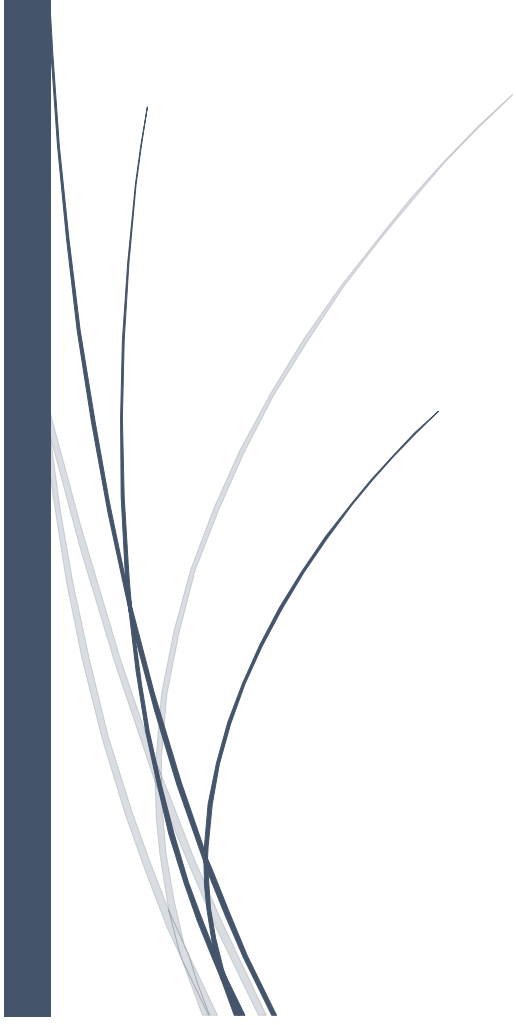
17-7-2023

GIT-GITHUB

Apuntes con lo básico



andres villota camacho

ÍNDICE

ESTE DOCUMENTO

Este documento lo iré realizando a medida que yo aprenda sobre GIT y GITHUB a modo de apuntes, por lo que no es un tutorial y puede tener información errónea.

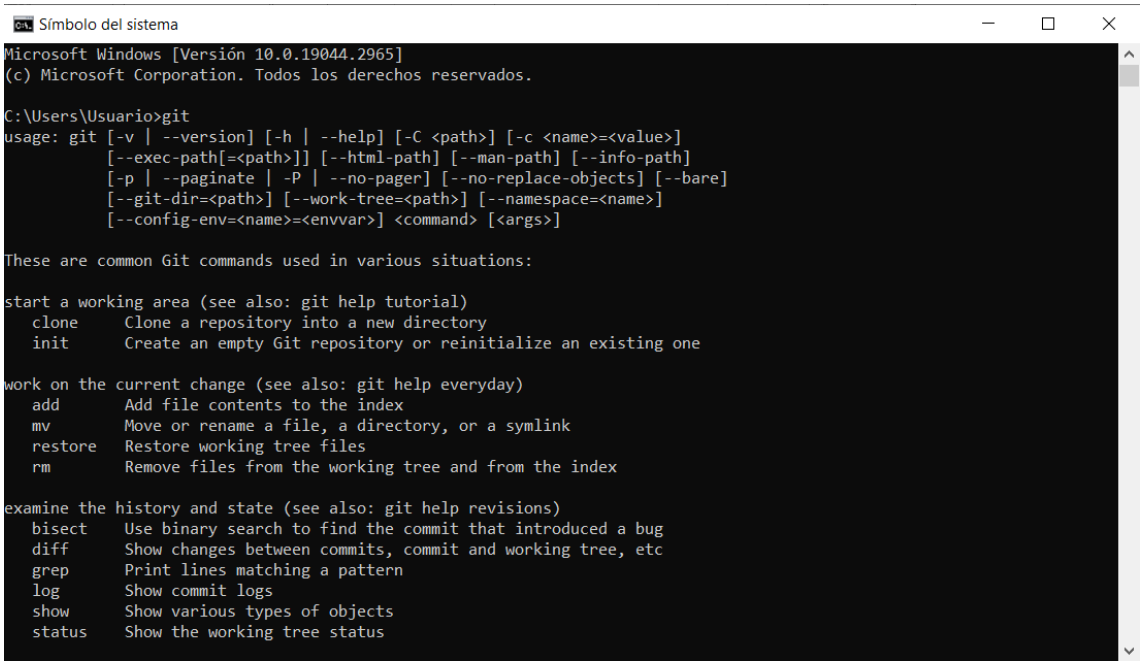
QUE ES GIT

GIT es un sistema de control versiones, creado por **Linus Torvalds** (figura importante del mundo de la programación, creador de Linux).

Es de código abierto, tiene un libro gratis y en español en la misma página web de Git. https://git-scm.com/ Desde esta web podemos **descargar GIT**.

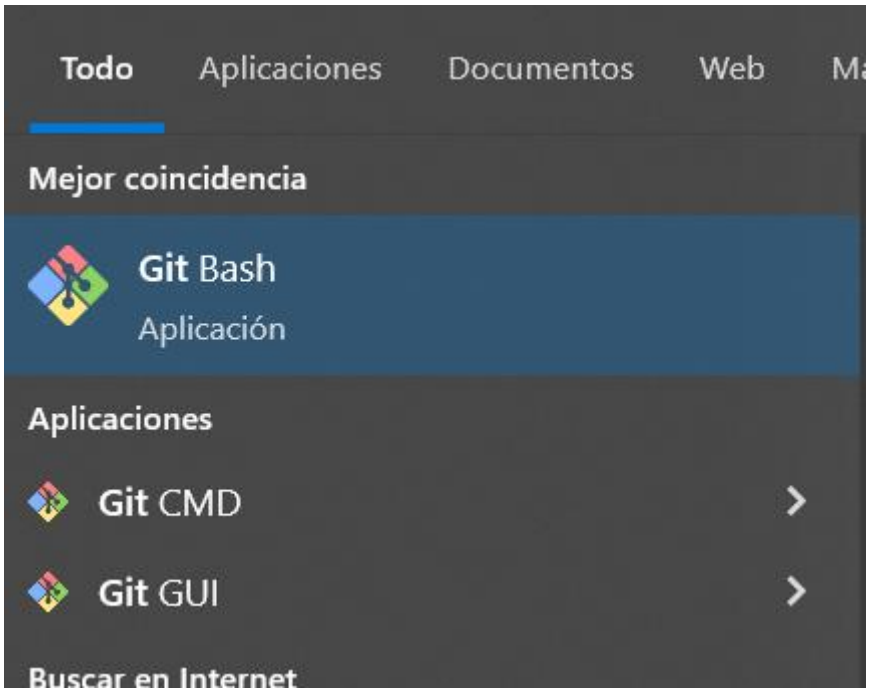
Tiene una versión de comandos y unas versiones gráficas. Para entender mejor el proceso es mejor usar la versión de comandos. Los comandos funcionan igual en Windows, MAC, Linux…

Una vez instalado, ya podemos trabajar con GIT. Si abrimos el CMD y ponemos git saldrá algo como esto.



Esto significa que ya tenemos instalado el software.

Ahora para utilizarlo, según entiendo puedes usarlo desde una ventana CMD, pero como la idea es moverse por el terminal, para Windows tenemos GIT Bash, que es como una terminal Linux para poder movernos por todos los directorios.



Git tiene muchos comandos, pero con unos 10 seremos capaces de realizar el 90% de lo que necesitemos hacer.

Configuración de GIT.

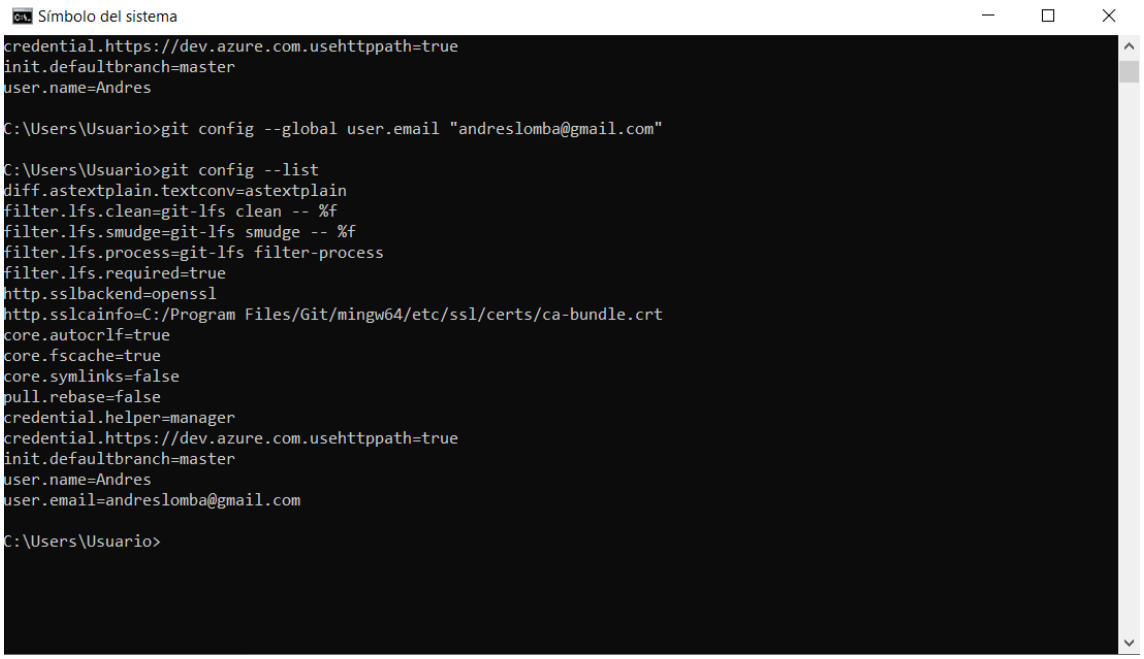
Lo principal e imprescindible que necesitas configurar para poder empezar a usar GIT es un nombre y un email. Para esto iremos a la consola y pondremos los siguientes comandos.

git config –-global user.name “nombreUsuario”

git config –-global user.email “email@email.com”

Si vamos a la carpeta base de Git y de ahí al archivo de configuración. Nos saldrán el nombre de usuario y el email que hemos puesto. También podemos ver si están desde la consola con el comando.

git config --list



INICIALIZAR GIT

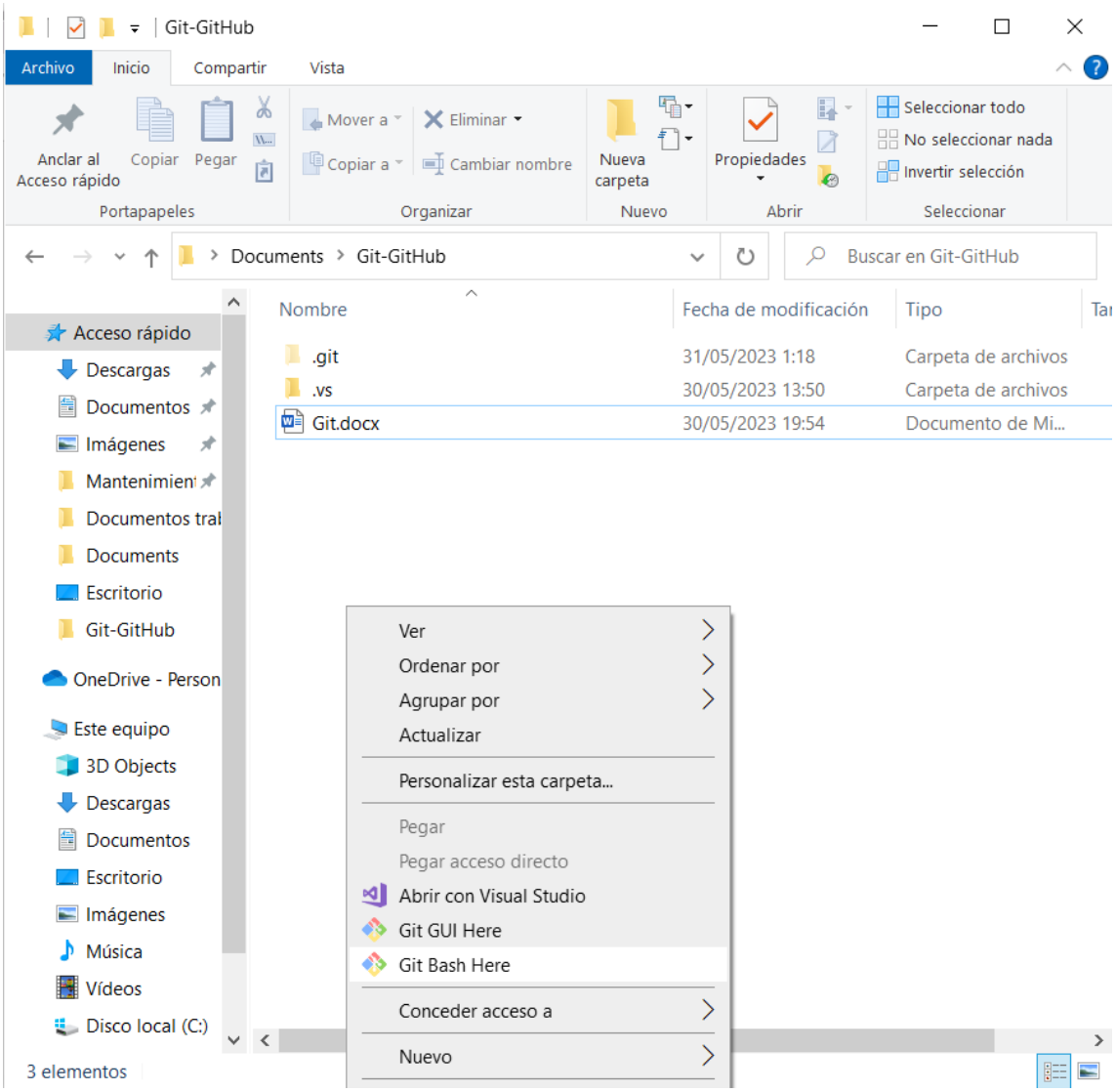
¿Cómo funciona GIT? ¿En qué carpeta estaría el repositorio donde tendremos todas las versiones de nuestro proyecto?

La carpeta la eliges tú. Iremos moviéndonos a través de la consola hasta la carpeta donde queramos alojar nuestro proyecto, y una vez dentro de la carpeta, ejecutaremos el comando.

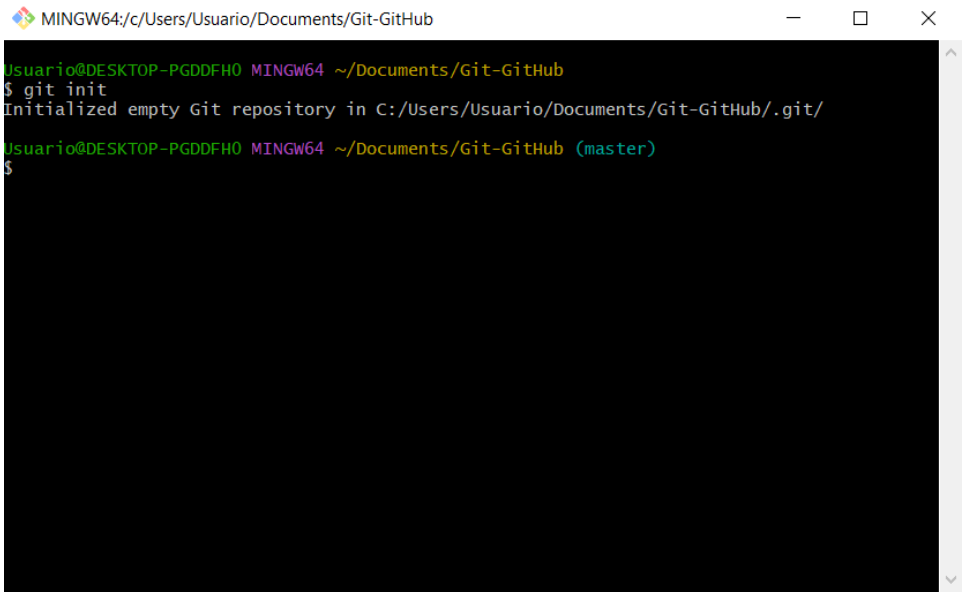
$$git\ init$$

Este comando nos creará, dentro de nuestra carpeta donde alojamos el proyecto, una carpeta oculta con el nombre de .git.

Nota: En este caso, lo que hice yo fue ir a la carpeta en cuestión de forma manual, dar click derecho una vez dentro de la carpeta y seleccionar la opción Git Bash Here.

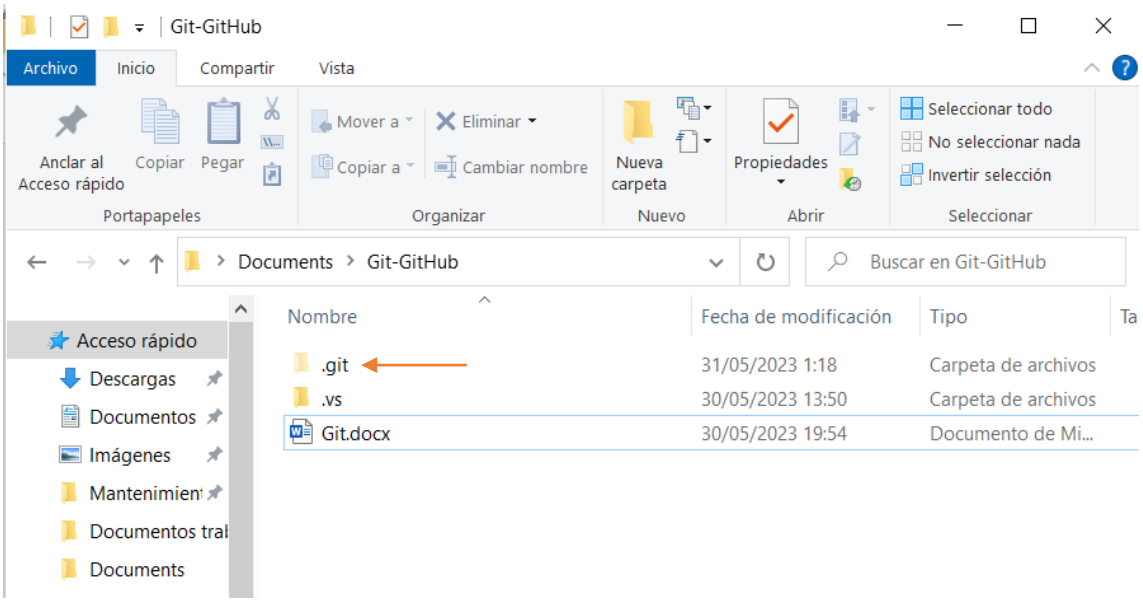


Se me abre una consola y en ella pongo el comando **git init**.

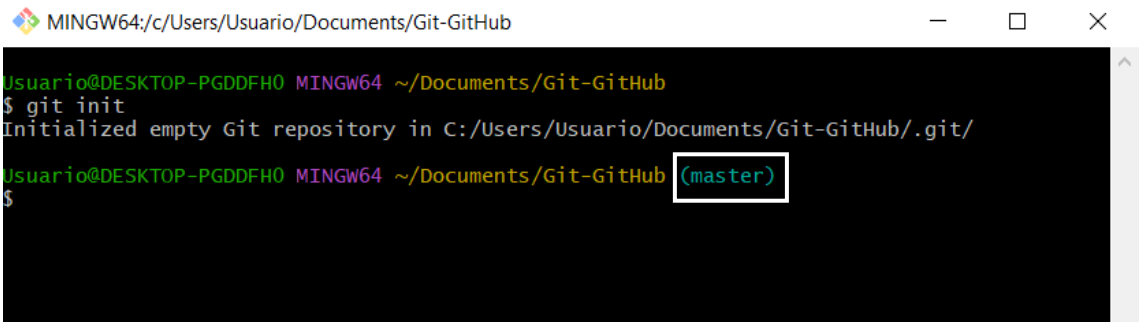


Ocurrieron dos cosas:

- Me generó una carpeta .git dentro de la carpeta donde tengo el proyecto.



- Ahora en el bash, si nos fijamos, en el directorio que estamos sale la palabra (master)



Git se entiende como una ramificación. Donde su rama principal sería esta carpeta que dice master.

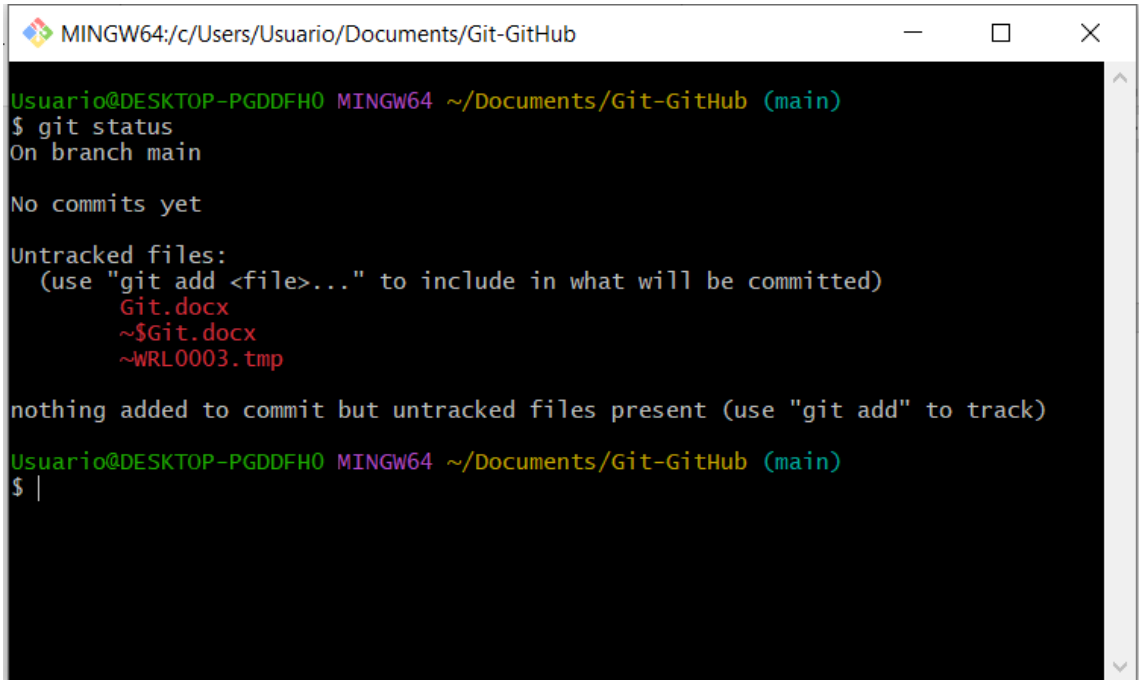
Si preferimos llamar de otra manera a nuestra rama principal, podemos utilizar el comando

$$git\ branch\ -m\ Nombre$$

Nota: Por lo visto, es común denominar a la rama principal como main.

Una vez tenemos nuestra rama principal podemos mirar el estado en el que está.

git status



Nos indica

- Que estamos en la rama main.
- Que no hay commits (más adelante en el documento explicaremos que es esto).
- Y que hay un archivo Git.docx. que no se ha subido como versión.

El archivo WRL0003.tmp es un archivo temporal que crea Windows aparentemente y no es relevante para la explicación.

CREAR VERSIONES

Hasta ahora se entendería que todo lo estamos trabajando de manera local. De momento nuestro Git no tiene ninguna versión de ningún archivo. Entonces lo primero que habría que hacer es añadir ese archivo, el cual guardará como si de una foto única se tratase y cada vez que subamos otra versión de ese mismo archivo será como otra foto diferente.

Para realizar esa "fotografía" de nuestros archivos realizaremos dos comandos.

git add Nombrearchivo

o también podemos usar

git add .

Para añadir todos los archivos de la carpeta a la vez.

```
Usuario@DESKTOP-PGDDFH0 MINGW64 ~/Documents/Git-GitHub (main)
$ git add Git.docx

Usuario@DESKTOP-PGDDFH0 MINGW64 ~/Documents/Git-GitHub (main)
```

Si hacemos git status vemos como ha añadido el fichero, pero aún falta realizar el commit para hacer esa "fotografía".

```
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   Git.docx

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:   Git.docx

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        ~$Git.docx
        ~WRL0003.tmp
        ~WRL2390.tmp
```

El comando que utilizaremos ahora será

$$git\ commit$$

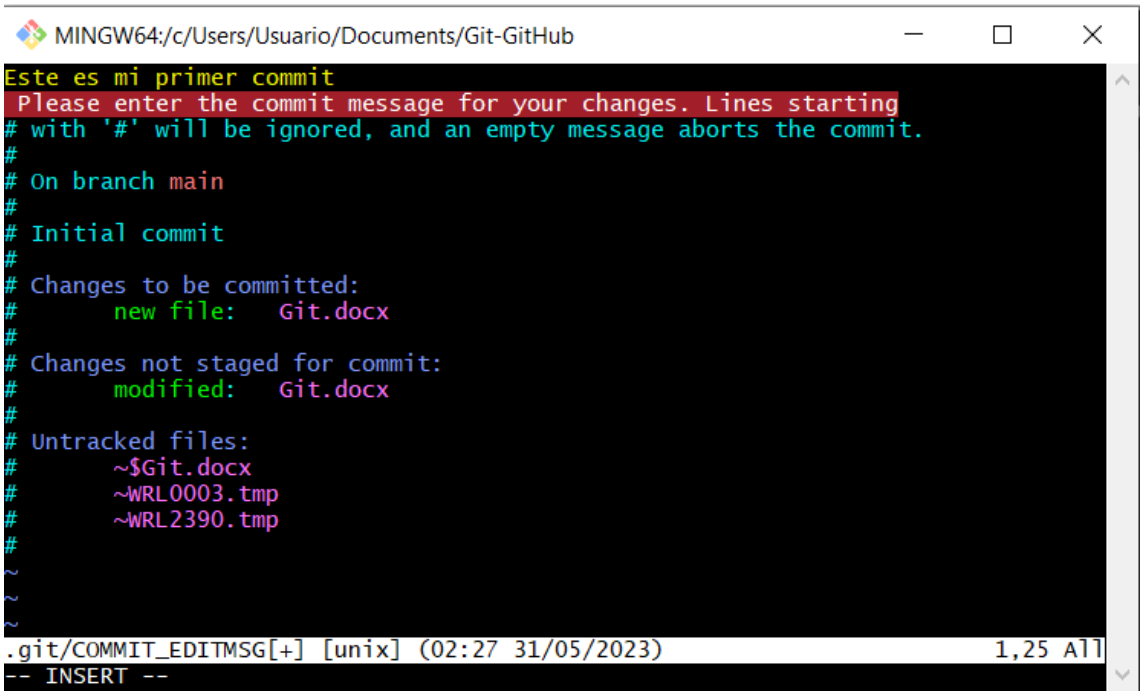
ó

$$git\ commit\ -m\ "Este\ es\ mi\ primer\ commit"$$

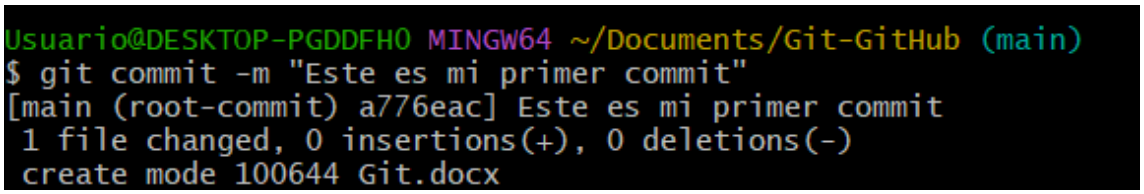
Si simplemente ponemos git commit nos saldrá el siguiente mensaje.

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
#
# Initial commit
#
# Changes to be committed:
#       new file:   Git.docx
#
# Changes not staged for commit:
#       modified:   Git.docx
#
# Untracked files:
#       ~$Git.docx
#       ~WRL0003.tmp
#       ~WRL2390.tmp
#
~
~
~
.git/COMMIT_EDITMSG [unix] (02:27 31/05/2023)                 1,0-1 All
"~/Documents/Git-GitHub/.git/COMMIT_EDITMSG" [unix] 19L, 350B
```

Esto nos dice que no podemos terminar el proceso sin poner un comentario. Con lo cual pondríamos en la primera línea un comentario.



En este caso, voy a abandonar el proceso anterior escribiendo :qa! y dando enter para realizar el commit de la forma sencilla.



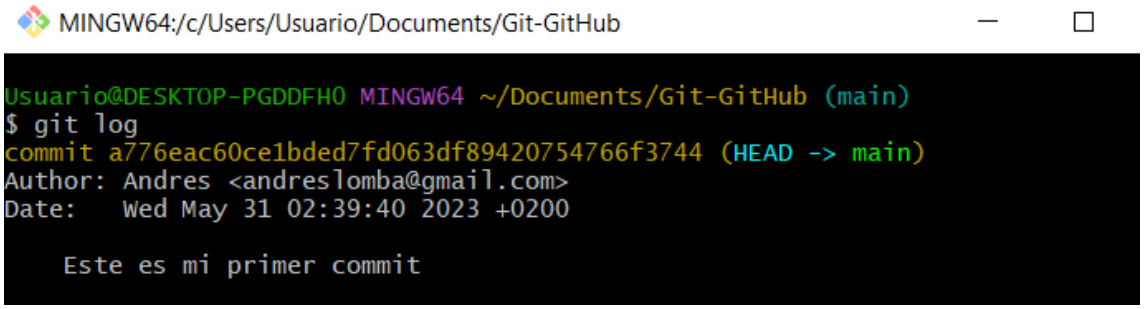
Esto nos crea la instantánea de nuestro archivo, generándole un identificador único que en este caso sería a776eac (a este identificador único se le denomina **hash**).

Con esto ya tendríamos la primera versión de nuestro archivo. Si hacemos status veremos cómo ha agregado la primera versión de nuestro archivo.

Comando log

git log

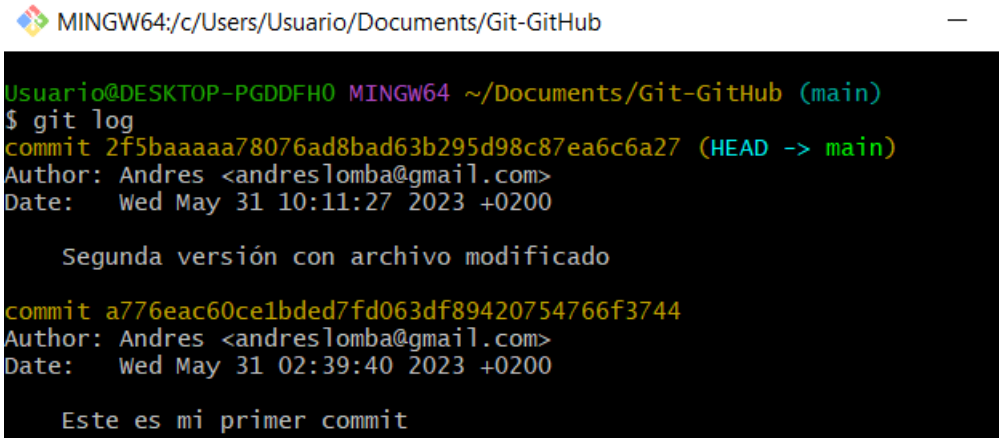
Este comando lo usaremos para verificar que efectivamente ha guardado la versión de nuestro archivo.



Nos dice el hash que tiene esa instantánea, el autor, la fecha y el comentario.

Por eso es muy importante tener configurado el nombre y el email, para poder identificar quien ha guardado esa versión.

He guardado una segunda versión de mi archivo. Si hacemos log se vería de la siguiente manera.



Otras formas de verlo, de forma que se te facilité su lectura cuando tengas muchas versiones es con los siguientes comandos.

$$git\ log\ --graph$$

$$git\ log\ --graph\ --pretty = oneline$$

$$git\ log\ --decorate\ --all\ --oneline$$

DESPLAZAMIENTO ENTRE VERSIONES

Volver a una version anterior.

$$git\ checkout\ nombrearchivo$$

$$git\ checkout\ hash$$

$$git\ reset$$

con estos comandos volveríamos a la última versión que tengamos guardada. Con checkout aparentemente especificamos un archivo en concreto y con reset se aplica a los archivos que el sistema encuentre que están modificados.

Si yo utilizo el checkout hash a una versión anterior, por ejemplo a la primera versión, volvería a cuando solo tenía un único archivo y tenía en el mucha menos información de la que tengo ahora. Pero esto no borra los archivos, pues yo podría volver a la última versión guardada de nuevo. Podríamos ver el checkout como un puntero, que señala allí donde le decimos.

Alias

El concepto de alias para GIT es simplemente cambiar el nombre de un comando por otro nombre más sencillo y fácil de recordar. Así los comandos más complejos serán más rápidos de escribir y recordar.

Ponerle un alias a un comando se hace desde la configuración. Por ejemplo, quiero ponerle un alias(quiero ponerle tree de alias) al comando git log --decorate --all --oneline. Para ello ejecutaremos el siguiente comando.

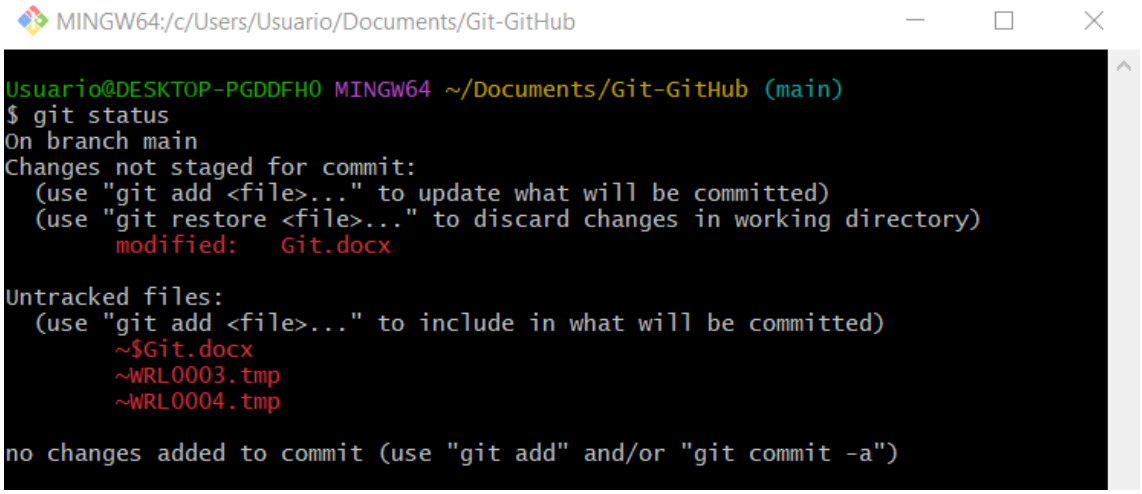
$$git\ config\ --global\ alias.tree\ "git\ log\ --decorate\ --all\ --oneline"$$

De ahora en Adelante el comando git tree equivale al comando git log --decorate --all --oneline

GIT IGNORE

El gitIgnore es un archivo de texto que crearemos nosotros. En el escribiremos los nombres de los archivos que queramos ignorar, ya sea porque no queremos subirlos a nuestro GIT, por comodidad, etc…

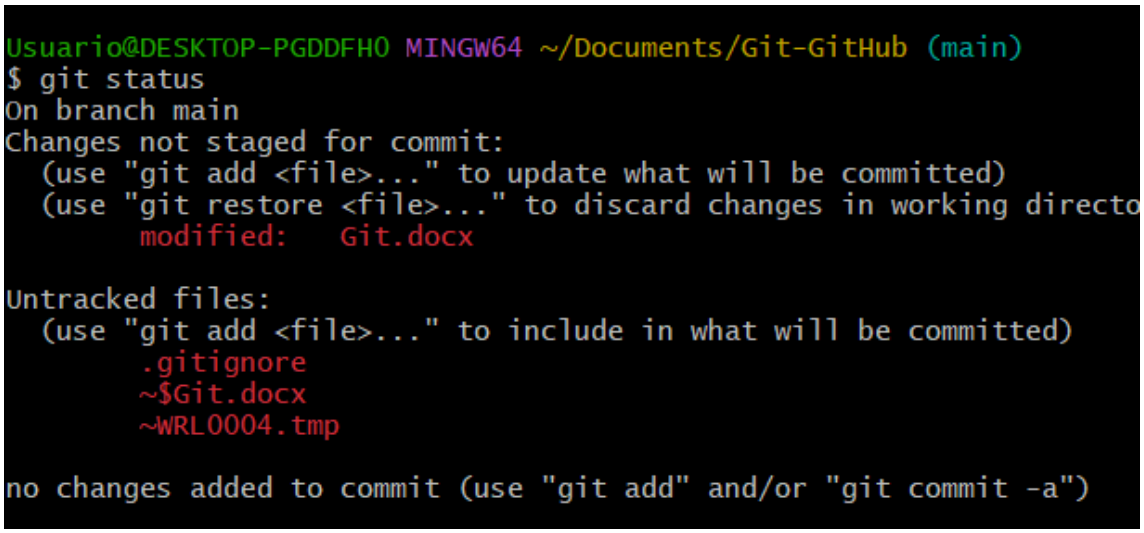
Por ejemplo, en mi caso quiero eliminar el archivo temporal ~WRL0003.tmp



Me creo mi archivo de texto, el cual tendrá de nombre .gitignore. Desconozco si es el nombre del archivo es necesario o si lo es la nomenclatura de dentro del archivo.

Dentro de mi archivo añadiré el nombre de el/los archivos que queramos ignorar con la siguiente nomenclatura.

Ahora si realizamos de nuevo un git status vemos como ahora ignora este archivo.



Nota: Aquel nombre que añadamos al bloc de notas se aplica a TODOS los archivos con dicho nombre en el proyecto, independientemente de que estén en otra carpeta.

GIT DIFF

Si en algún momento necesitásemos mirar que cambios tenemos entre la versión guardada y el archivo que tenemos actualmente utilizaríamos el comando

$$git\ diff$$

Este comando nos diría que se ha cambiado de cada archivo modificado para hacer una comparación.

GIT RESET HARD

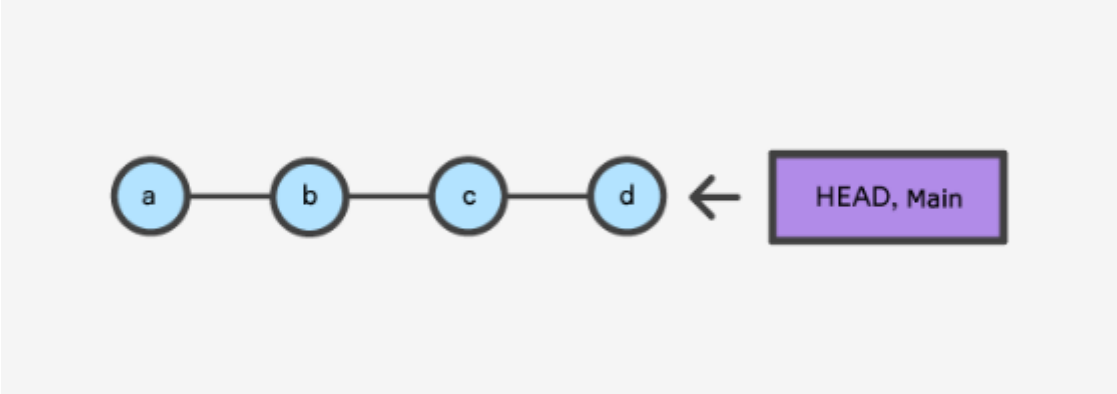
Sirve para volver a una versión anterior de tu proyecto desapareciendo todo aquello que está por delante de esa versión. Por ejemplo, si tengo 4 versiones de un proyecto y quiero volver a la dos, con reset hard volveríamos a la 2 y eliminaríamos la 3 y la 4.

$$git\ reset\ --hard\ hash$$

Nota: No se borran del todo, si tu pusieses el comando git ref log (Historial completo de todas las interacciones de nuestro GIT) veríais que ahí si salen esas versiones supuestamente borradas.

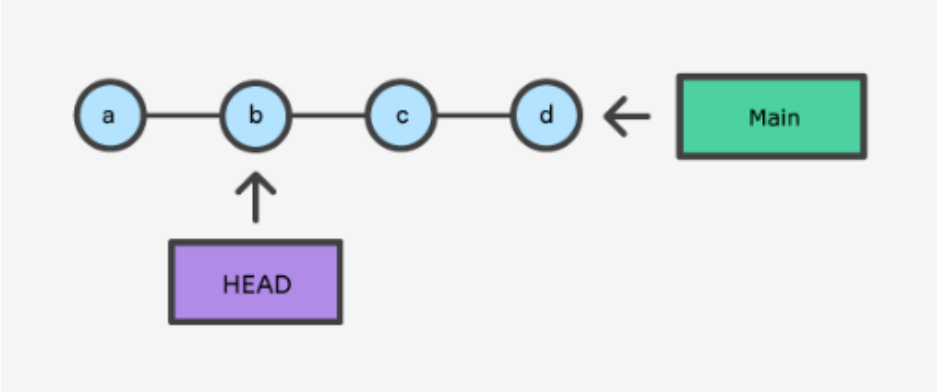
Si quisiéramos volver a la versión 4 del ejemplo anterior, volveríamos a usar git reset --hard utilizando el hash de la última versión.

Concepto: Hay que tener claro un concepto para utilizar el reset y el checkout. El concepto de puntero (HEAD) y el concepto de la rama actual (main). Para entender mejor el concepto vamos u utilizar un ejemplo:



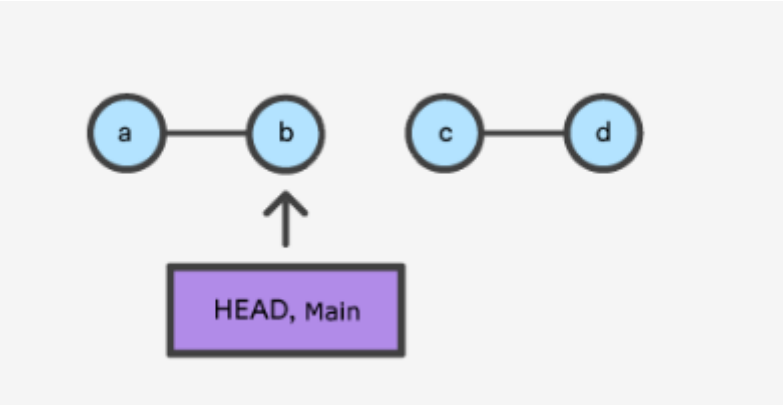
Este ejemplo nos muestra 4 versiones confirmadas en la rama Main. En estos momentos tanto el puntero HEAD como la rama Main apuntan a la versión d.

Si ejecutamos git checkout b:



Vemos como la referencia main sigue apuntando a la versión d, pero el puntero HEAD ahora se ha movido a la versión b.

Si por el contrario ejecutásemos el reset.



Moveríamos tanto el puntero HEAD como la referencia Main a la posición b. Movemos los dos.

Con el comando git log --decorate --all --oneline podemos ver a donde apuntan el HEAD y el Main.

GIT TAG

Sirve para añadir una etiqueta a cada versión. Un nombre identificativo que nos hará diferenciar de forma sencilla las distintas versiones.

git tag Etiqueta

Si lo ejecutamos vemos como nos muestra el tag para dicha versión.

Comandos más avanzados

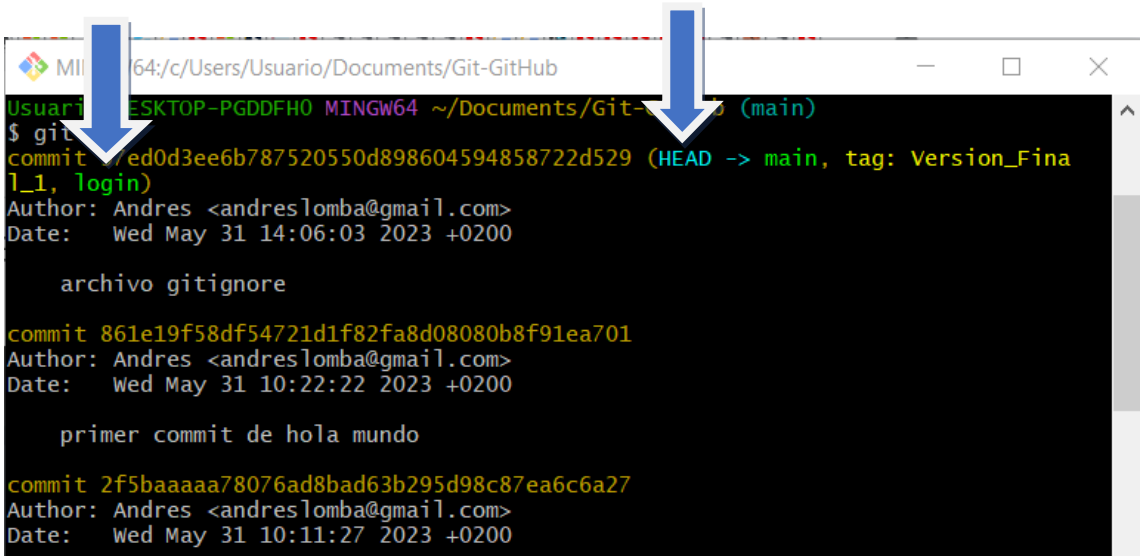
Hasta ahora hemos visto comandos que nos sirven sobre todo en local o en proyectos pequeños. Ahora vamos a empezar a hablar de casos en los que los proyectos son más grandes y donde hay más gente involucrada.

BRANCH Y SWITCH

Hasta ahora sólo hemos trabajado con una misma rama (Main). Si queremos empezar con otra cosa del proyecto en una rama diferente usaremos Branch (que literalmente significa rama).

git branch NombreDeLaRama

Si ahora mismo hacemos una nueva rama llamada login



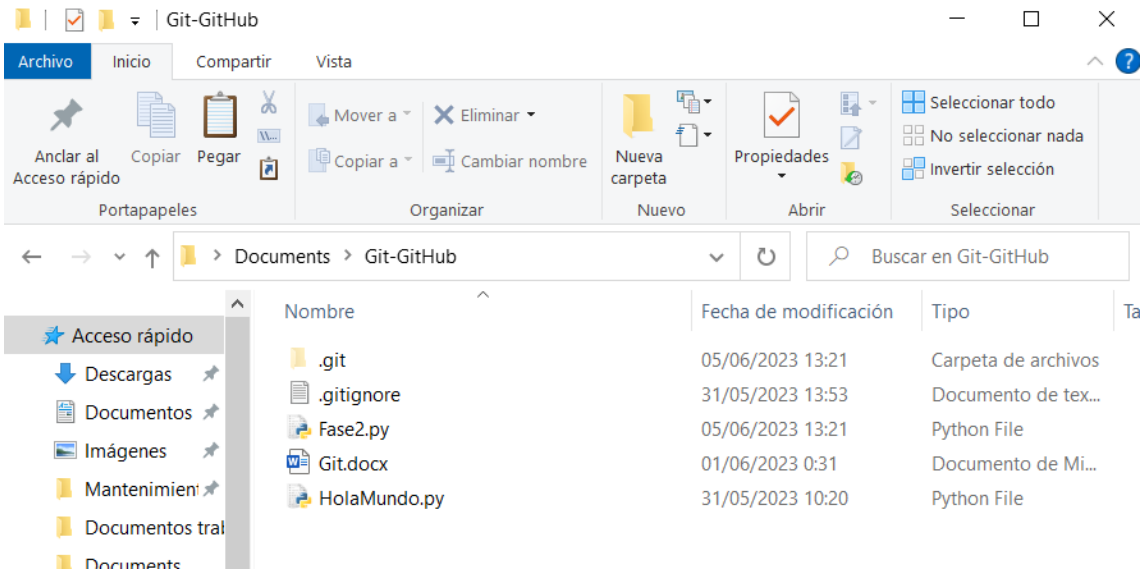
Si nos fijamos ahora tenemos además del main la rama login. La pregunta lógica ahora es. ¿Como me muevo entre ramas? nótese que el puntero HEAD está apuntando a nuestra rama main.

git switch login

Si realizamos un git switch login y después mostramos los commits con git log vemos que ahora el puntero HEAD ahora apunta a la rama login.

Nota: Esta nueva rama sería una ramificación a partir de la última versión guardada de la rama main. Es decir, si yo tengo 4 versiones previas guardadas en main y hago la nueva rama login, ambas ramificaciones tienen como pasado esas 4 versiones previas. Una vez que tengo las dos ramas con su pasado igual, todo lo que añada sí que cambiará, por ejemplo:

Una vez he creado la rama login, crearé en la misma carpeta que he estado trabajado siempre un archivo .PY al que llamaré Fase2



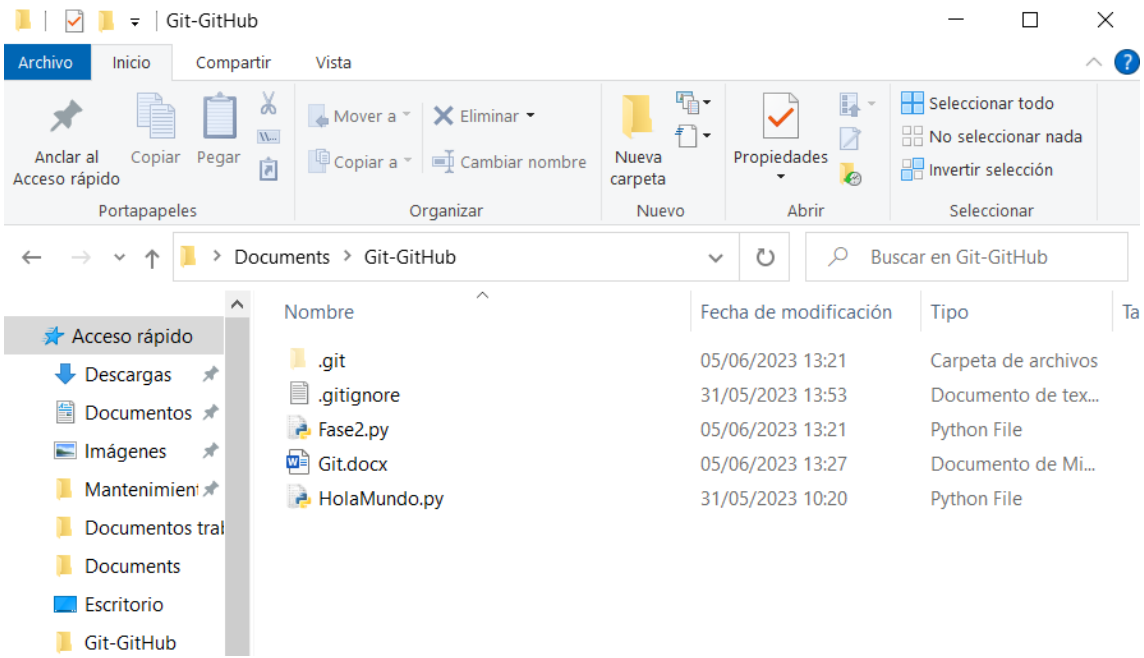
Una vez creado usare el bash para agregar un commit en la rama login. Es importante estar en la rama login.

```
Usuario@DESKTOP-PGDDFHO MINGW64 ~/Documents/Git-GitHub (login)
$ git status
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:   Git.docx

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Fase2.py
        ~$Git.docx

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

```
Usuario@DESKTOP-PGDDFHO MINGW64 ~/Documents/Git-GitHub (login)
$ git add Fase2.py

Usuario@DESKTOP-PGDDFHO MINGW64 ~/Documents/Git-GitHub (login)
$ git commit -m "primer commit fase2"
[login 5a99dae] primer commit fase2
 1 file changed, 1 insertion(+)
 create mode 100644 Fase2.py
```

Según la explicación de la nota anterior, ahora la rama login ya se puede diferenciar de la rama main puesto que esta tiene un archivo más llamado Fase2.py

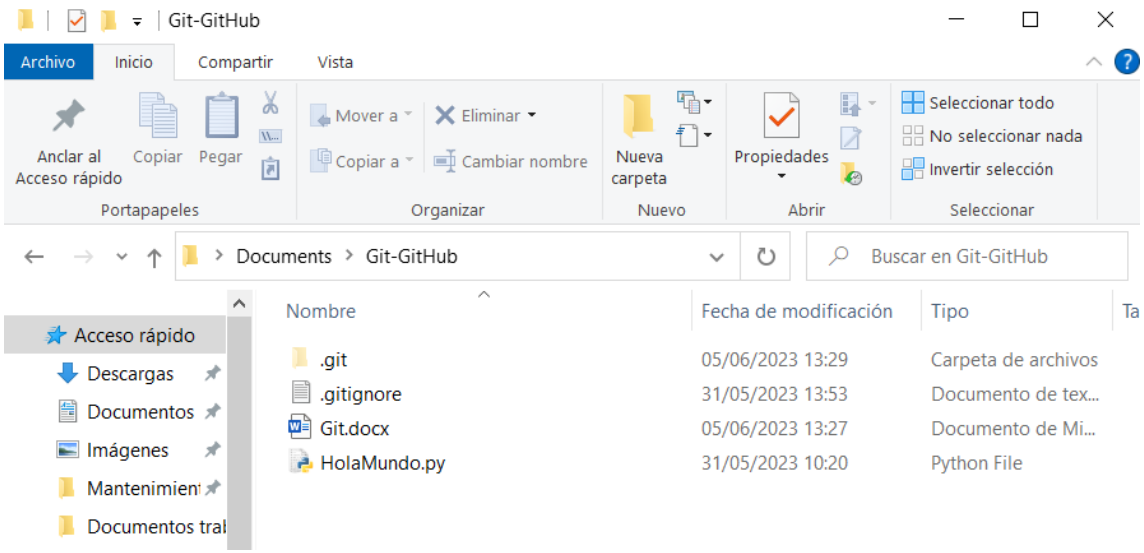
Pues bien, si miramos la carpeta en la que hemos estado trabajando durante todo el tutorial tendremos los siguientes archivos.



Pero si realizamos un cambio de rama en el bash a la rama main.

```
Usuario@DESKTOP-PGDDFH0 MINGW64 ~/Documents/Git-GitHub (login)
$ git switch main
Switched to branch 'main'
M       Git.docx

Usuario@DESKTOP-PGDDFH0 MINGW64 ~/Documents/Git-GitHub (main)
$
```

¡Ahora en la carpeta tendremos que el archivo Fase2.py no está! Puesto que estamos en la rama main, donde este archivo no existe.



GIT MERGE

Este comando se usa sobretodo cuando hay varias ramas independientes y hay equipos trabajando en cada rama de forma autónoma. Si por ejemplo yo estoy trabajando en mi rama Login, sin saber que está haciendo el que está trabajando en la rama Main. Y después de varios días de trabajo, quiero saber si lo que he hecho sigue siendo compatible con lo que ha estado haciendo mi compañero que trabaja en la rama main. Es decir, este nuevo comando sirve para poder mantener una coherencia entre ramas.

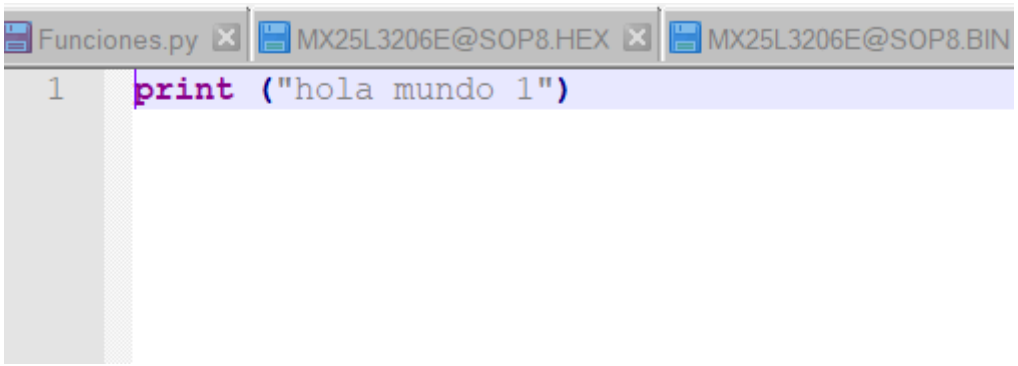
$$git\ merge\ RamaQueQuieroVer$$

En mi caso que quiero comprobar la rama main pondría:

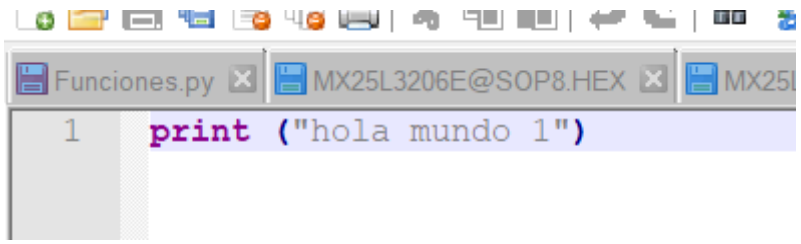
$$git\ merge\ main$$

En mi caso lo que hará será generar una nueva versión en la rama login con los datos que se hayan modificado en los archivos de la rama main. Por ejemplo.

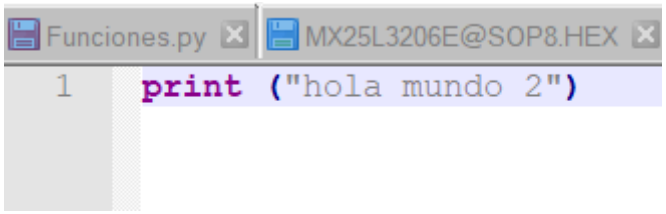
Si abro el archivo HolaMundo.py de la rama main, vemos que tiene el siguiente texto.



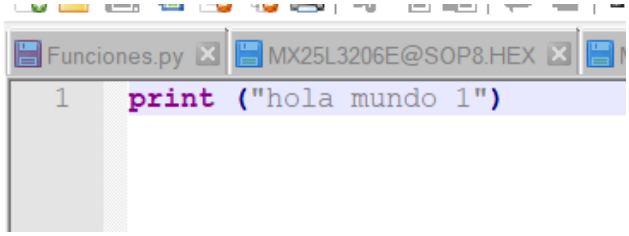
Si abro el mismo archivo de la rama login, también comprobaremos que tiene el mismo texto.



Pues ahora, vuelvo a la rama main y modifico el archivo. Lo agrego a mi git con git add y con su commit. Ahora ya tenemos en la rama main el archivo modificado.



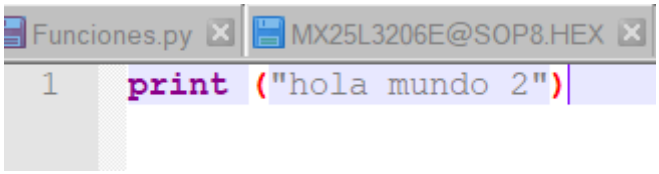
Si nos vamos a la rama login veremos como el archivo sigue teniendo el texto orignial.



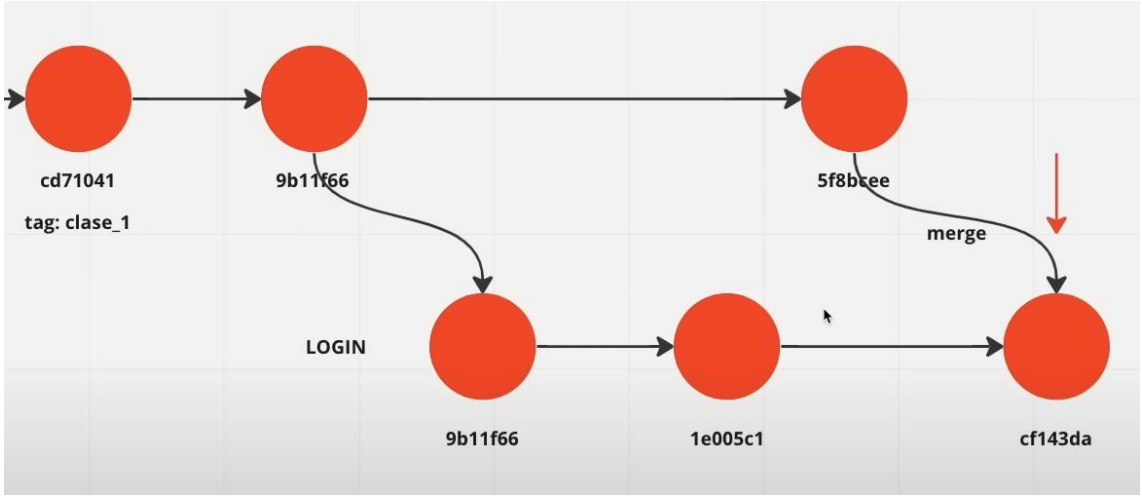
Entonces si fuésemos un equipo que está trabajando en la rama login y quisiéramos comprobar que lo que tenemos modificado en nuestra rama login sigue siendo compatible y sigue funcionando con lo que hay en la rama main, utilizaríamos el merge.

Nota: Es necesario un commit. En caso de que nos salga una ventana con texto, para salir de ella deberemos escribir :exit

Ahora si abrimos el archivo HolaMundo desde la rama vemos como ya está modificado como lo tenían en la rama main



Ahora si realizamos un git log veremos como lo ha guardado como si de una nueva versión de la rama login se tratara.



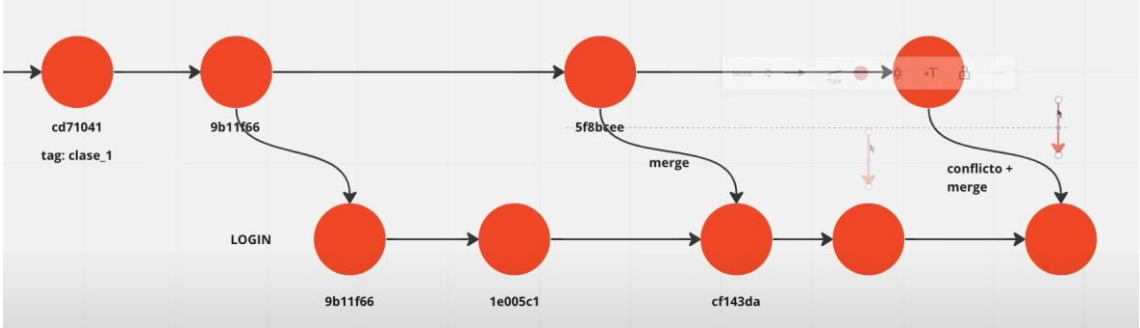
Resolución de conflictos

Pues bien, si se hace un merge y las dos ramas siguen siendo compatibles y sigue funcionando todo perfectamente pues genial, ¿pero… y que pasa si hay conflictos?

Se genera un conflicto cuando desde dos ramas distintas se toca una misma línea de un mismo archivo. Si se tocan líneas diferentes del código no se crea conflicto, pero si se toca la misma línea, sí.

Al hacer un margin y haber conflicto, podremos ver el archivo en concreto donde hay conflicto. Si abrimos el archivo nos pondrá donde está la línea que se ha tocado en ambas ramas y nos mostrará las diferencias.

La forma de hacer la corrección sería dejando el archivo como tendría que quedar y luego en el Bach haciendo un git add y git commit para resolver el conflicto.



GIT STASH

El comando Stash es un comando que sirve para hacer un guardado de algo en lo que estás trabajando que al no estar completo no quieres hacer commit. Si por ejemplo estás en medio de un código en la rama login y te llaman que te cambies urgente a la main, cuando intentas cambiarte de rama el GIT no te va a dejar hasta no guardar los cambios. Como el código está a medias y no quieres hacer commit de eso, utilizas el comando Stash. Es una especie de guardado pero en local, donde sólo lo puedas ver tu y no afecta al árbol para nada.

$$git\ stash$$

para ver que stash tenemos pendientes

$$git\ stash\ list$$

Para usar lo que está guardado en Stash

$$git\ stash\ pop$$

Si no te interesa recuperar lo que has guardado en Stash, puedes eliminarlo

$$git\ stash\ drop$$

Eliminación de ramas

Una vez se termina el trabajo en la rama login, esto se implementará en la rama principal y así con todas las ramas que van derivando de ella, cada vez que se termina se implementa en la rama main (lo normal sería pensar que se puede implementar con la instrucción merge, pero veremos más adelante como se hace).

Entonces, ¿qué hacemos ahora con las ramas que ya se han implementado en la rama main y no se van a usar? Se eliminan con el comando:

$$git\ branch\ -d\ NombreRama$$

Nota: Evidentemente las ramas no se eliminan del todo. A efectos generales si que se eliminan, pero si realizamos un checkout lo encontraremos.

Hasta aquí la parte de GIT

WEB CON TODOS LOS COMANDOS

Página con el resumen de comandos GIT

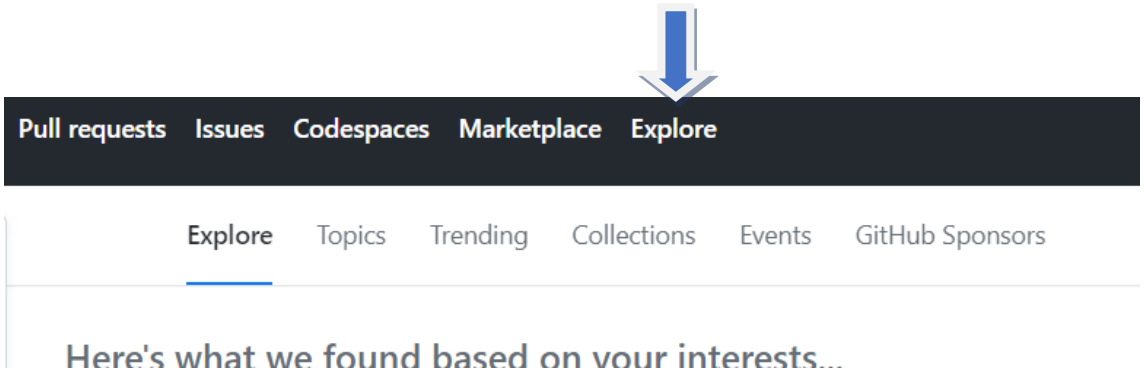
https://training.github.com/downloads/es_ES/github-git-cheat-sheet.pdf

GITHUB

Lo primero a dejar claro es que GIT y GITHUB no es lo mismo. GIT es un sistema de control de versiones, un software que nos sirve para trabajar con código de forma segura. Todo de forma local.

Github es una nube, donde vamos a poder ir dejando los cambios que hagamos y en el que otras personas también podrán realizar cambios.

Se podría decir que es la red social por excelencia para los desarrolladores, así que además de dejar allí nuestros proyectos es importante que tenga un buen aspecto para aquellos que vean nuestro perfil.



En Explore podremos ver cositas que tengan subidas otros usuarios. Esto se aprende mejor cacharreando por tu cuenta.

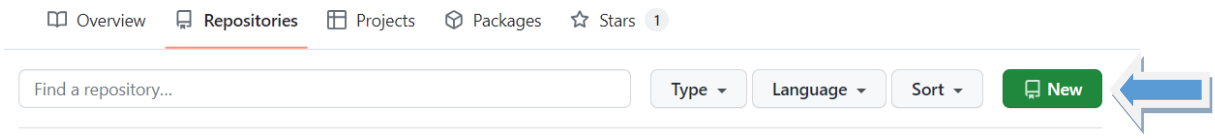
REPOSITORIO

El primer concepto que vamos a ver es el de Repositorio. Va a ser un sitio donde se va a trabajar con GIT.

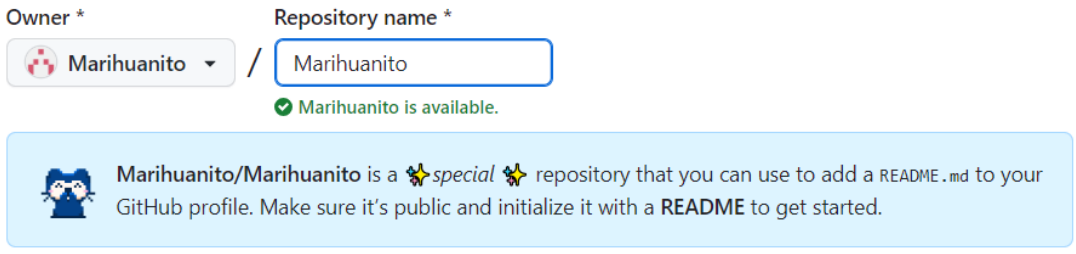
Para crear un repositorio nos da dos opciones:

- Subir un proyecto que hemos creado en nuestra máquina con GIT
- Crear desde GitHub el proyecto con GIT.

Si nos vamos a repositorios y le damos a crear un nuevo



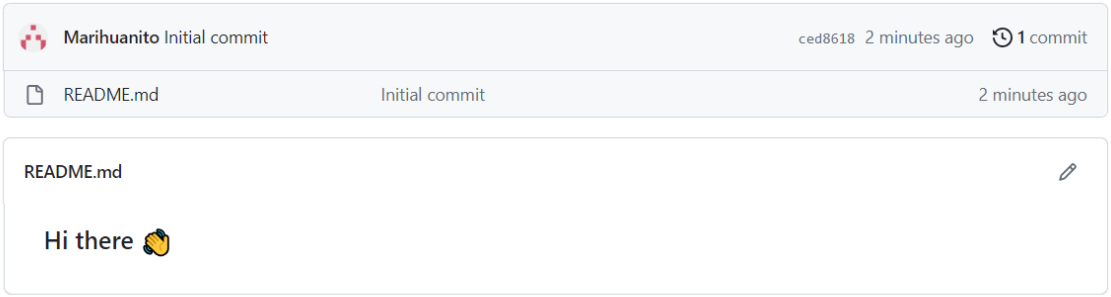
Para el primer repositorio se recomienda ponerle el mismo nombre de tu cuenta para hacerlo como tu página personal.



Se clicaría la opción de README, es importante que todos los repositorios tengan un README que nos introduzca lo que nos vamos a encontrar en dicho repositorio.

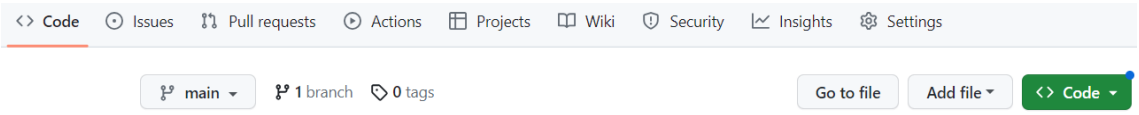
El resto de opciones se puede dejar por defecto y ya se podría crear el repositorio.

Lo de las licencias es algo que de momento no se explicará.



Con esto, si entramos en https://github.com/Marihuanito ya entraríamos a mi repositorio, que en un futuro se convertirá en mi página principal de GitHub.

Estas son las opciones que nos aparece en nuestro repositorio.



Issues: Alguien tiene incidencias abiertas

Pull request: Alguien nos ha enviado código para meter en nuestro repositorio

CONEXIÓN

¿Cómo conectaríamos nuestro ordenador con GitHub?

Aquí hay un tutorial que nos explicaría como hacerlo:

https://docs.github.com/es/authentication

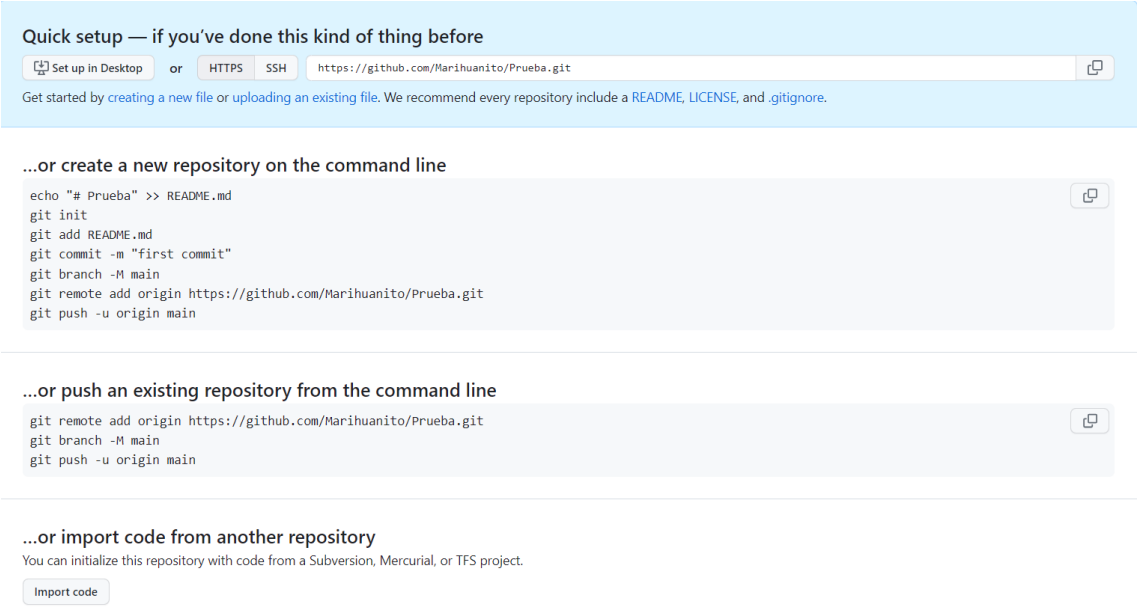
La manera más común es por medio de SSH y seguiré el siguiente tutorial para realizar la comunicación entre nuestro ordenador a GitHub.

El tutorial nos dice

- Como crear la clave pública y privada en nuestro ordenador.
- Como agregar la clave al GitHub.
- Comprobar que nuestra cuenta se ha asociado correctamente.

Añadir nuestro proyecto al repositorio

He creado un nuevo repositorio llamado “Prueba” para subir en él nuestro proyecto Git. Este, ahora mismo tendría la siguiente pinta. No he añadido ReadMe.



Para indicar con que repositorio vamos a enlazar el proyecto usamos el siguiente comando.

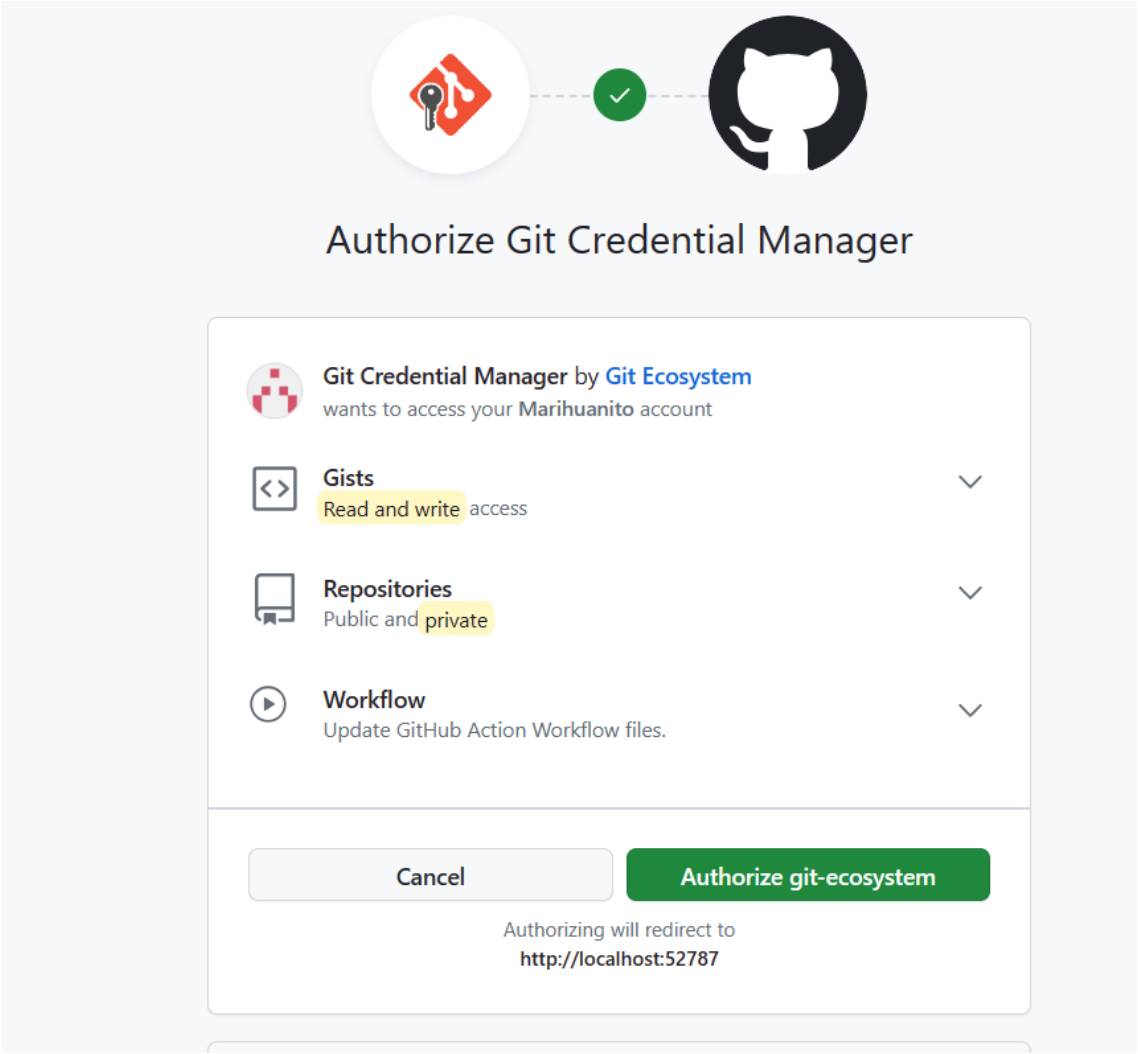
```
git remote add origin
https://github.com/Marihuanito/NombreDeRepositorio.git
```

Si no da error es que está correcto.

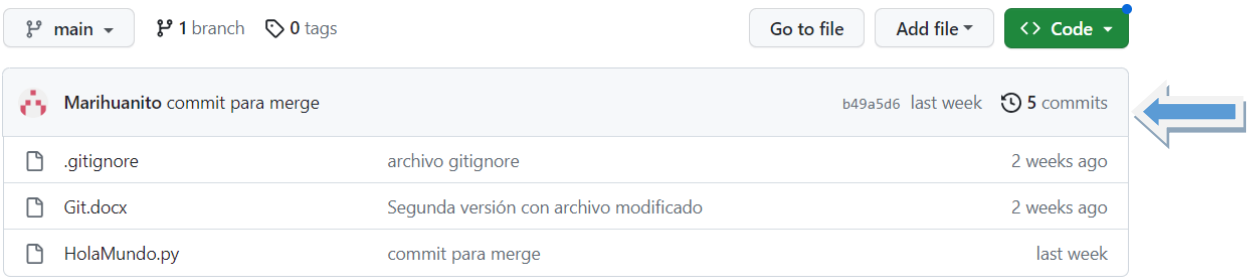
Una vez tenemos enlazados el proyecto y el repositorio, el siguiente paso es subir lo que tenemos en el proyecto al repositorio. Se realiza con el siguiente comando.

```
git push -u origin main -> Este se usa la primera vez que se suben
archivos.

Git push -> Este se usa para subir los archivos de normal.
```

Una vez lo usas, te pide que te autentiques con tu usuario y contraseña de GitHub y una vez te logueas te sale el siguiente mensaje.



```
Usuario@DESKTOP-PGDDFH0 MINGW64 ~/Documents/Git-GitHub (login)
$ git push -u origin main
Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 4 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (15/15), 522.42 KiB | 13.75 MiB/s, done.
Total 15 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/Marihuanito/Prueba.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.

Usuario@DESKTOP-PGDDFH0 MINGW64 ~/Documents/Git-GitHub (login)
$
```



Y ya tenemos los archivos subidos en nuestro repositorio. Vemos como tienen ya los commits realizados.

Si queremos descargar el historial de cambios que ha habido, sin descargar los cambios perse, usaríamos el comando

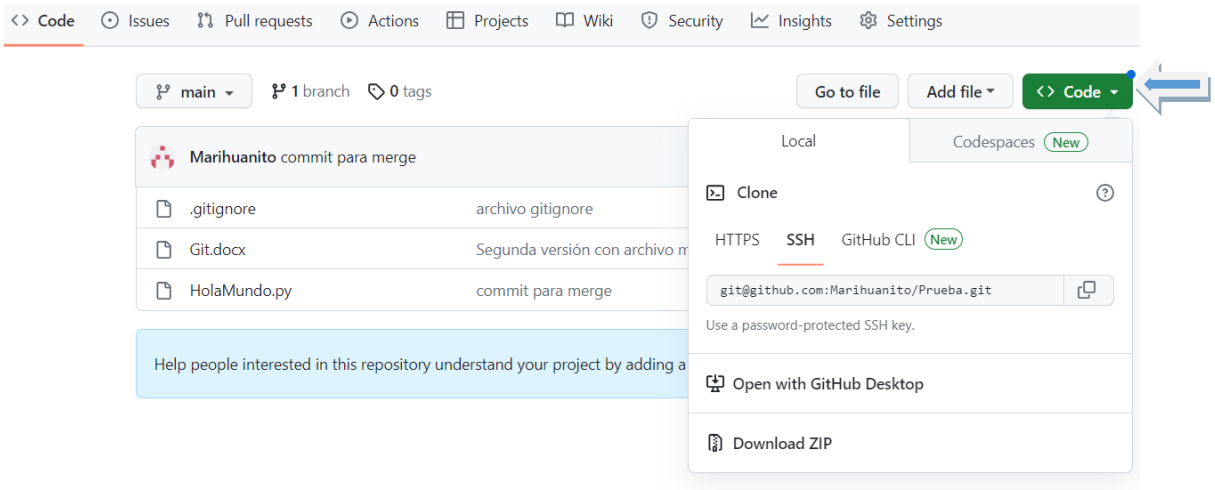
$$git\ fetch.$$

Si queremos descargar tanto el historial, como los cambios usaríamos:

$$git\ pull$$

En el caso de que seas una persona que acaba de entrar al proyecto y quieras bajar el proyecto se puede con el comando clone.

$$git\ clone\ "enlace\ SSH"$$

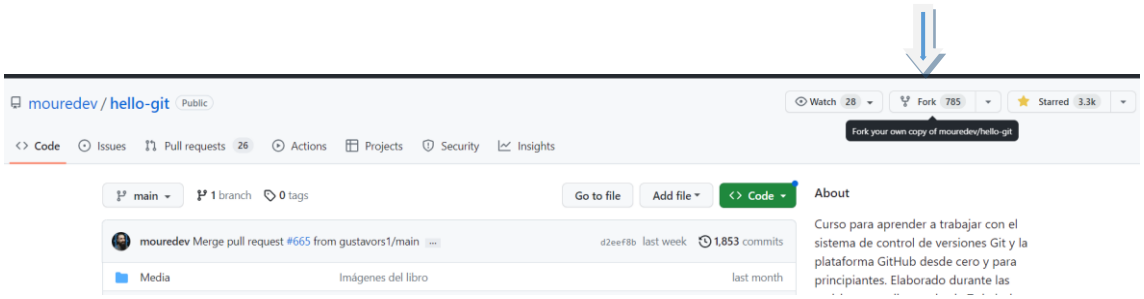


Desde aquí podemos ver el enlace SSH y podríamos descargarlo manualmente desde el botón download ZIP

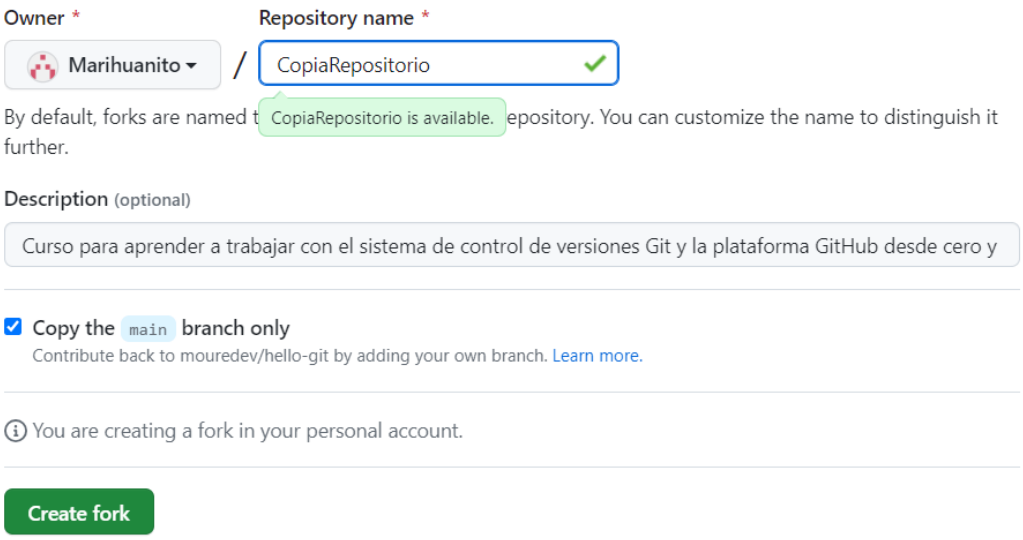
Copiar repositorio de otro usuario

El método que se utiliza para poder modificar un repositorio de otro usuario del cual no tienes permisos es copiando ese repositorio a un repositorio tuyo.

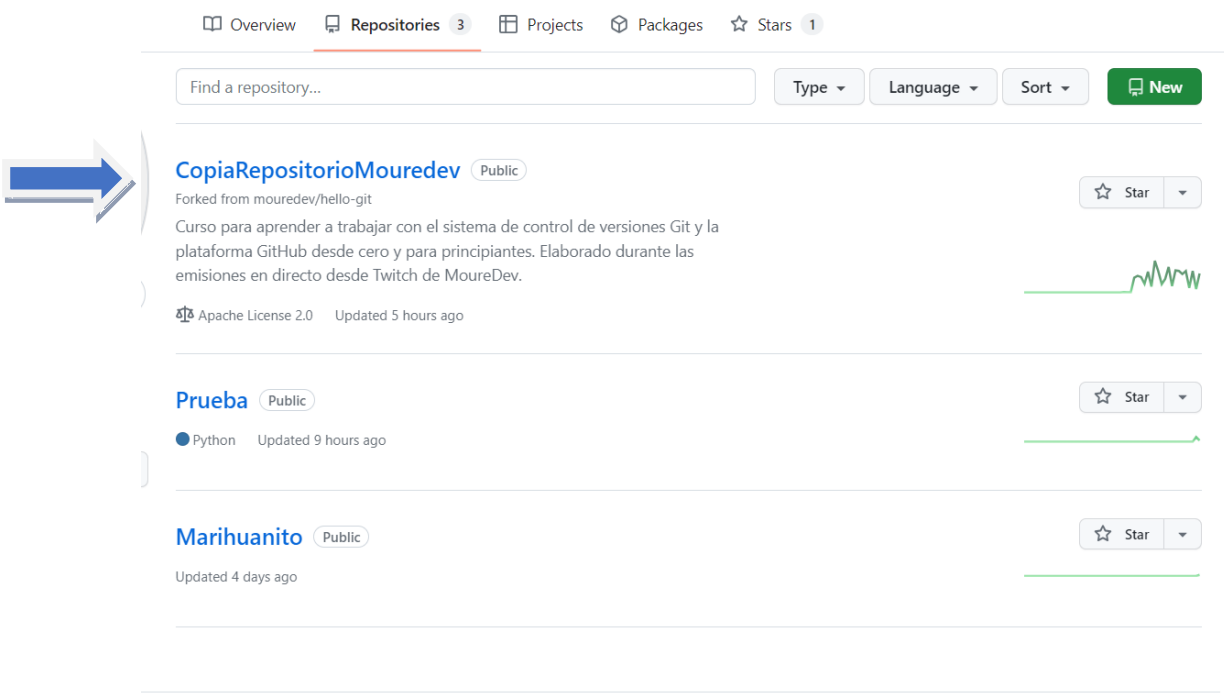
En este caso buscaríamos el repositorio del usuario que queremos copiar y le damos a FORK



A continuación, nos sale para crear un nuevo repositorio y el check es por si queremos copiar solo la rama main o todas las ramas.



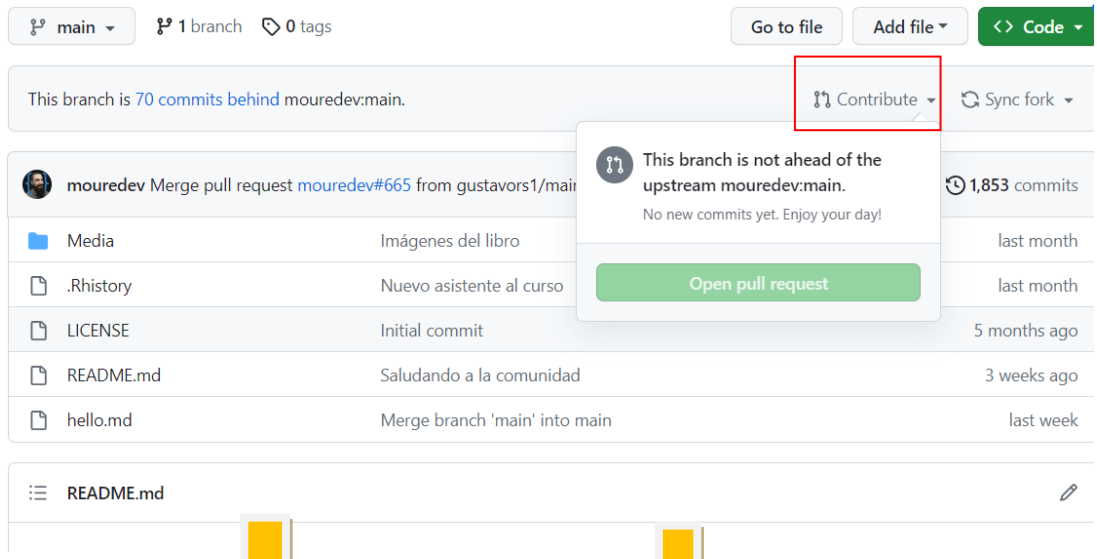
Ahora si miro mis repositorios me sale uno mas.

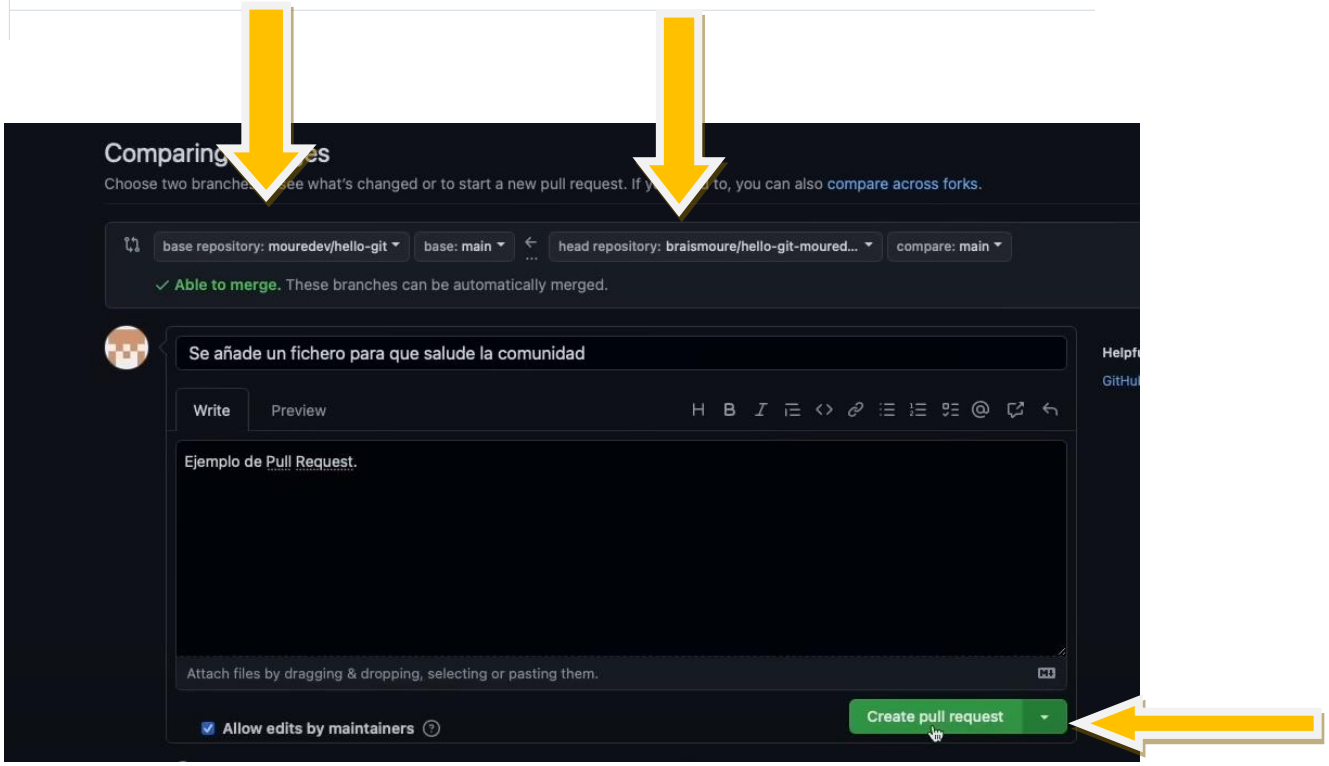


Una vez copiado, lo bajaríamos a nuestro ordenador con un git clone de esta copia y ahora ya podríamos modificar lo que queramos.

Como es un clon del repositorio de mouredev, vemos que podemos sincronizarlo. Cada vez que nos deje hacer un sync, significa que el repositorio original ha sido modificado.

Si hemos realizado cambios y creemos que son cambios importantes que deberían estar en el repositorio original, realizaríamos una petición para que se acepten nuestros cambios. Para ello deberíamos de tener la sincronización al día y clicar en la opción contribute.





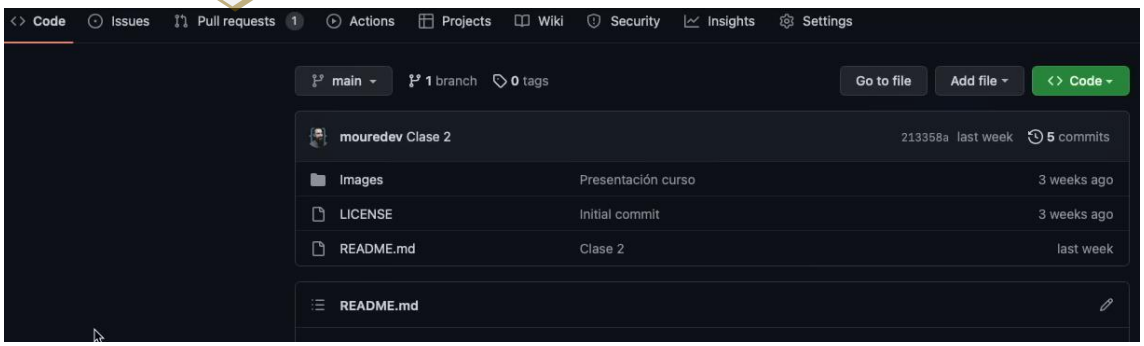
Base repository sería el repositorio al que se hace la solicitud para el cambio

Head repository el repositorio solicitante

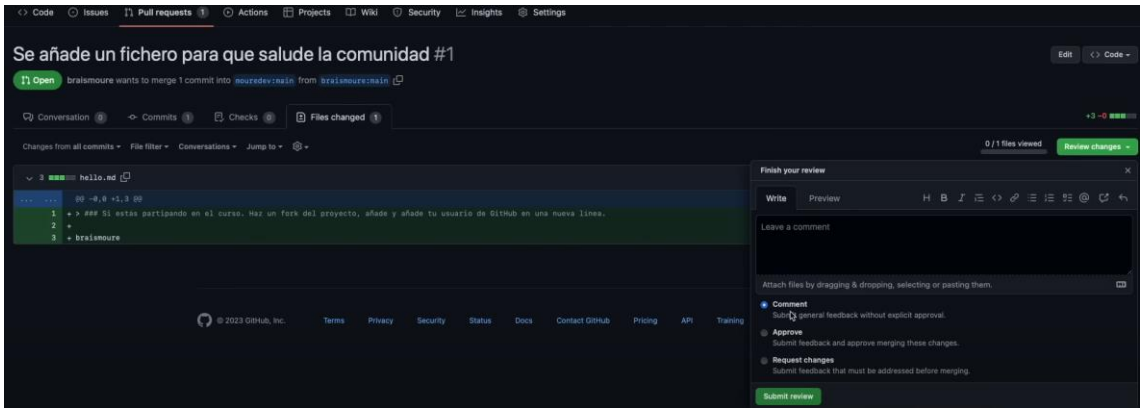
Create pull request crea la solicitud

Más abajo te dice que cambios estarías realizando en comparación al repositorio destino.

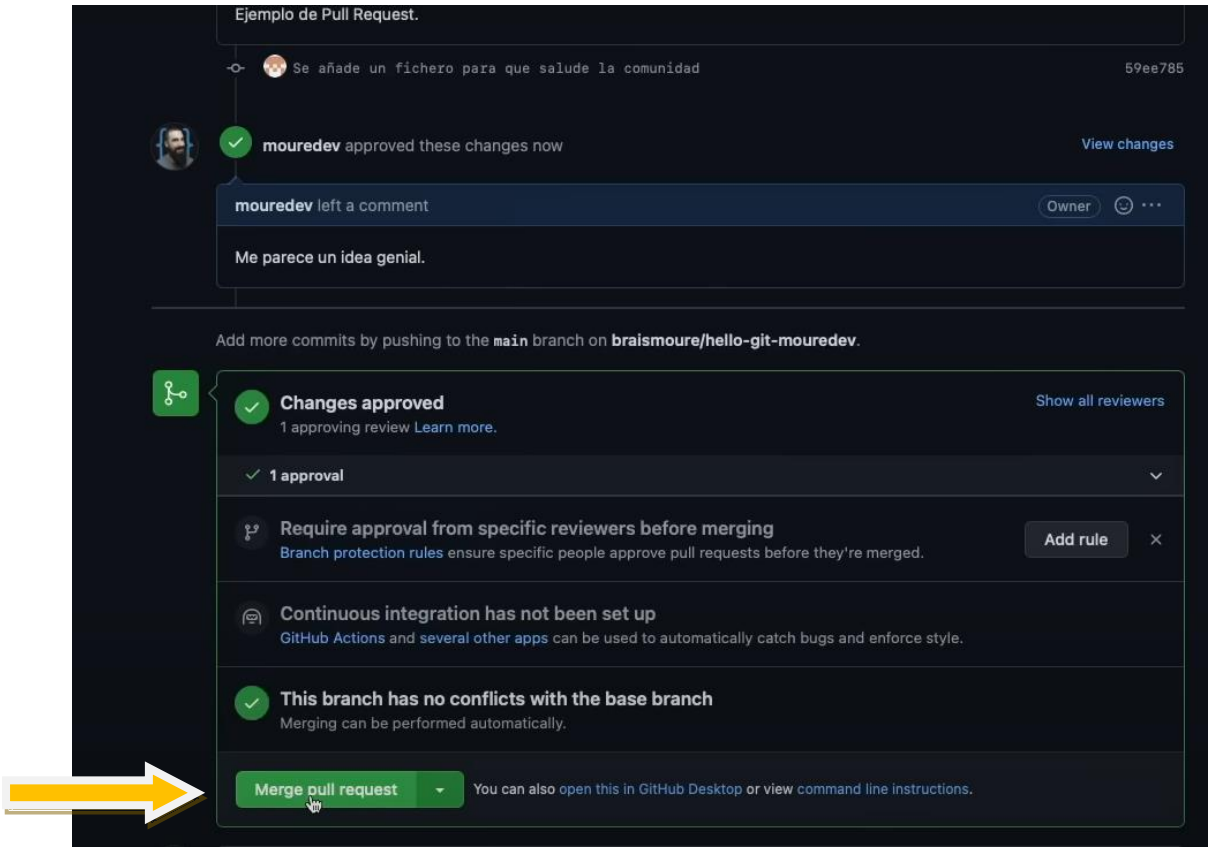
Una vez realizada la solicitud, la solicitud le saldrá al usuario del repositorio original.



Ahora esa persona puede ponerte un comentario, aceptar tus cambios o denegarlos



Una vez aprobado se hace un merge para terminar de añadir los cambios.



Con esto que hemos aprendido hasta ahora tenemos los conceptos básicos con los que podríamos prácticamente trabajar en proyectos como desarrolladores.