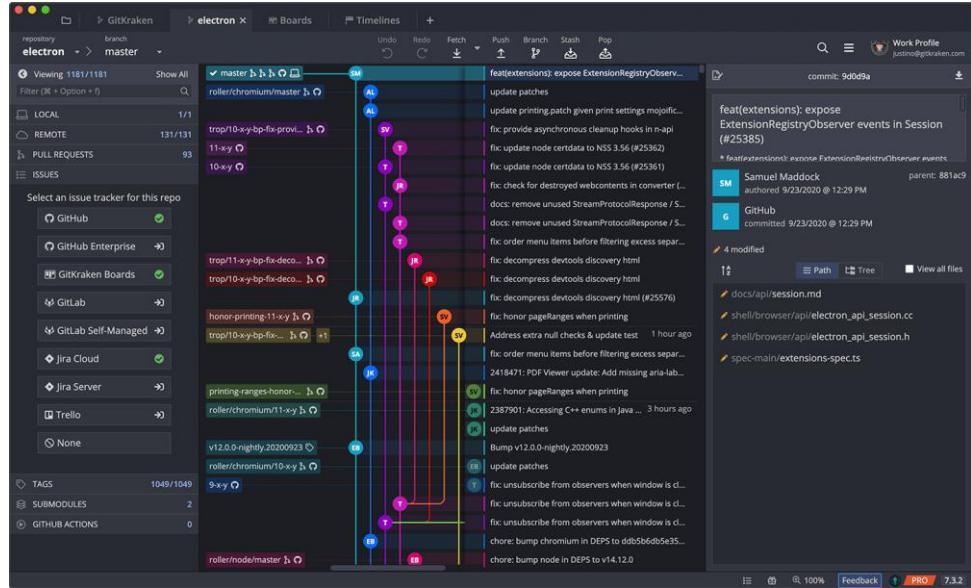


B. GITKRAKEN

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.

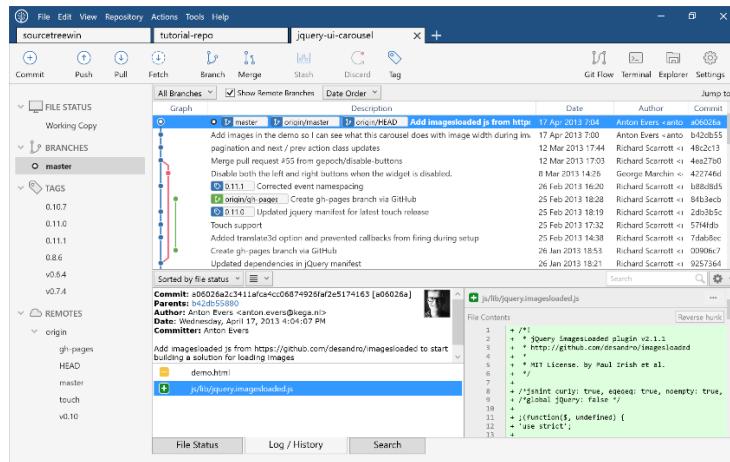


C. SOURCETREE

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.



D. 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.

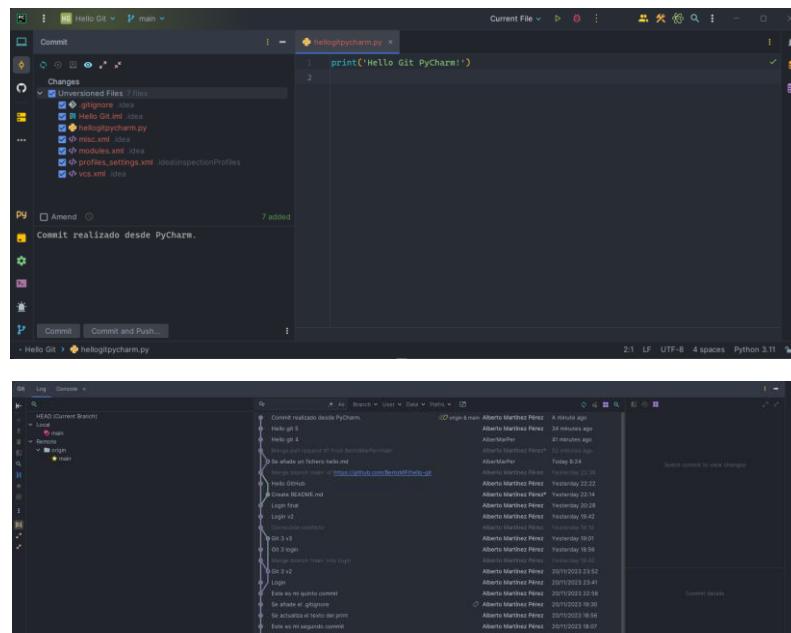
Ofrece integración con GitHub y está disponible en Windows, macOS y Linux.



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:



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.