GIT Y GITHUB

¿Git? ¿GitHub?

Ambos son sistemas de control de versiones de archivos, en lugar de tener muchas versiones de un mismo documento estos almacenan la versión final lograda, además de la versión final editada por otros usuarios permitidos.

A la hora de querer subir código a la nube se hace uso de GitHub

Git init:

Para crear un repositorio solo basta con crear la carpeta donde van a estar tus archivos y desde terminal escribir “git init” dentro de la carpeta.

Ejecutar el comando dentro de la carpeta del proyecto

```
Admin@DESKTOP-40NS536 MINGW64 ~/Desktop/repo1
$ git init
Initialized empty Git repository in C:/Users/Admin/Desktop/repo1/.git/
```

ls -al -> Para ver los archivos ocultos de una carpeta

Git status:

Nos permite ver cual es el status del proyecto actual.

Ejecutar el comando dentro de la carpeta del proyecto

```
Admin@DESKTOP-40NS536 MINGW64 ~/Desktop/repo1 (master)
$ git status
On branch master

No commits yet

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

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

Lo que dice es que aún no hemos añadido el archivo “historia.txt” por lo cual los cambios no se verán reflejados

Git add “nombre archivo”:

Permite que git este pendiente de los cambios hechos a cierto archivo para así controlar sus versiones.

```
Admin@DESKTOP-40NS536 MINGW64 ~/Desktop/repo1 (master)
$ git add historia.txt
```

Siempre hay que hacer un git add antes de un commit

Git add .:

Agrega los cambios realizados de todos los archivos pertenecientes a la carpeta.

Siempre hay que hacer un git add antes de un commit

Git rm --cached “nombre archivo”:

Hace que git deje de estar pendiente en los cambios hechos al archivo; lo contrario al git add.

Git commit -m “mensaje”:

Envía los cambios hechos al archivo a la BD de git (repositorio).

```
Admin@DESKTOP-40NS536 MINGW64 ~/Desktop/repo1 (master)
$ git commit -m "Primer commit"
[master (root-commit) 7ec68cf] Primer commit
 1 file changed, 5 insertions(+)
 create mode 100644 historia.txt
```

Siempre hay que hacer un git add antes de un commit

Git config:

Lista de todas las configuraciones que tiene git

Git config --list:

Configuración por defecto de mi git.

Git config --list --show-origin:

Donde están las mis configuraciones guardadas.

Git config - -global user.name “nombre usuario” :

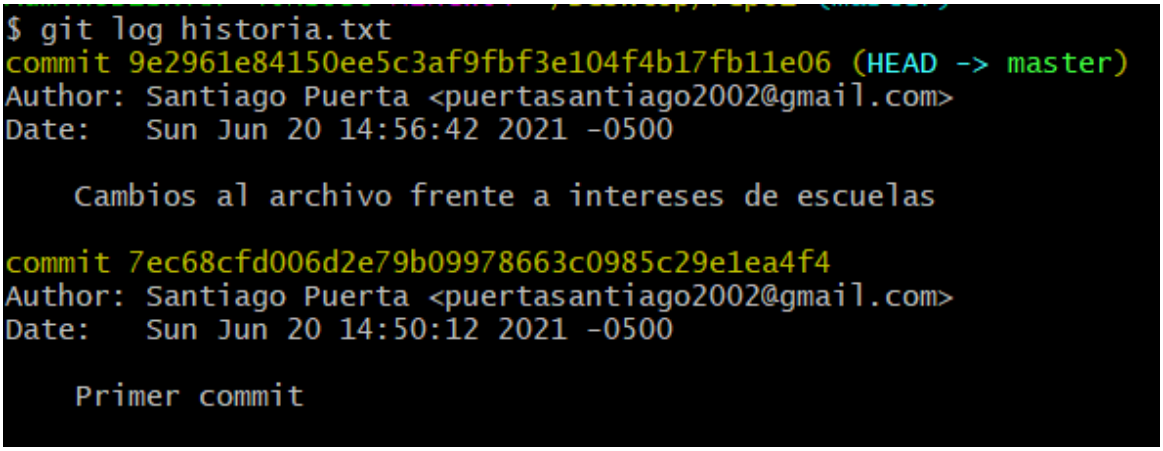
Poner nombre de usuario en git.

Git config - -global user.email “correo electronico”:

Poner correo electronico en git.

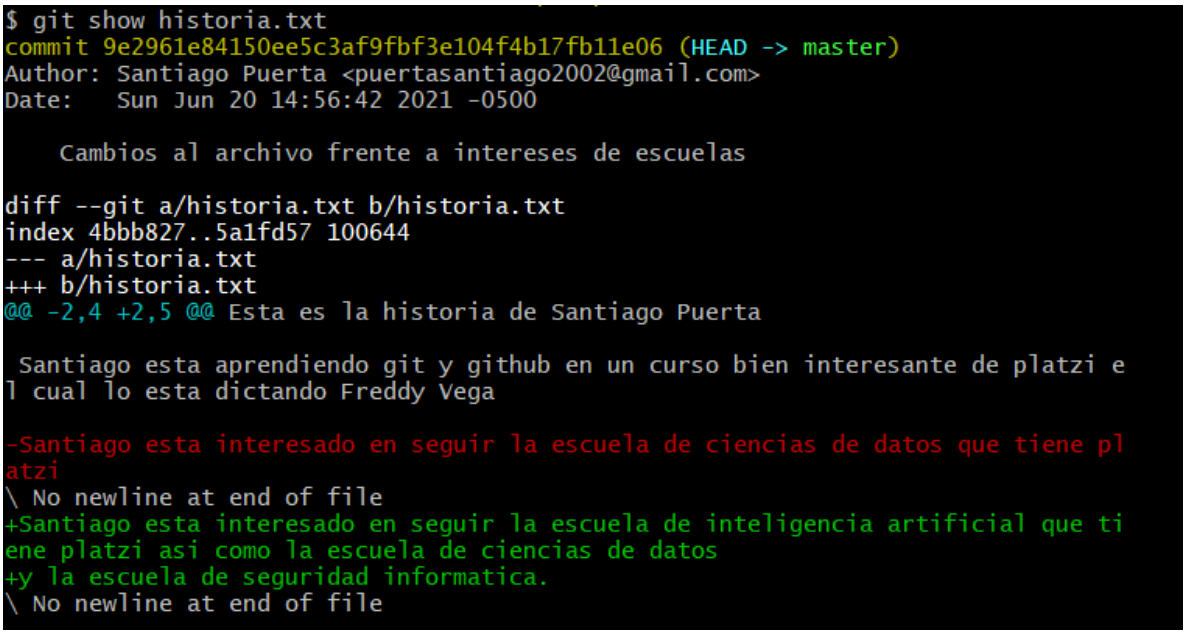
Git log “nombre archivo”:

Permite ver la historia de commits del archivo.



Análisis de cambios en archivos de proyecto:

git show “nombre archivo” -> Muestra los cambios hechos sobre un archivo



commit “…” -> Ultimo commit del archivo

author -> autor

date -> fecha

mensaje del ultimo commit

diff -> Coger versión anterior del archivo y compararla con la nueva versión de este

index “…” -> Indicador dentro de BD de git de donde están los cambios

Hay versión a

Hay versión b

Cuantos bytes cambiaron, lugares donde cambiaron

Archivo y cambios al archivo

No new line …

Git diff “comit ID1” “comit ID2” -> Compara las diferencias entre commit ID1 y commit ID2

```
$ git diff 7ec68cfd006d2e79b09978663c0985c29e1ea4f4 e711cda131b24f6698ff685e802af5be154818f0
diff --git a/historia.txt b/historia.txt
index 4bbb827..5b3d311 100644
--- a/historia.txt
+++ b/historia.txt
@@ -2,4 +2,7 @@ Esta es la historia de Santiago Puerta

 Santiago esta aprendiendo git y github en un curso bien interesante de platzi el cual lo esta dictando
 Freddy Vega

-Santiago esta interesado en seguir la escuela de ciencias de datos que tiene platzi
\ No newline at end of file
+Santiago tiene 19 años y estudia sistemas.
+
+Santiago esta interesado en seguir la escuela de inteligencia artificial que tiene platzi asi como la
escuela de ciencias de datos
+y la escuela de seguridad informatica.
\ No newline at end of file
```

Ciclo básico de trabajo en git:

El comando git init da paso a dos cosas, la primera es la creación del staging en memoria RAM, el área donde se van a ir guardando tus cambios al principio; y la segunda es que se crea el repositorio, el área donde se van a guardar los cambios al final.

El comando git add guarda el archivo en el staging área; dicho archivo se encuentra en espera de ser enviado al repositorio.

Con el comando git commit el archivo se va al repositorio

Sin git add {estado untracked (sin rastrear)}

Git add {estado tracked (rastreado), staging área}

Git commid {pasa de estar tracked en staging a estar tracked en el repositorio}

Git reset “commit ID1” - -hard:

Todo vuelve a la versión del commit ID1

Git reset “commit ID1” - -soft:

Todo vuelve a la versión del commit ID1; lo que haya en staging se queda en staging, lo que haya en disco se queda en disco.

Git diff:

Nos muestra los cambios que hay en staging vs los que aun no se mandan a staging.

Git log - -stat:

Muestra los archivos a los que se le han hecho cambios específicos; muestra cada commit desde el más reciente hasta el más viejo.

Git checkout “commit ID3” “nombre archivo”:

Permite regresar a como era el archivo en el momento del commit ID3

Git checkout master “nombre archivo”:

Permite regresar a la última versión del archivo.

Git reset HEAD:

Sacar archivos del área de staging.

Ramas o branches:

Copia del ultimo commit en un lugar diferente; forma en la cual podemos hacer cambios sin afectar la rama principal (master).

El commit más reciente es la cabecera (HEAD), que versión de commit estoy viendo de los archivos; el HEAD también indica en que rama estoy trabajando

Git branch “nombre de la rama”:

Para crear una nueva rama o brach de git.

Git status -> Me permite ver en que rama me encuentro

Git checkout “nombre de la rama” -> Me permite cambiar de rama

Merge -> Unión de los cambios de una rama con otra

Git branch:

Da una lista de las ramas que hay en el repositorio; dice en que rama estas con un * al lado de la rama y la rama resaltada en verde.

```
$ git branch
  master
* rama1
```

Git merge “nombre rama”:

Trae el ultimo commit de la rama1 y el ultimo commit de la rama2 y los fusiona

La HEAD debe de estar en la rama principal, por ejemplo, si llamo al merge de la rama 1 desde la rama2, la rama2 se convertirá en mi rama principal.

Manejo de erros en el merge de dos ramas:

Cuando se hace un merge entre dos ramas y en las dos se han dado cambios se llega a un área de conflicto, ante esto lo que debemos de hacer es aceptar uno de los dos cambios hechos en las ramas para dejar atrás el conflicto.

Demostración gráfica:

Tenemos dos ramas master y root

```
$ git branch
  master
* root
```

Arrancamos con este código en la rama máster y en la rama root

```python
def recursive(n):
    if n == 1:
        print(f'Number: 1')
        return 1
    else:
        print(f'Number: {n}')
        recursive(n-1)

if __name__ == '__main__':
    recursive(15)
```

Modificamos el código en la rama master:

```python
def recursive(n):
    if n == -1:
        print(f'Number: {n}')
        return -1
    else:
        print(f'Number: {n}')
        recursive(n-1)

if __name__ == '__main__':
    recursive(15)
```

Modificamos el código en la rama root:

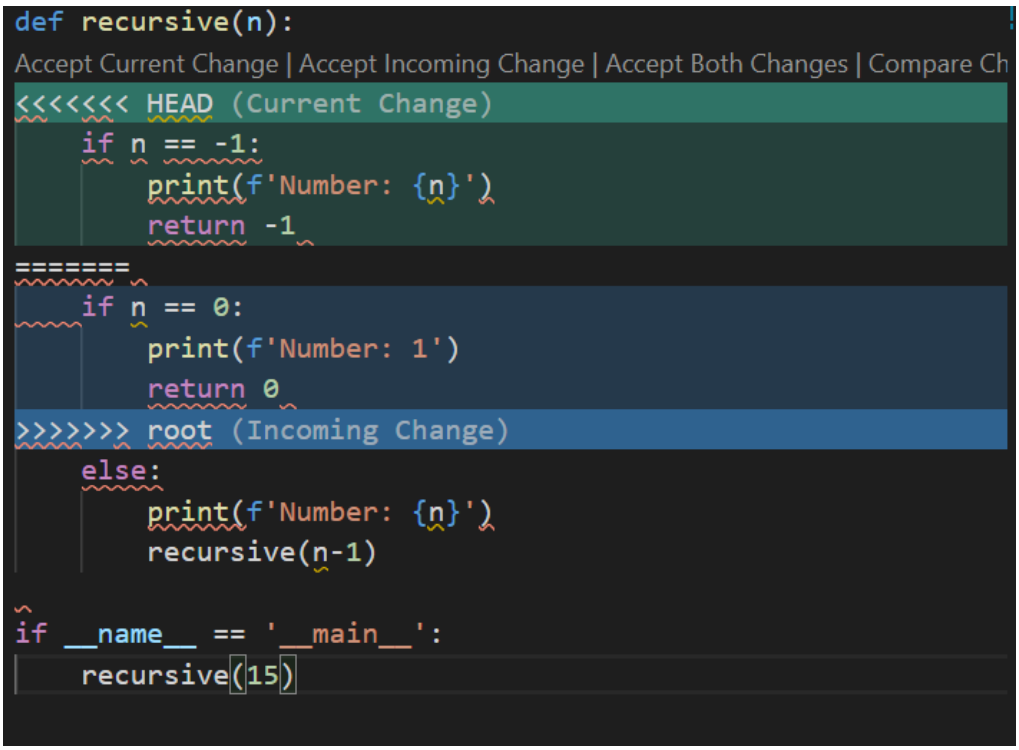
```python
def recursive(n):
    if n == 0:
        print(f'Number: 1')
        return 0
    else:
        print(f'Number: {n}')
        recursive(n-1)

if __name__ == '__main__':
    recursive(15)
```

Hacemos merge de root en master:

```
$ git merge root
Auto-merging code.py
CONFLICT (content): Merge conflict in code.py
Automatic merge failed; fix conflicts and then commit the result.
```

Miremos de cerca el conflicto:



Para quitar este conflicto debemos de aceptar alguno de los dos cambios clickeando cualquiera de los dos botones.

Git branch -M “nombre de la rama”:

Sirve para renombrar la rama principal (master) de git

Uso de GitHub:

Git remote add “nombre de la conexión” “dirección del repo github https”:

Decirle a git que agregue un origen remoto de nuestros archivos, decirle que cree una conexión con el repo de githubo ósea con la url.

Git remote:

Listado de conexiones con github.

Git remote -v:

Ver lista más especifica de conexiones con github.

Git push “nombre conexión” “nombre rama local”:

Antes de hacer un push traer la información del repositorio de GitHub al repositorio local (hacer un pull)

Le envía a la conexión GitHub datos de la rama especificada.

Git pull “nombre conexión” “nombre rama remota”:

Trae la información del repositorio de GitHub a la rama local especificada.

Git pull “nombre conexion” “nombre rama remota” –allow-unrelated-histories:

En caso de tener un error en el pull por diferente historia entre repositorio local y remoto (GitHub) usar esta línea.

Git log --all --graph --decorate –oneline (tree):

Muestra el historial de un repositorio de una manera mas grafica con colores que muestran las ramas y los commits hechos.

Git log --all:

Para ver todos los commit hechos a un repositorio.

Funcionamiento de llaves públicas y privadas:

Llaves públicas y privadas también conocidas como cifrado asimétrico de un solo camino.

La llave publica se conecta a la llave privada.

La llave privada es la que puede descifrar lo que contiene la llave publica por esto el mensaje o cosa que se va a cifrar debe de ser cifrado con la llave publica de a quién se lo quieras mandar para que así el pueda saber cuál es el mensaje.

Configuracion de llaves SSH en local:

Crear entorno de llaves publicas y privadas con git y GitHub

Funciona así:

En tu ordenador se creará una llave publica y una privada.

Le enviaras la llave publica a git y GitHub y le dirás a GitHub para el repositorio “x” quiero que uses la llave publica “y” y nos conectamos por medio de SSH

GitHub te va a enviar cifrado con tu propia llave publica su llave publica (la de GitHub)

Las llaves SSH son por persona, no por repo ni por proyecto

Comando para llaves SSH:

ssh-keygen -t rsa -b 4096 -C “puertasantiago2002 @gmail.com”

Revisar que el servidor SSH comprobación este prendido:

eval $(ssh-agent -s)

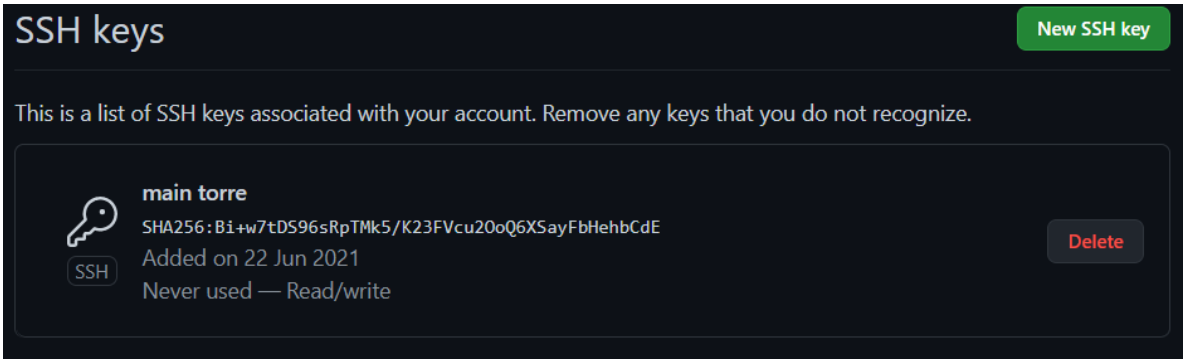
```
$ eval $(ssh-agent -s)
Agent pid 1347
```

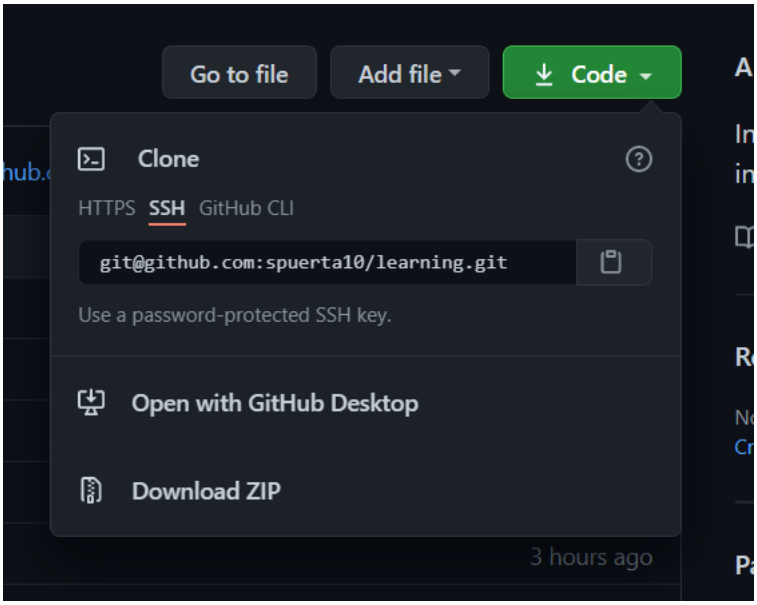
Agregar la llave al sistema:

```
Admin@DESKTOP-40NS536 MINGW64 ~
$ ssh -add ~/.ssh/id_rsa
```

Conectando GitHub por SSH:

Ir a GitHub settings, SSH and GPG keys, new SSH key, y agregar la clave publica SSH





Cambiar url de conexión remota (git remote) por la obtenida de SSH:

Git remote set-url “nombre conexión remota” “url SSH”

Ejecutar git remote -v, y ver que ya cambio la URL de conexión.

Tags y versiones:

Los tags son etiquetas las cuales te permiten identificar y marcar ciertos puntos del historial de tu repositorio (commits)

Git tag -a “nombre de tag” -m “mensaje” “ID Commit”:

Nos deja crear nuevos tags

Git tag:

Lista de todos los tags

Git show-ref --tags:

Forma de saber a que commit está conectado un tag.

```
$ git show-ref --tags
f4d813d2ccb2842b621b7327f49709aa7b360be0 refs/tags/learning_v0.1
```

Git push “nombre de la conexión” --tags:

Enviar tags a GitHub

Git tag -d “nombre tag”:

Borrar tags no deseados

Git push “nombre de la conexión” :refs/tags/“nombre del tag”:

Primero hay que borra el tag en el repositorio local antes de borrarlo en el repositorio remoto

Borrar un tag en GitHub.

Manejo de ramas en GitHub:

Git show-branch:

Ramas que hay y su historia (commits)

Git show-branch –all:

Todas las ramas que hay y su historia.

Gitk:

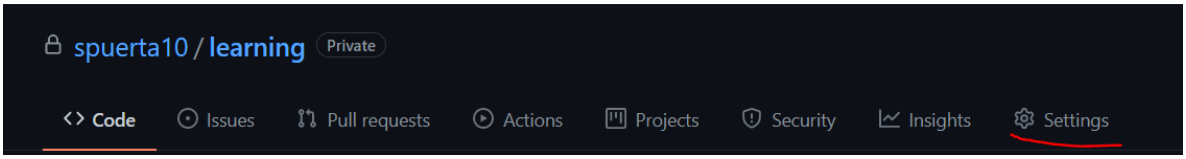
Software de git para visualización de historia de ramas.

Git clone “dirección del repositorio GitHub”:

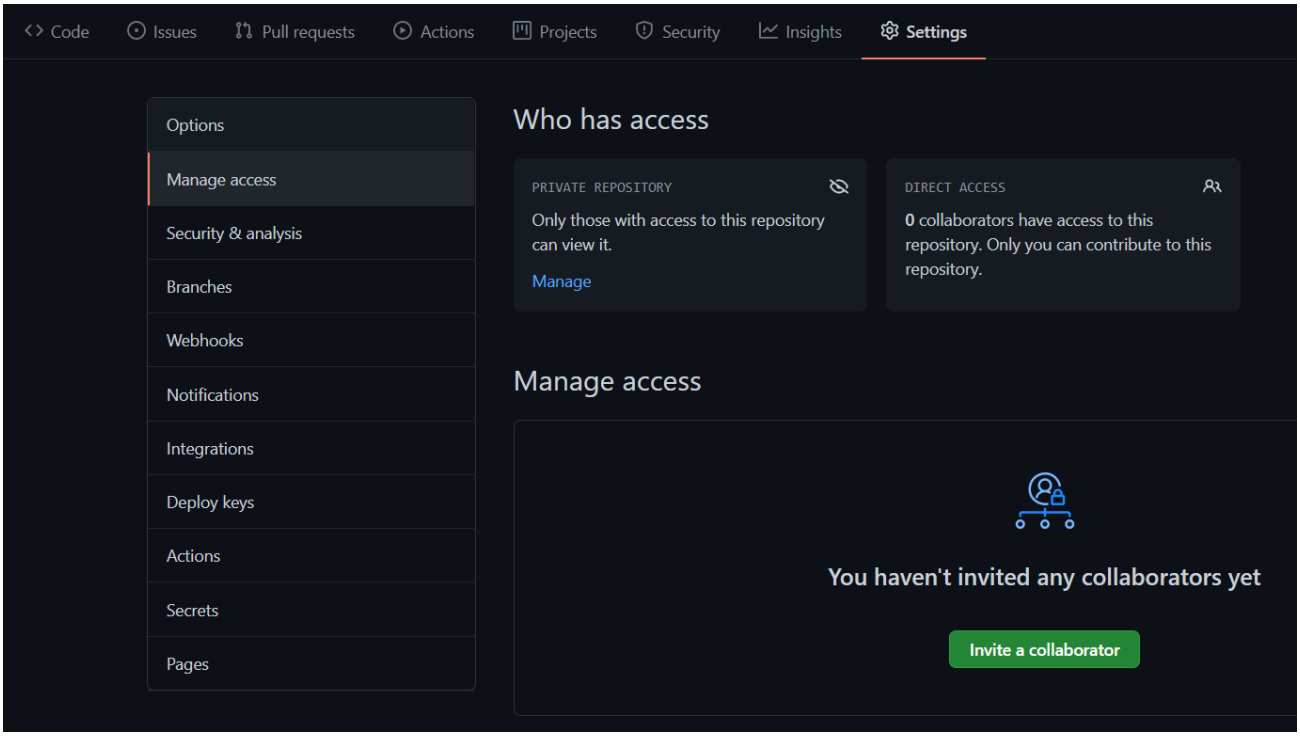
Permite clonar un repositorio publico GitHub.

Agregar colaboradores a un proyecto en GitHub:

Ir a las settings en el repositorio



Ir a Manage access



Añadir el correo o nombre de usuario de la cuenta de GitHub del colaborador.

Pull Requests:

Estado antes del merge para poder hacer una revisión de los cambios hechos, si estos gustan el pull request es aprobado y se hace el merge; esto es una característica exclusiva de GitHub, también existe en GitLab pero este se llama merge request.

Forks o Bifurcaciones:

Clonar el estado actual de un repositorio ajeno, para estar al día hay que actualizar el fork

Gir rebase “nombre rama”:

¡Solo se debe de implementar en repositorios locales! Pegar una rama a otra, es como un merge solo que el rebase no deja rastro de que la rama pegada existió cambia la historia de lo logs y commits hechos.

Primero rebase a la rama que se quiere desaparecer y luego rebase a la rama main.

Git stash:

Stash nos sirve para cuando tenemos cambios y queremos cambiar de rama, pero sin hacerle comimt a esos cambios

Git stash save “mensaje para identificar stash”:

Guardar un stash con un mensaje para poder identificarlo luego.

Git stash list:

Sirve para ver la lista de cambios WIP (Work in progress) que hay en el stash.

Git stash pop:

Sirve para volver a los cambios guardados en el stash.

Git stash branch “nombre rama”:

Crea una rama con el cambio del ultimo stash de la stash list.

Git stash branch “nombre rama” stash@{<num_stash>}:

Crea una rama con el cambio de un stash especifico de la stash list.

Git stash drop:

Sirve para eliminar el ultimo stash añadido a la stash list.

git stash drop stash@{<num_stash>}:

Sirve para eliminar un stash en específico de la stash list.

Git stash clear:

Sirve para eliminar todos los stash de la stash list.

Git clean -f:

Sirve para borrar archivos no deseados, por ejemplo, copias de archivos; esto se debe de hacer antes del add y mucho antes del commit.

Git clean --dry-run:

Sirve para simular, para ver lo que se va a borrar sin borrarlo.

Git cherry-pick “commitID”:

Sirve para traer un commit especifico de otras ramas

Git log –oneline:

Ver más grande el log de git.

Git commit –amend:

Si por casualidad olvidé hacer un cambio y le di el commit.

Primero dar add a los cambios hechos

Luego dar git commit –amend, esto lo que hace es pegar los cambios que agregué con add al último commit que hice.

Git branch -D “nombre rama”:

Borrar una rama y su contenido.

Git reflog:

Se ve toda la historia transcurrida de una rama.

¡Usar solo en caso de EMERGENCIA!

Git reset –SOFT “numero de header”:

Mantiene lo que haya en staging, a lo que se le haya puesto el add.

Git reset –HARD “numero de header”:

No mantiene lo que haya en staging vuelve drásticamente al header anterior.

Git grep “palabra a buscar”:

Nos retorna en que archivo hemos usado esa palabra.

Git grep -n “palabra a buscar”:

Retorna la línea en la que aparece la palabra a buscar dentro de nuestros archivos.

Git grep -c “palabra a buscar”:

Retorna cuantas veces hemos usado la palabra a buscar y en que archivos.

Git log -S “palabra a buscar”:

Sirve para encontrar la palabra a buscar en el historial de los commits.

Git config –global alias.“nombre alias” “comando”:

Crea un alias local de un comando de git

Se invoca así: git “alias”

git shortlog -sn = muestra cuantos commit han hecho cada miembro del equipo.

git shortlog -sn --all = muestra cuantos commit han hecho cada miembro del equipo hasta los que han sido eliminado

git shortlog -sn --all --no-merge = muestra cuantos commit han hecho cada miembro quitando los eliminados sin los merges

git blame -c ARCHIVO = muestra quien hizo cada cosa linea por linea

git COMANDO --help = muestra cómo funciona el comando.

git blame ARCHIVO -Llinea_inicial,linea_final -c = muestra quien hizo cada cosa linea por linea indicándole desde que linea ver ejemplo -L35,50

****git branch -r **** = se muestran todas las ramas remotas

git branch -a = se muestran todas las ramas tanto locales como remotas