Contenido

[**GIT** 2](#_Toc179204835)

[**GIT DIFF** 3](#_Toc179204836)

[**MODIFICAR Y DESHACER COMMITS** 3](#_Toc179204837)

[**RAMAS/BRANCHES** 5](#_Toc179204838)

[**FUCIONAR RAMA/MERGE + MERGE CONFLICTS** 6](#_Toc179204839)

[**.GITIGNORE** 7](#_Toc179204840)

[**ALIAS** 9](#_Toc179204841)

[**GIT REFLOG** 10](#_Toc179204842)

[**GIT HUB** 11](#_Toc179204843)

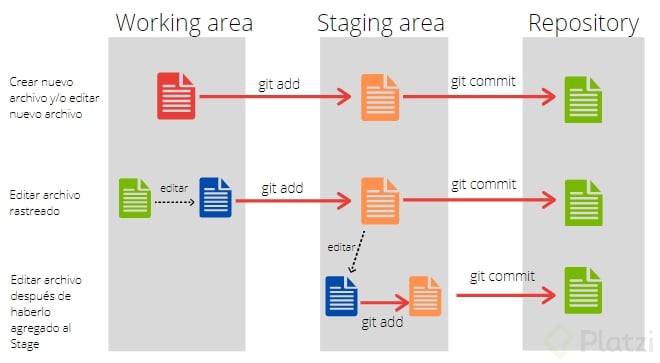
[**GIT CLONE** 11](#_Toc179204844)

[**GIT PUSH** 12](#_Toc179204845)

[**GIT PULL** 13](#_Toc179204846)

[**GIT FETCH** 15](#_Toc179204847)

# **GIT**

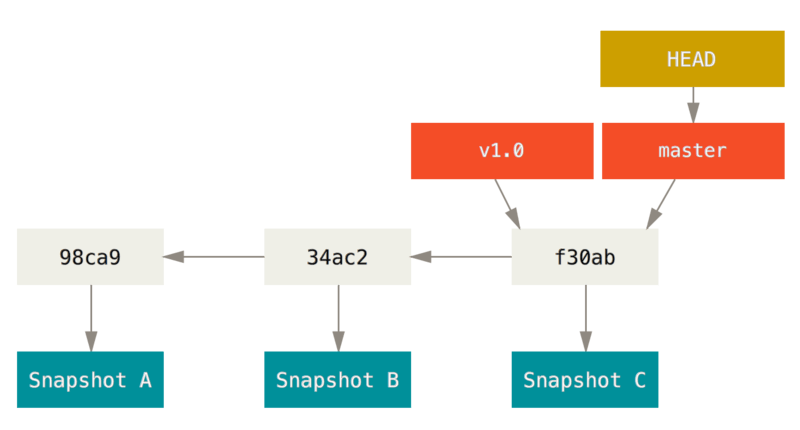


* mkdir 🡺 Crear directorio
* toutch 🡺 Crear archivo
* rmdir 🡺 Remover directorio
* git init 🡺 Crear repositorio
* pwd 🡺 Mostrar ruta
* ls 🡺 Muestra el contenido del directorio
* ls -a 🡺 Muestra archivos ocultos del directorio
* git add + nombre del archivo 🡺 Enviamos el archivo al área de preparación (Staging área)
* git rm -- cached + nombre del archivo 🡺 remover archivo
* git status 🡺 Muestra información sobre nuestro directorio de trabajo y el área de preparación
* git status -s 🡺 Muestra información sobre nuestro directorio de trabajo y el área de preparación abreviada
* git commit--m 🡺 Guarda los cambios, sube lo del área de preparación al repositorio con un mensaje.
* git commit -a 🡺 Sube los cambios al repositorio SIN PASAR POR EL AREA DE STAGING
* git restore 🡺 Restaura el archivo eliminado del área de preparación
* git checkout 🡺 Volver al último commit, esto funciona en caso de que NO esté en área de preparación
* git reset --hard 🡺 Reseteo forzado al último commit, descarta los cambios en el repositorio y en el área de preparación
* git mv + nombre del archivo + nuevo nombre 🡺 renombrar archivo
* git show + nombre de archivo 🡺 Se utiliza para ver detalles ampliados de objetos Git, como blobs, árboles, etiquetas y confirmaciones, si está en el área de commit.
* git log 🡺 Nos muestra la identificación de los commit, fecha de modificación y quien lo modificó
* git log --oneline 🡺 Nos muestra la identificación de los commit, fecha de modificación y quien lo modificó de forma abreviada
* git log --oneline --all --graph 🡺 Nos muestra la identificación de los commit,en forma de grafica
* git log --oneline --all --graph --pretty=format:"%C(auto)%h%d %s %C(black)%C(bold)%cr" 🡺 Muestra hace cuanto tiempo se hizo el commit, agrega un color distintivo
* git config --global core.abbrev + número de caracteres 🡺 Cambiar la cantidad de caracteres a mostrar del ID del commit

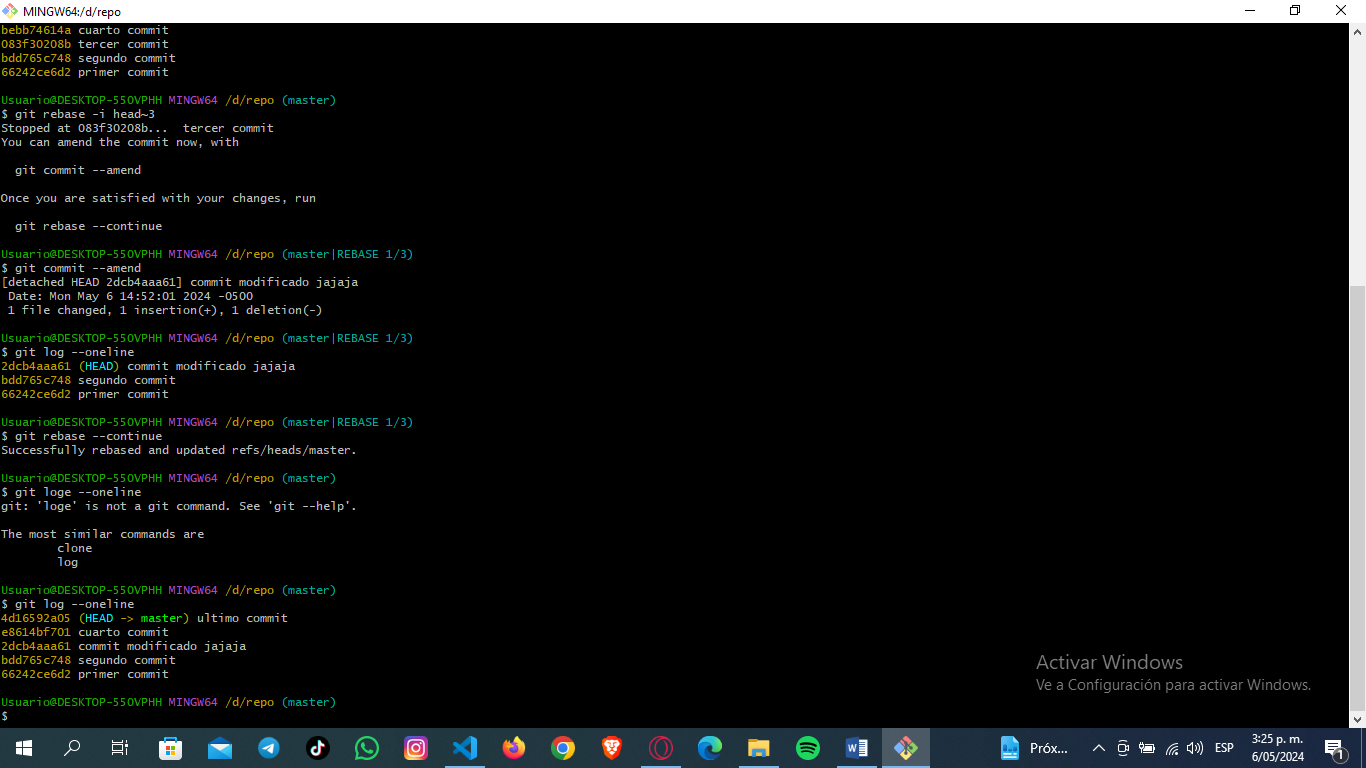
## **GIT DIFF**

* git diff --staged 🡺 Saber la diferencia entre lo que está el área de preparación y el área de commit
* git diff + ID commit + ID commit a comparar 🡺 Comparar commit
* git diff --name-only + ID commit + ID commit a comparar 🡺 Compara solo el nombre de los archivos que cambiaron
* $ git diff --word-diff + ID commit + ID commit a comparar 🡺 Compara las líneas de los archivos que cambiaron

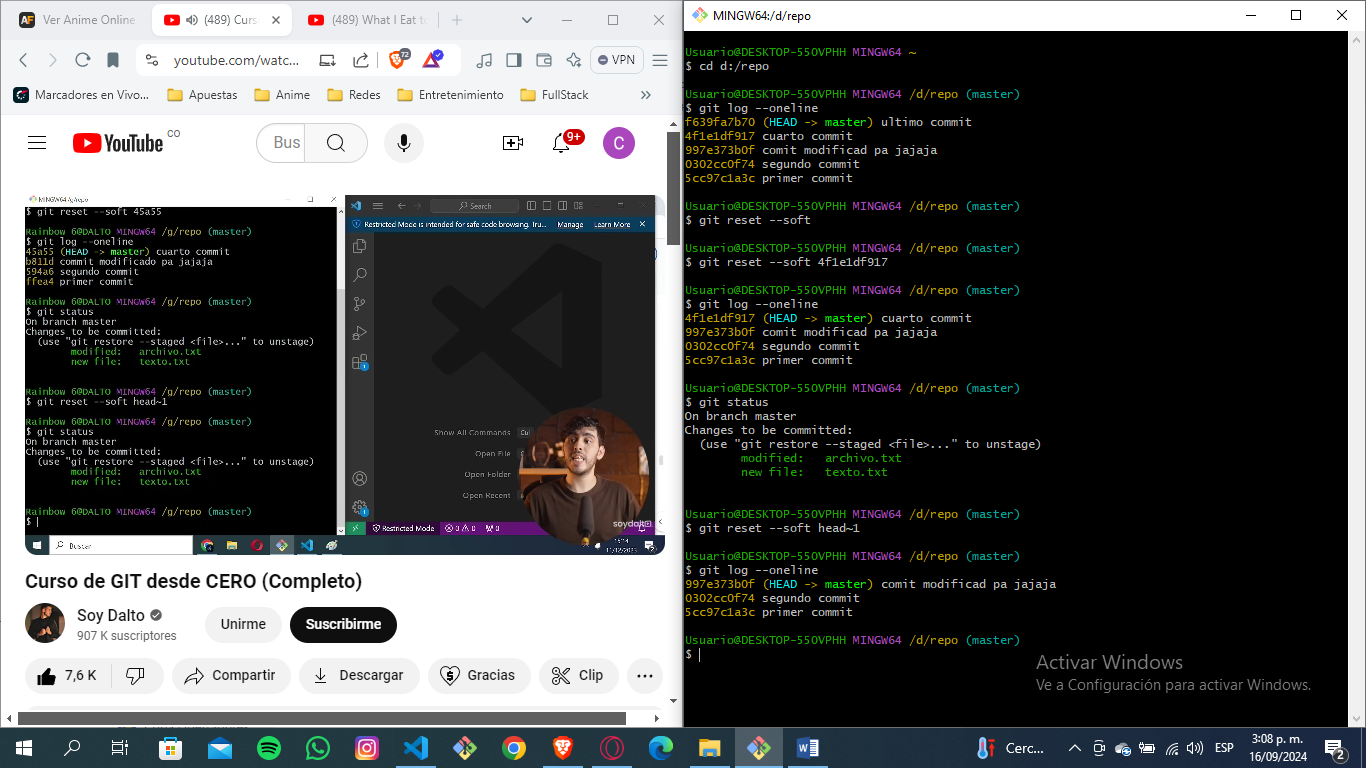
## **MODIFICAR Y DESHACER COMMITS**



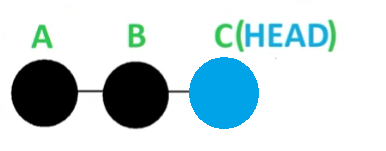
* git commit --amend 🡺 El comando git commit --amend es una forma conveniente de modificar la confirmación más reciente. Le permite combinar cambios preparados con la confirmación anterior en lugar de crear una confirmación completamente nueva. Al final cambia el id del commit
* git rebase -i head~ + número de commit a modificar (hacia atrás) 🡺 Te posicionas en el commit que quieres modificar, y se eliminan los que están por delante
* git rebase --continue 🡺 Después de modificar un commit, con este comando devuelves los commits eliminados



* git reset --soft + ID Commit 🡺 Vuelve al commit indicado
* git reset --soft head ~ + número de posición 🡺 Mueve el head a la posición que queremos



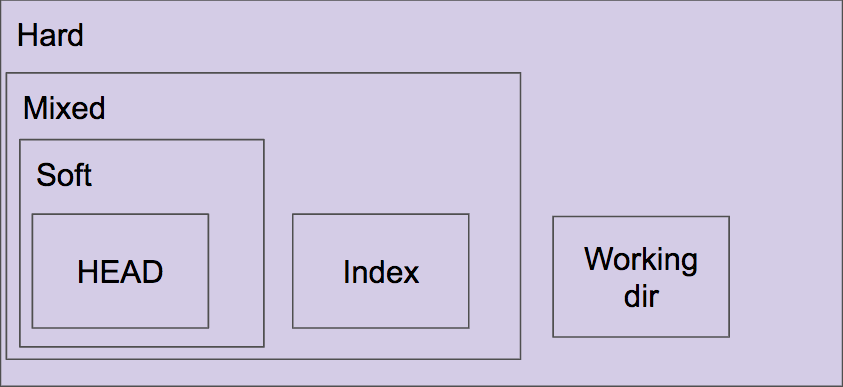
Dado: - A - B - C (maestro)



Para aquellos que usan Terminal con el color activado (git config -

color global.ui automático):

* git reset --soft A 🡺 Verás las cosas de B y C en verde (preparadas y listas para enviarse (git commit))
* git reset --mixed A (o git reset A) 🡺 Verás el material de B y C en rojo (sin preparar y listo para ser preparado (git add) y luego confirmado (git commit) )
* git reset -- hard A 🡺 Ya no verás los cambios de B y C en ninguna parte (será como si nunca hubieran existido)

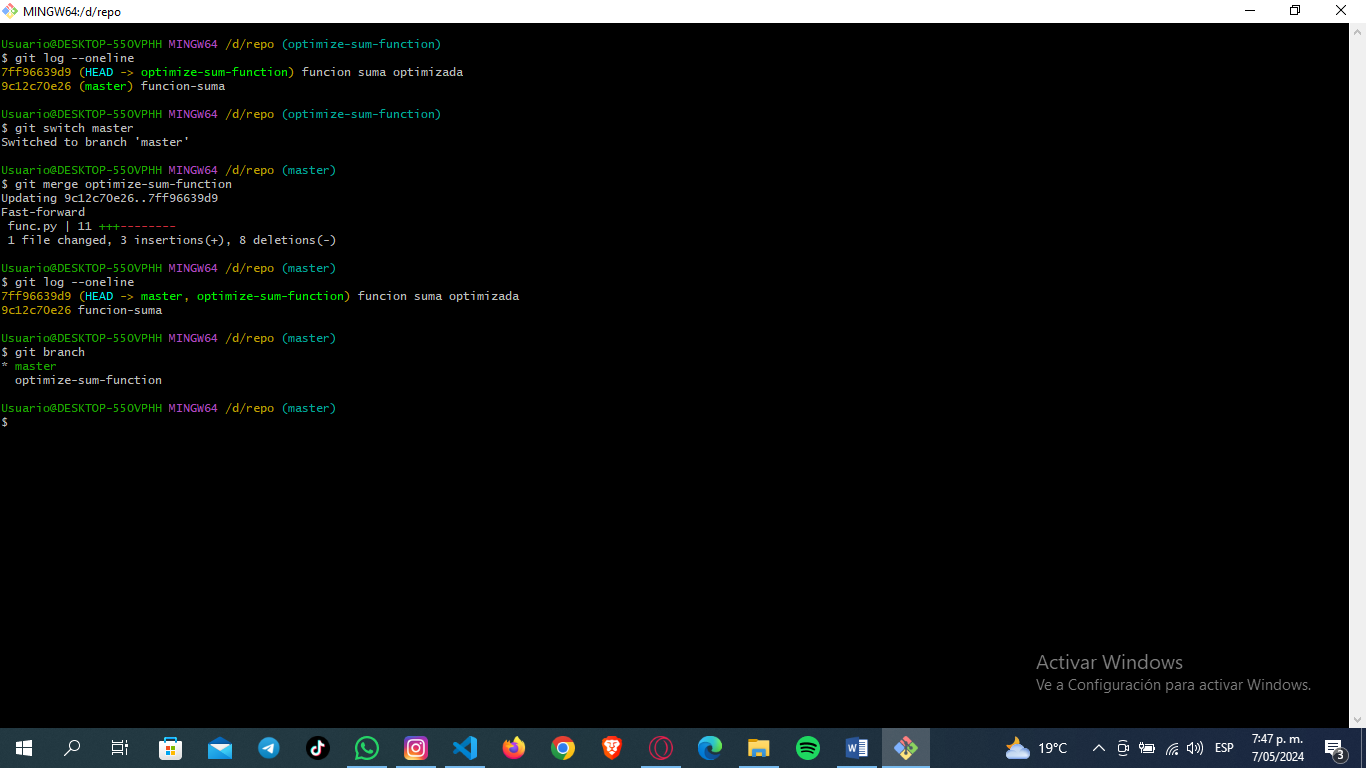


## **RAMAS/BRANCHES**

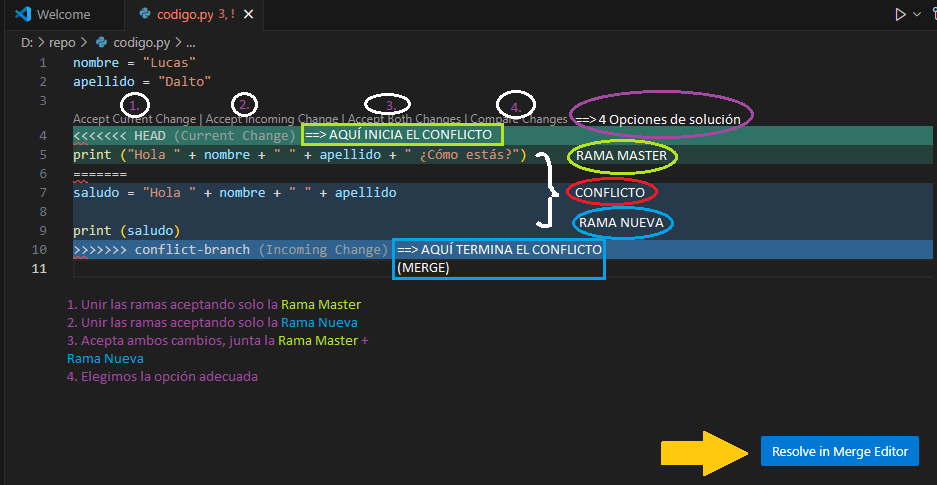
* git branch 🡺 Muestra las ramas creadas
* git branch + nombre de rama (kebab case) 🡺 Crea la rama (una copia de la rama master) con el nombre. Ejemplo, git branch modificar-dev
* git switch + nombre de rama 🡺 Se ubica en la rama que nombramos
* git switch -c + nombre de rama 🡺 Crea y se ubica en la rama que nombramos, desde la rama que estamos
* git branch -d + nombre de rama 🡺 Eliminar rama. Ojo: hay que estar fuera de la rama
* git branch -m + nombre de rama + Nuevo nombre 🡺 Modificar el nombre de una rama externa
* git branch -m + nuevo nombre 🡺 Modificar el nombre de la rama actual

## **FUCIONAR RAMA/MERGE + MERGE CONFLICTS**

* git merge + nombre de la rama a fucionar 🡺 Fuciono los commits de una rama con otra. Ojo, debo estar en la rama que quiero fucionar



* git log --oneline --all 🡺 Nos muestra los commit de todas las ramas



## **.GITIGNORE**

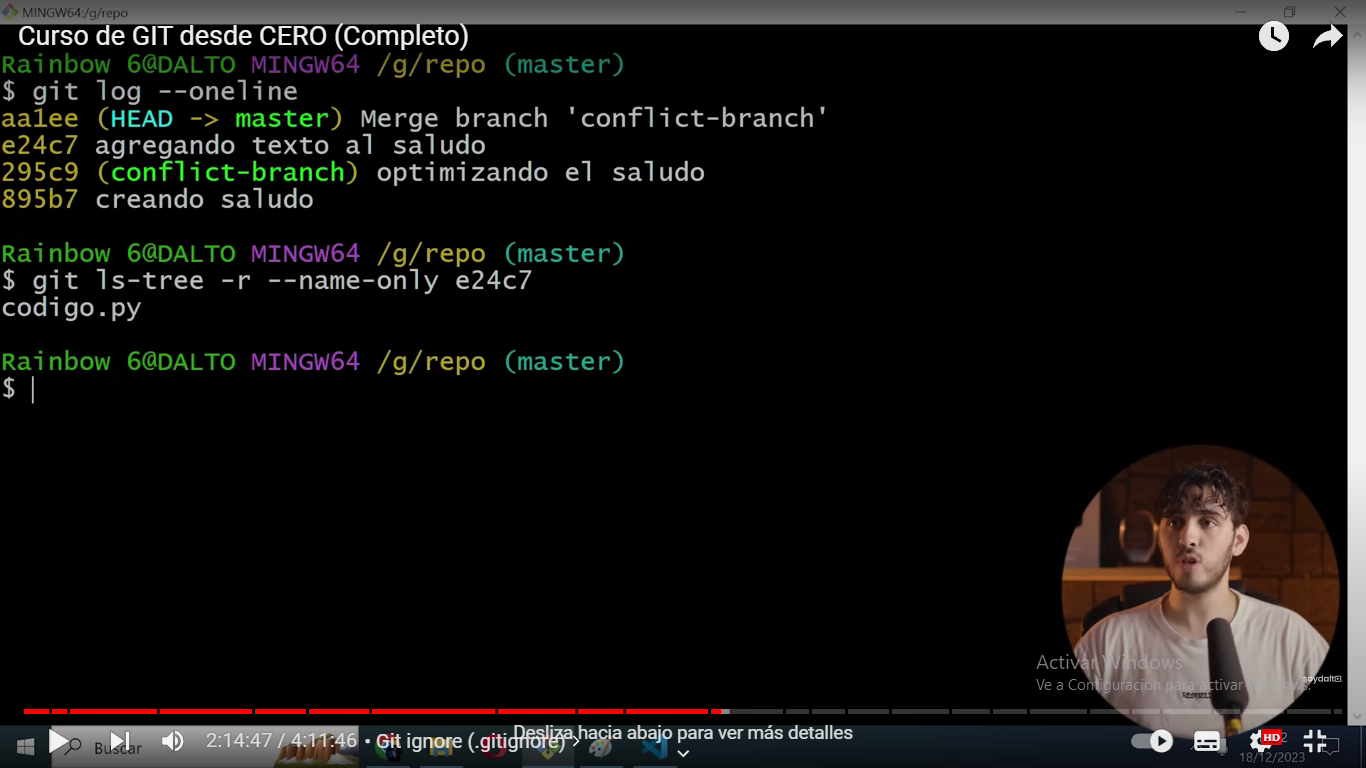
* Git ve cada archivo de tu copia de trabajo de una de las siguientes maneras:
  1. Con seguimiento: Un archivo que se ha preparado o confirmado previamente.
  2. Sin seguimiento: Un archivo que no se ha preparado o confirmado.
  3. Ignorado: Un archivo que se le ha indicado explícitamente a Git que ignore.

Los archivos ignorados suelen ser artefactos de compilación y archivos generados por el equipo que pueden derivarse de tu fuente de repositorios o que no deberían confirmarse por algún otro motivo. Estos son algunos ejemplos habituales:

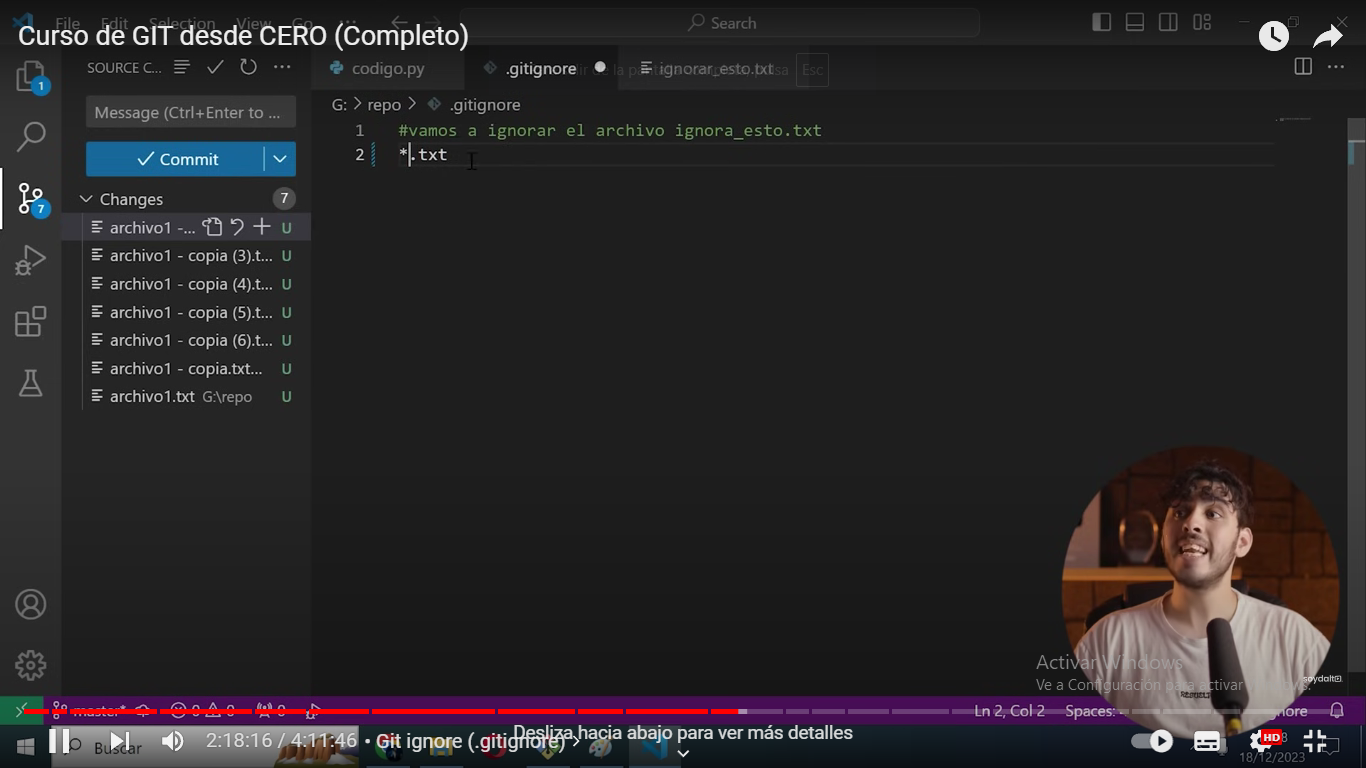
* Cachés de dependencias, como es el caso del contenido de /node\_modules o /packages.
* Código compilado como, por ejemplo, los archivos .o, .pyc y .class.
* Directorios de salida de compilación, como es el caso de /bin, /out o /target.
* Archivos generados en tiempo de ejecución como, por ejemplo, .log, .lock o .tmp.
* Archivos ocultos del sistema, como es el caso de .DS\_Store o Thumbs.db.
* Archivos personales de configuración de IDE como, por ejemplo, .idea/workspace.xml.

A los archivos ignorados se les hace un seguimiento en un archivo especial llamado .gitignore que se incorpora en la raíz de tu repositorio. En Git no hay ningún comando explícito para ignorar archivos: en su lugar, cuando tengas nuevos archivos que quieras ignorar, deberás editar y confirmar manualmente el archivo .gitignore. Los archivos .gitignore contienen patrones que establecen coincidencias con los nombres de archivo de tu repositorio para determinar si deberían ignorarse o no.

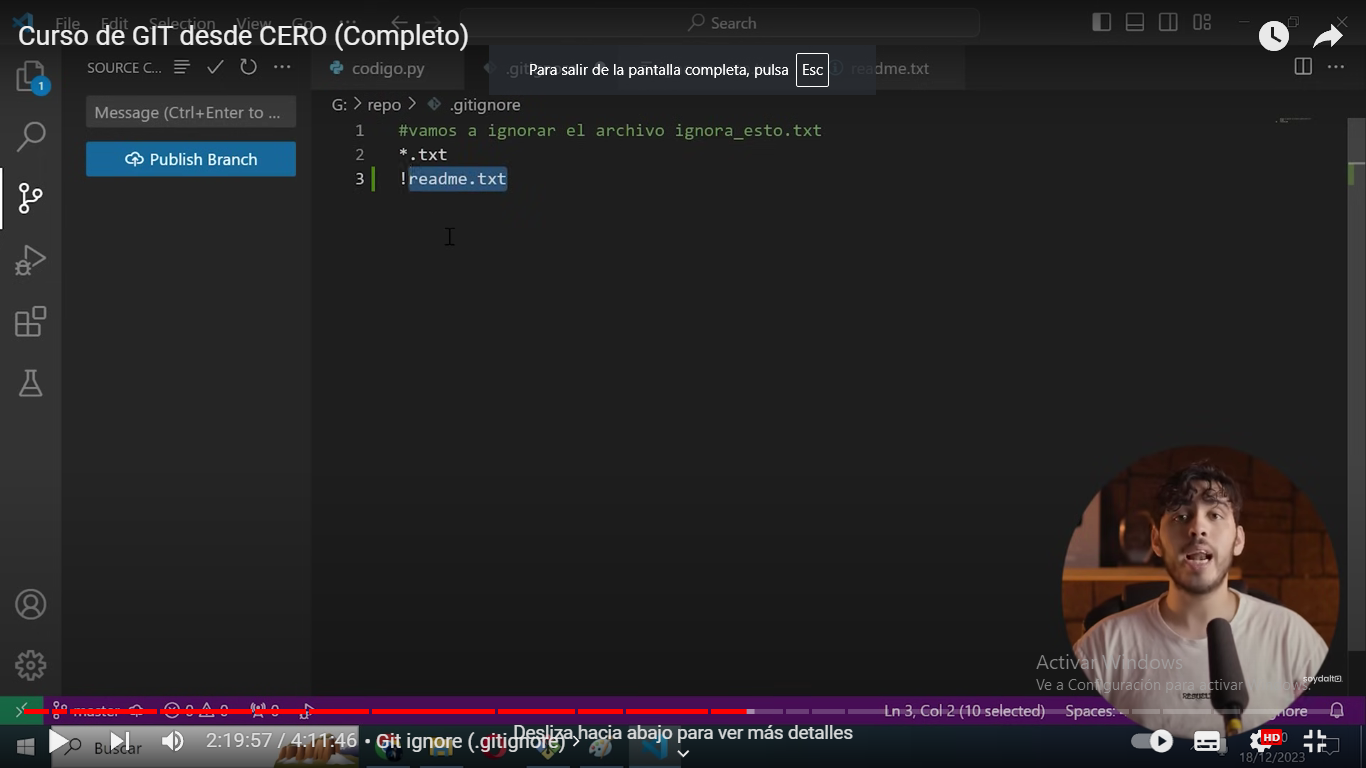
* Patrones del .gitignore 🡺 <https://www.atlassian.com/es/git/tutorials/saving-changes/gitignore#git-ignore-patterns>
* git ls-tree -r --name-only (para que solo muestre el nombre) + ID del commit 🡺 Nos muestra todos los archivos y directorios que tiene un commit especifico



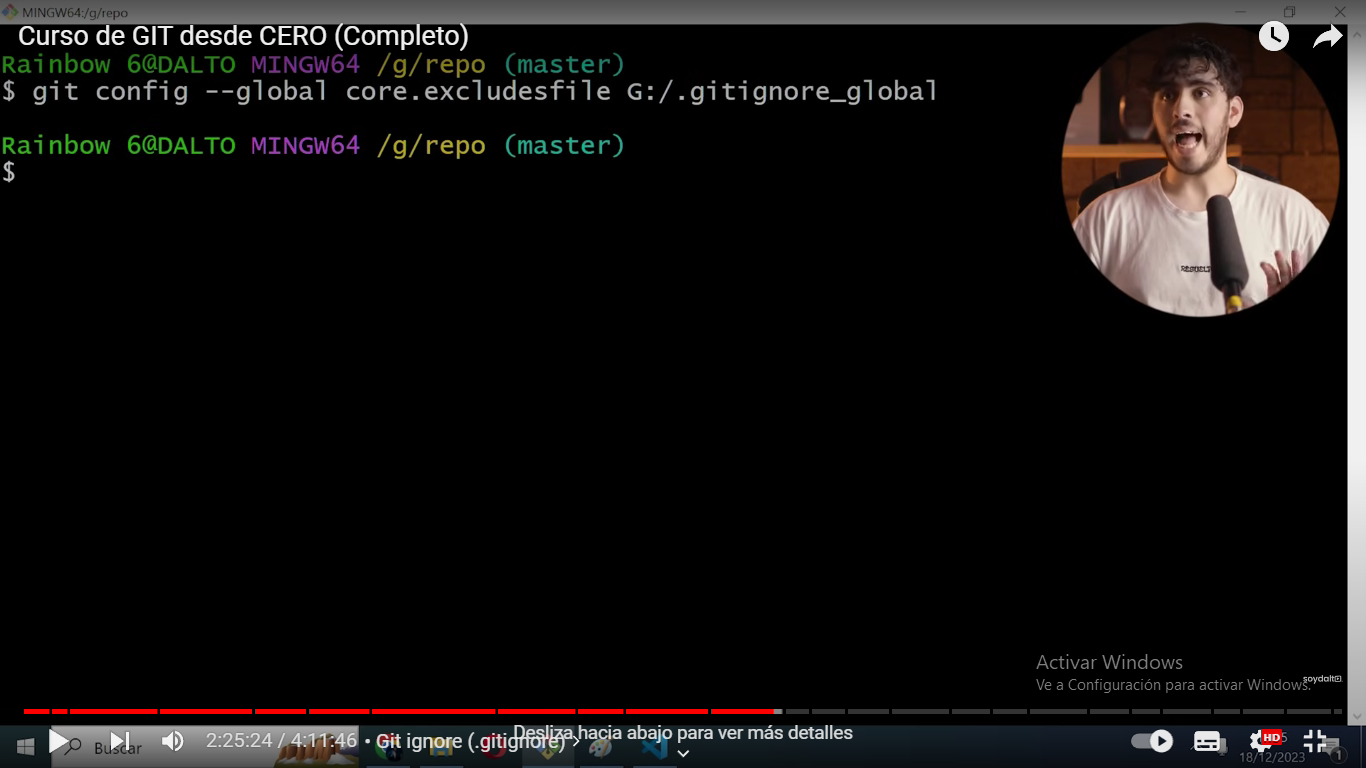
* \*.txt 🡺 Con el asterisk (\*) se ignora todos los archivos con un mismo formato



* !readme.txt 🡺 Con el asigno de admiracion (!) se DESignora el archive



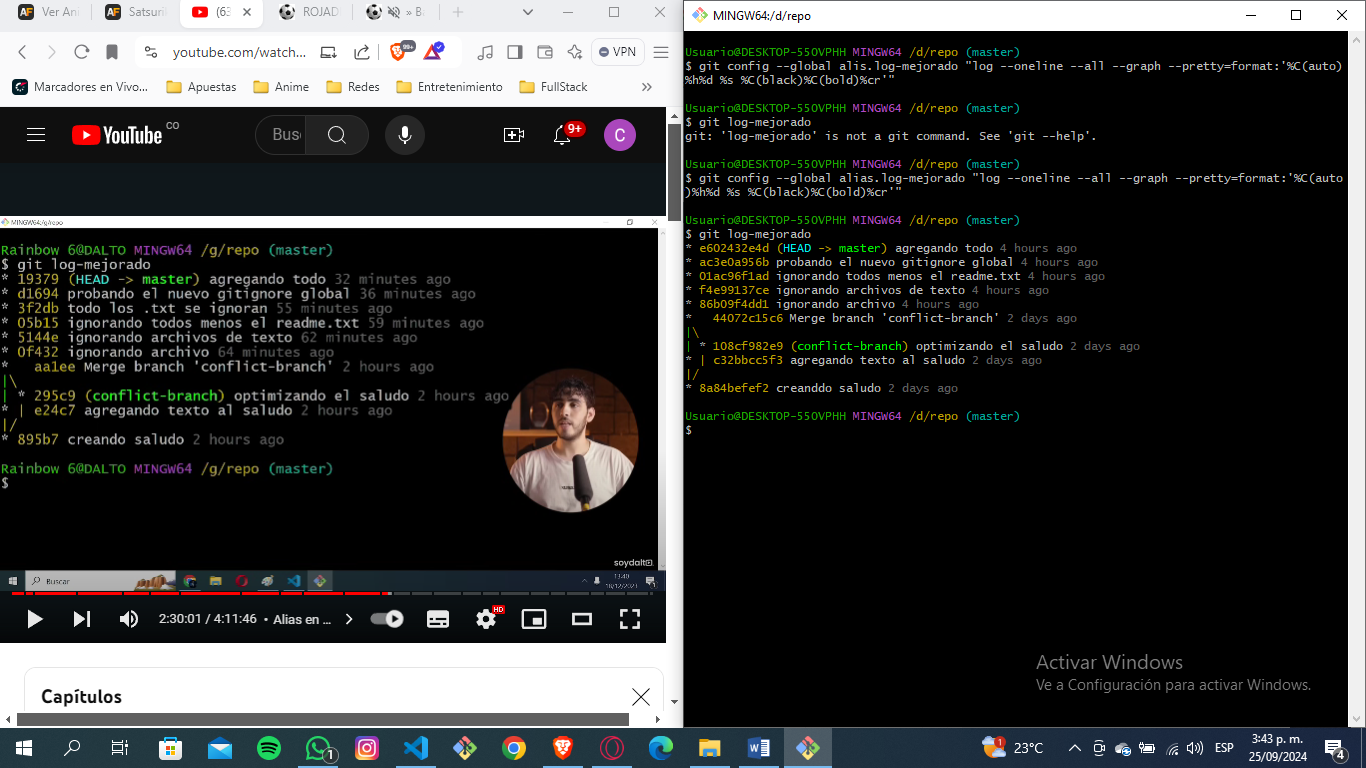
* Configurar un archivo .gitignore global



* Si queremos que un repositorio no este dentro del gitignore\_global, creamos un gitignore en el area local, teniendo como prioridad esta area

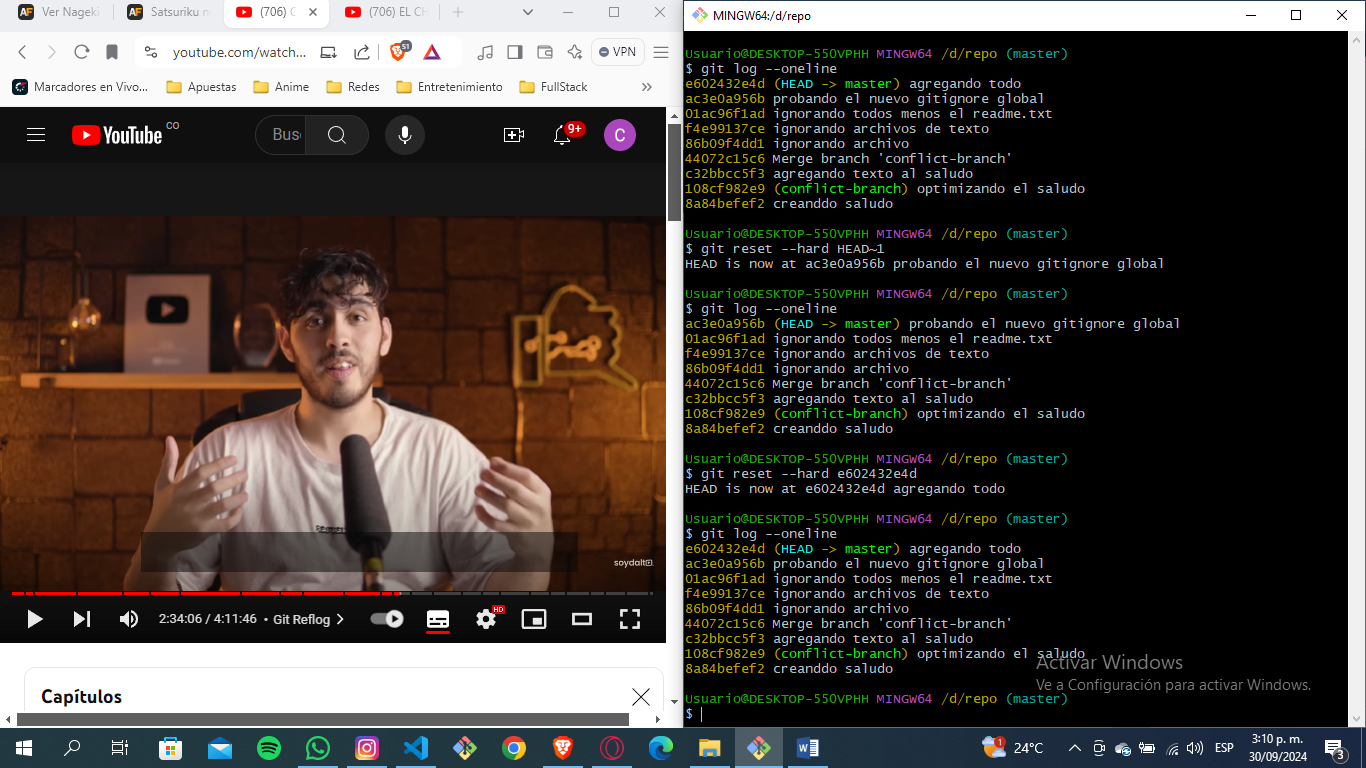
## **ALIAS**

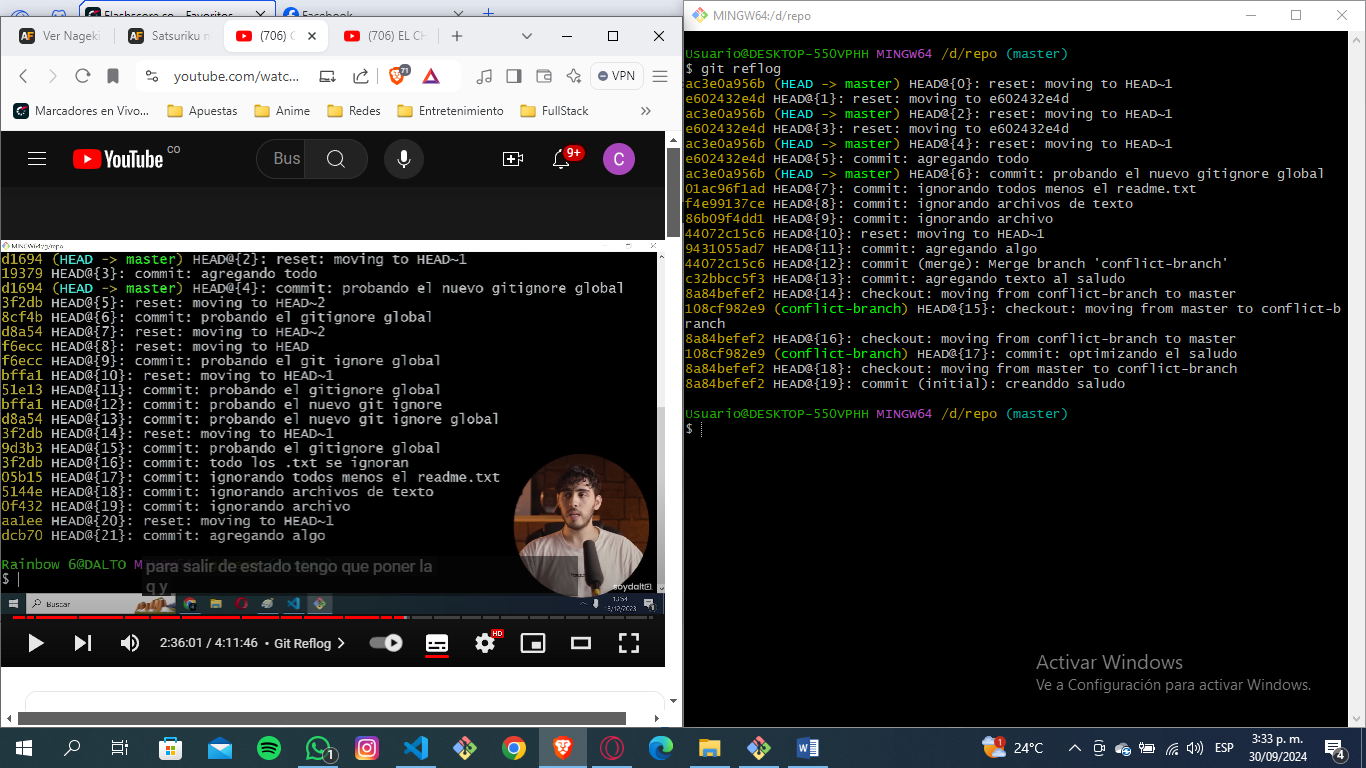
* Los alias se usan para crear comandos más cortos que se asignan a comandos más largos. Dado que obligan a pulsar menos teclas para ejecutar un comando, aumentan la eficiencia de los flujos de trabajo. Pongamos un breve ejemplo tomando en consideración el comando git checkout .



## **GIT REFLOG**

Git reflog es un almacén aislado que se utiliza para mantener un historial preciso de las modificaciones realizadas en el puntero HEAD de su repositorio. No se ve afectado por operaciones destructivas como “git reset —hard”. Además, git reflog se puede utilizar para recuperar confirmaciones o ramas perdidas en el repositorio.

* Eliminamos la referencia del commit git reset --hard HEAD~1 y luego recuperamos el mismo con git reset --hard + ID del commit eliminado
* ¿Que pasa si no tenemos la referencia del commit? 🡺 usamos Git Reflog

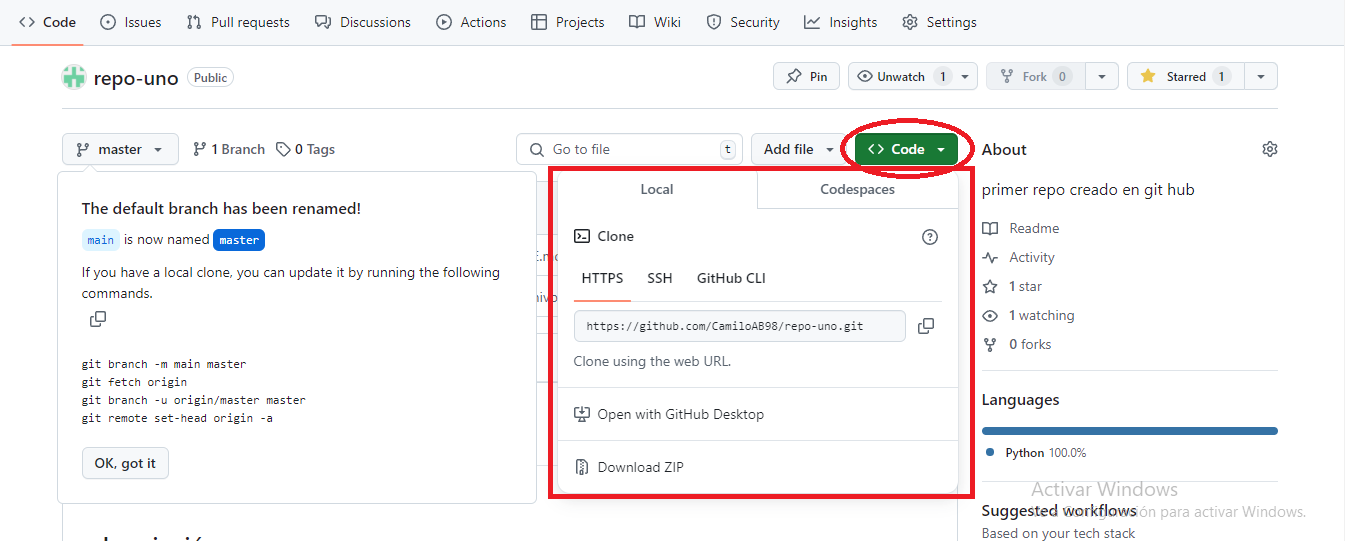


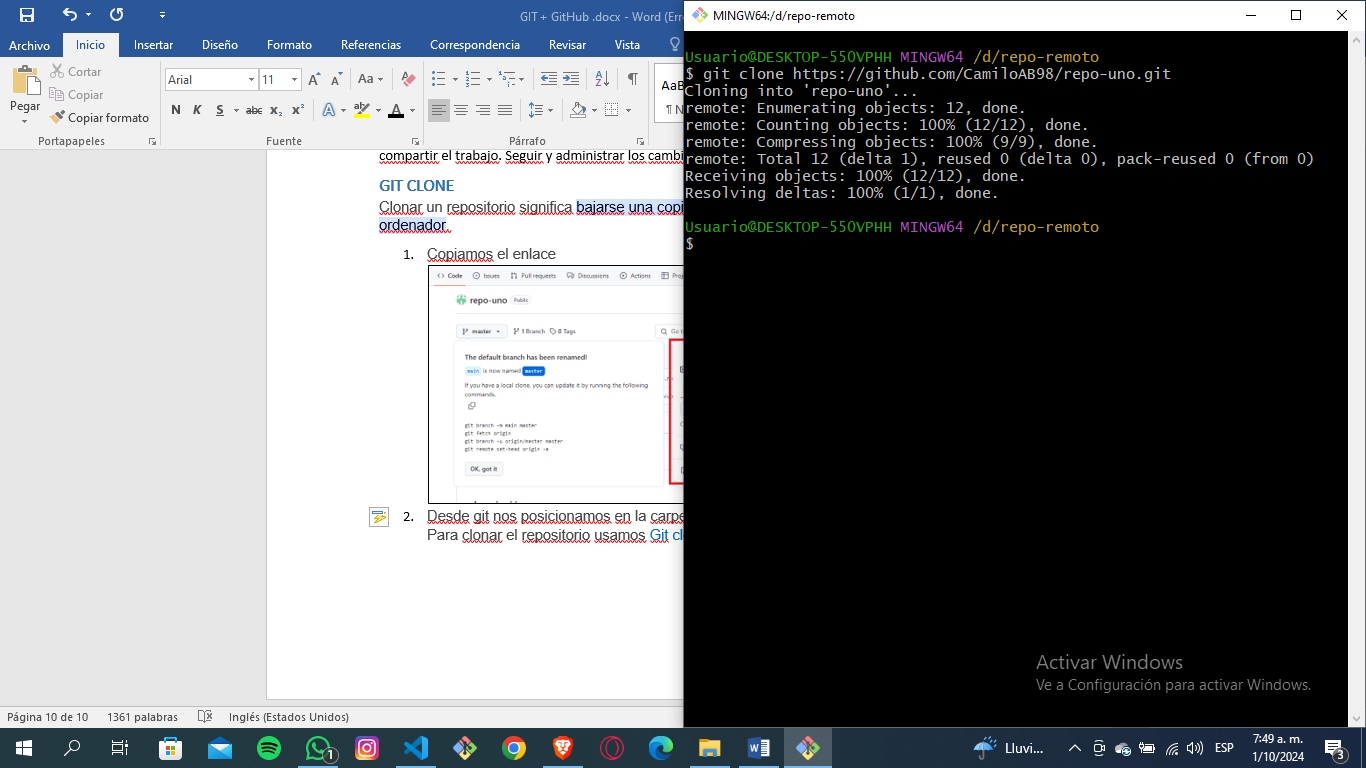
# **GIT HUB**

GitHub es una plataforma donde puedes almacenar, compartir y trabajar junto con otros usuarios para escribir código. Almacenar tu código en un "repositorio" en GitHub te permite: Presentar o compartir el trabajo. Seguir y administrar los cambios en el código a lo largo del tiempo.

## **GIT CLONE**

Clonar un repositorio significa bajarse una copia completa del mismo a nuestro a ordenador.

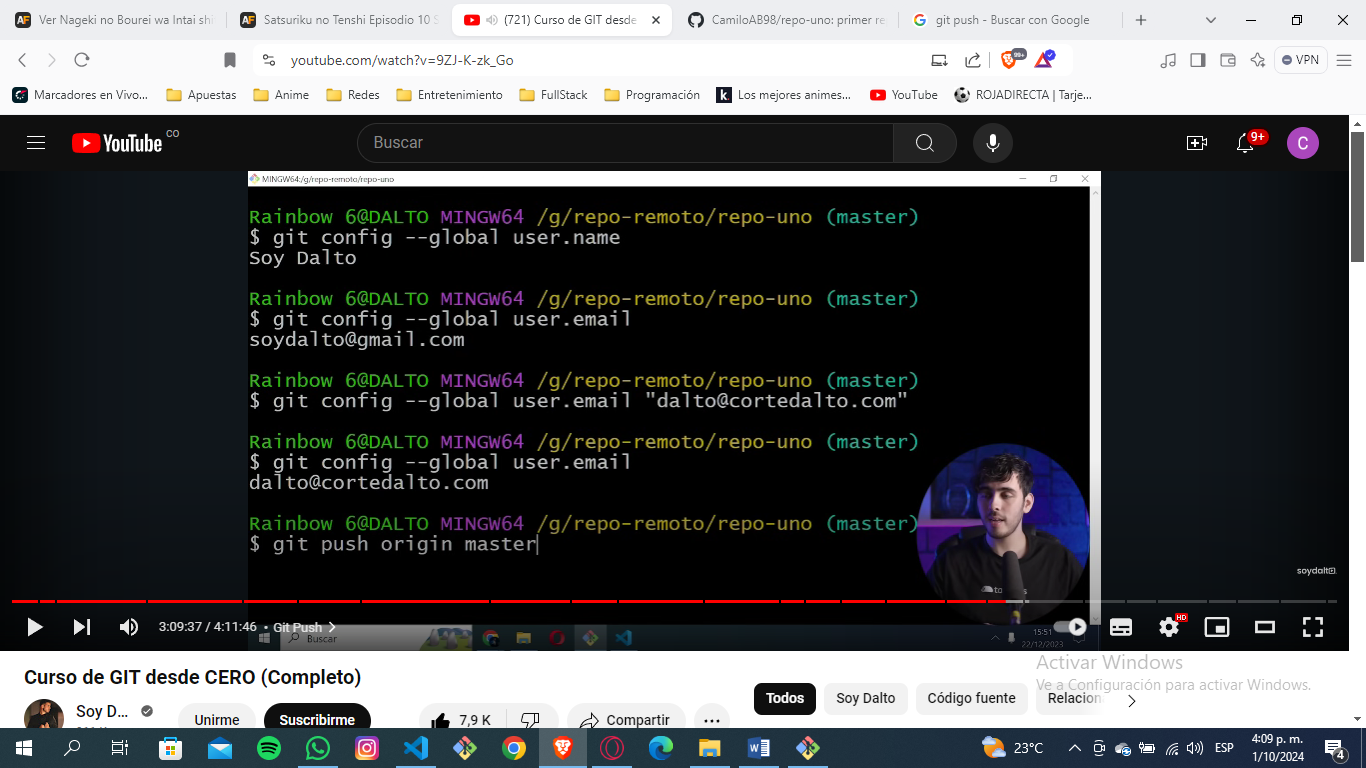
1. Copiamos el enlace
2. Desde git nos posicionamos en la carpeta donde vamos a clonar el repositorio. Para clonar el repositorio usamos Git clone + URL del repositorio



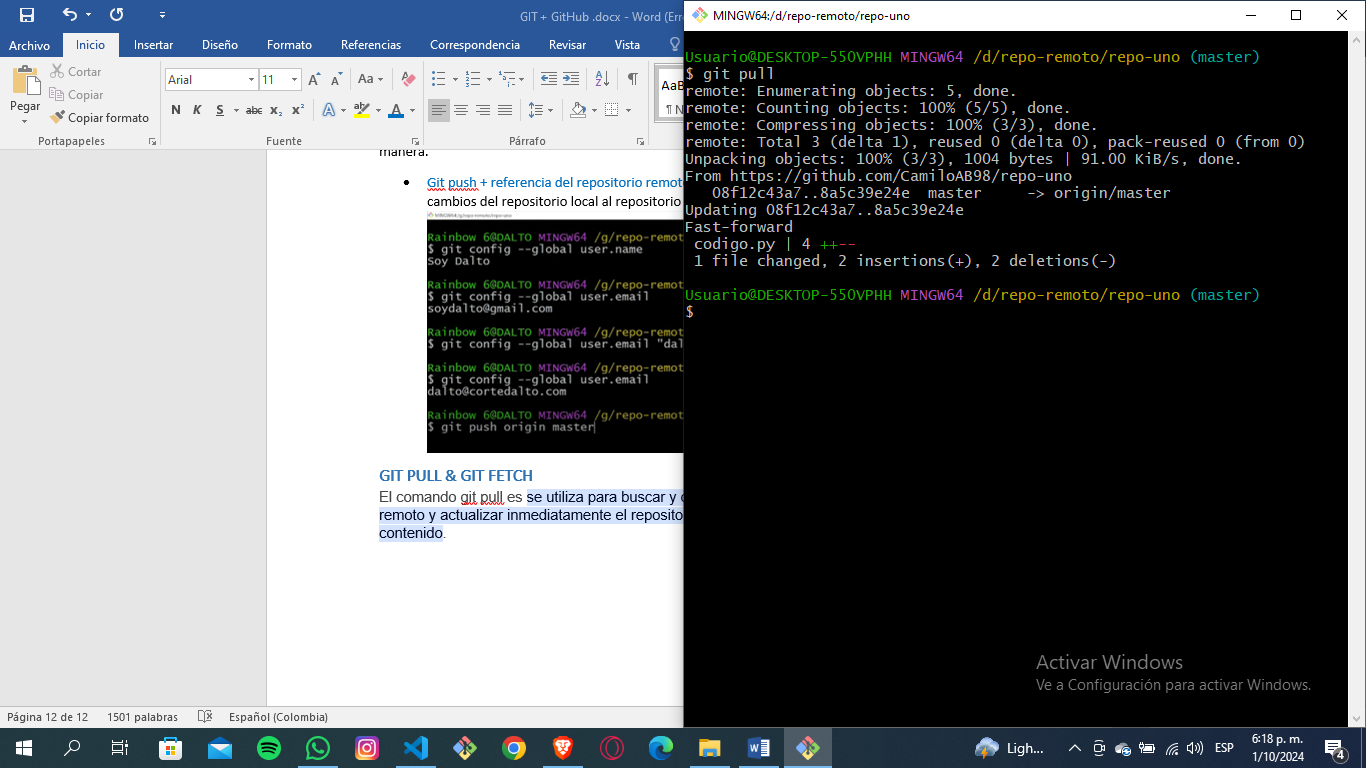
## **GIT PUSH**

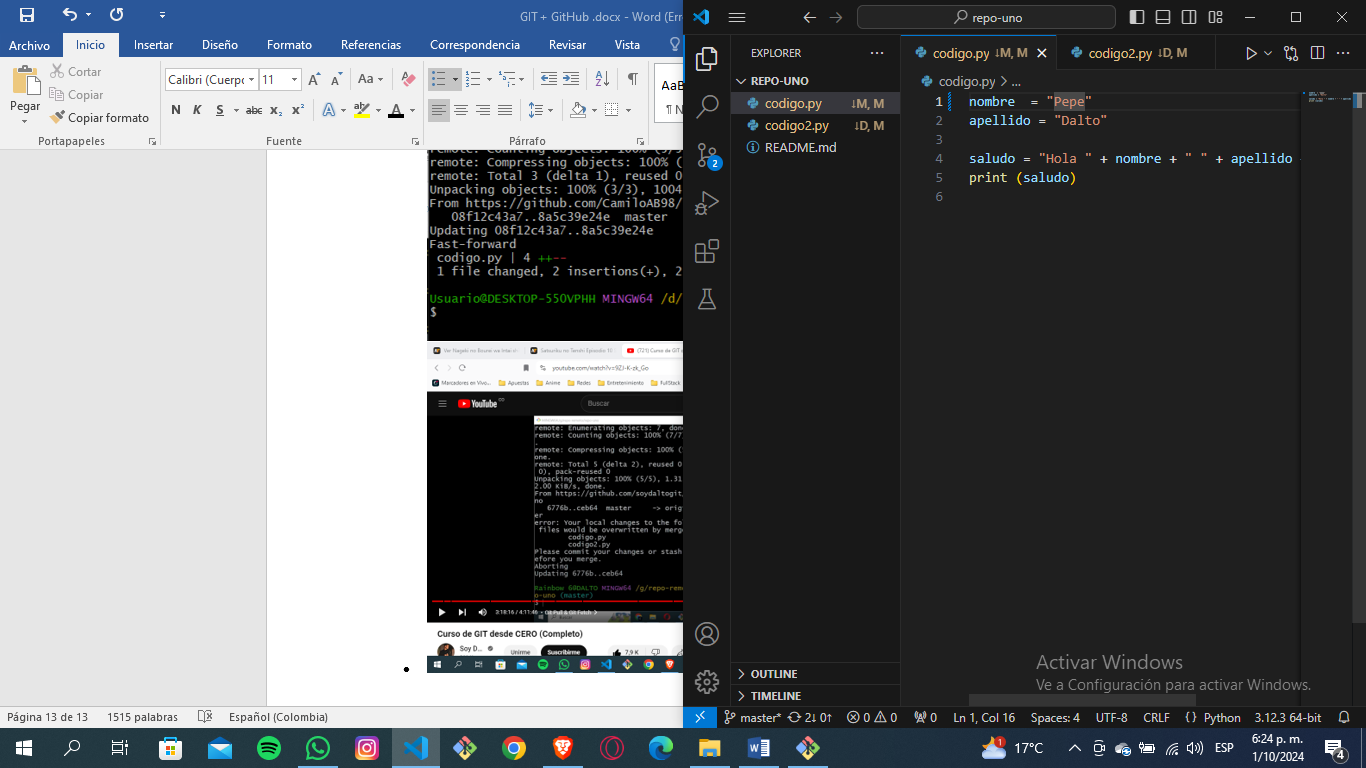
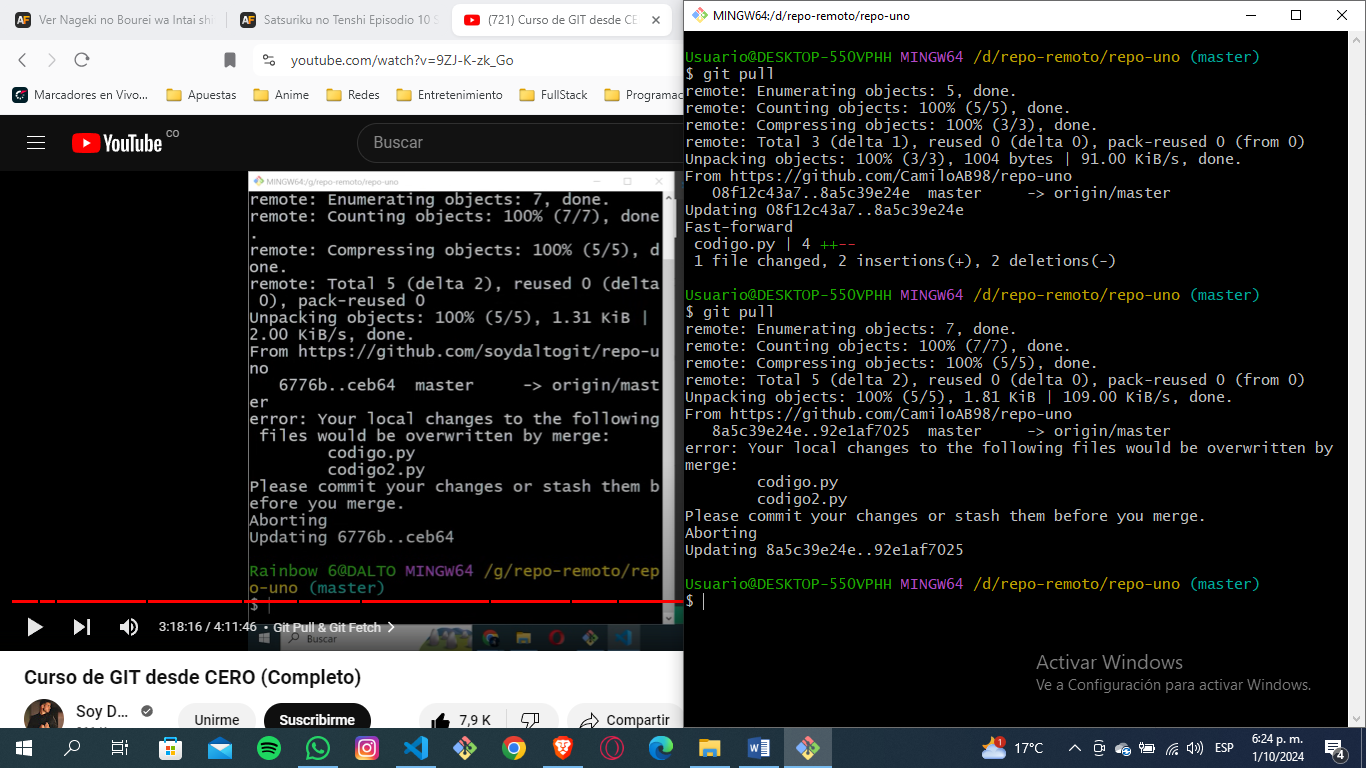
El comando git push se utiliza para comunicar con otro repositorio, calcular lo que tu base de datos local tiene que la remota no tiene, y luego subir (push) la diferencia al otro repositorio. Se requiere acceso de escritura al otro repositorio y por tanto normalmente se autentica de alguna manera.

* Git push + referencia del repositorio remoto (origin) + rama (master) 🡺 Subimos los cambios del repositorio local al repositorio remoto (git hub)

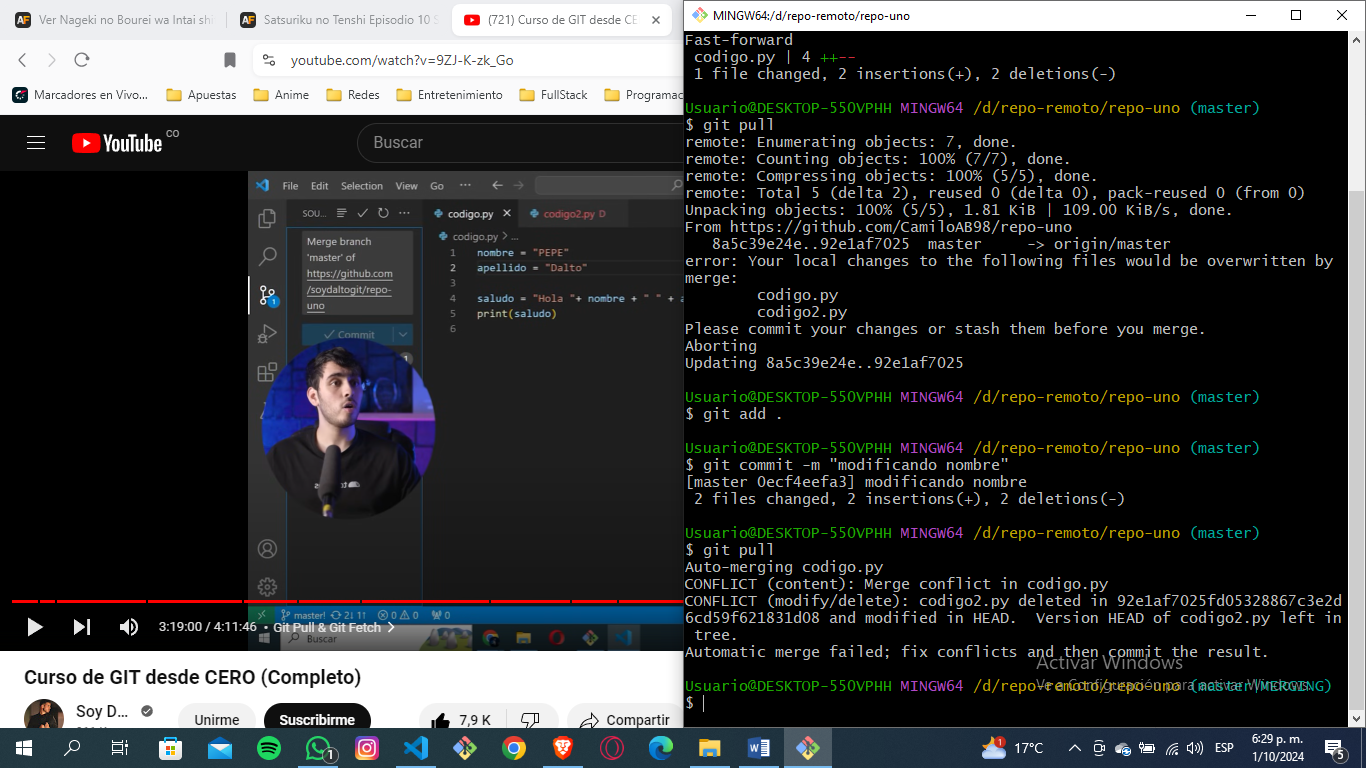


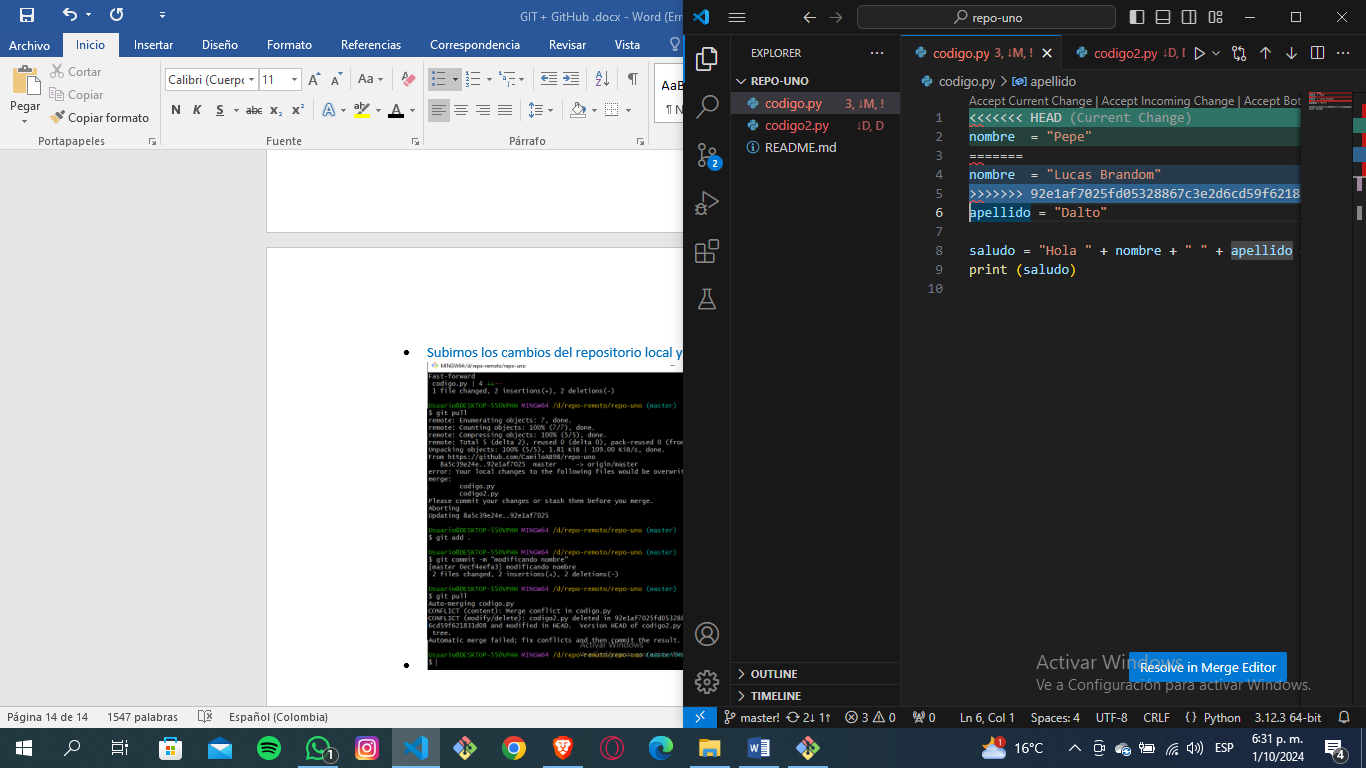
## **GIT PULL**

* El comando git pull es se utiliza para buscar y descargar contenido de un repositorio remoto y actualizar inmediatamente el repositorio local para que coincida con ese contenido. **En pocas palabras, trae los cambios y luego los fusiona con el repositorio local**
* **Error al hacer git pull** 🡺 esto sucede porque tenemos cambios en repositorio remoto y en el local, **error al fusionar (merge).**



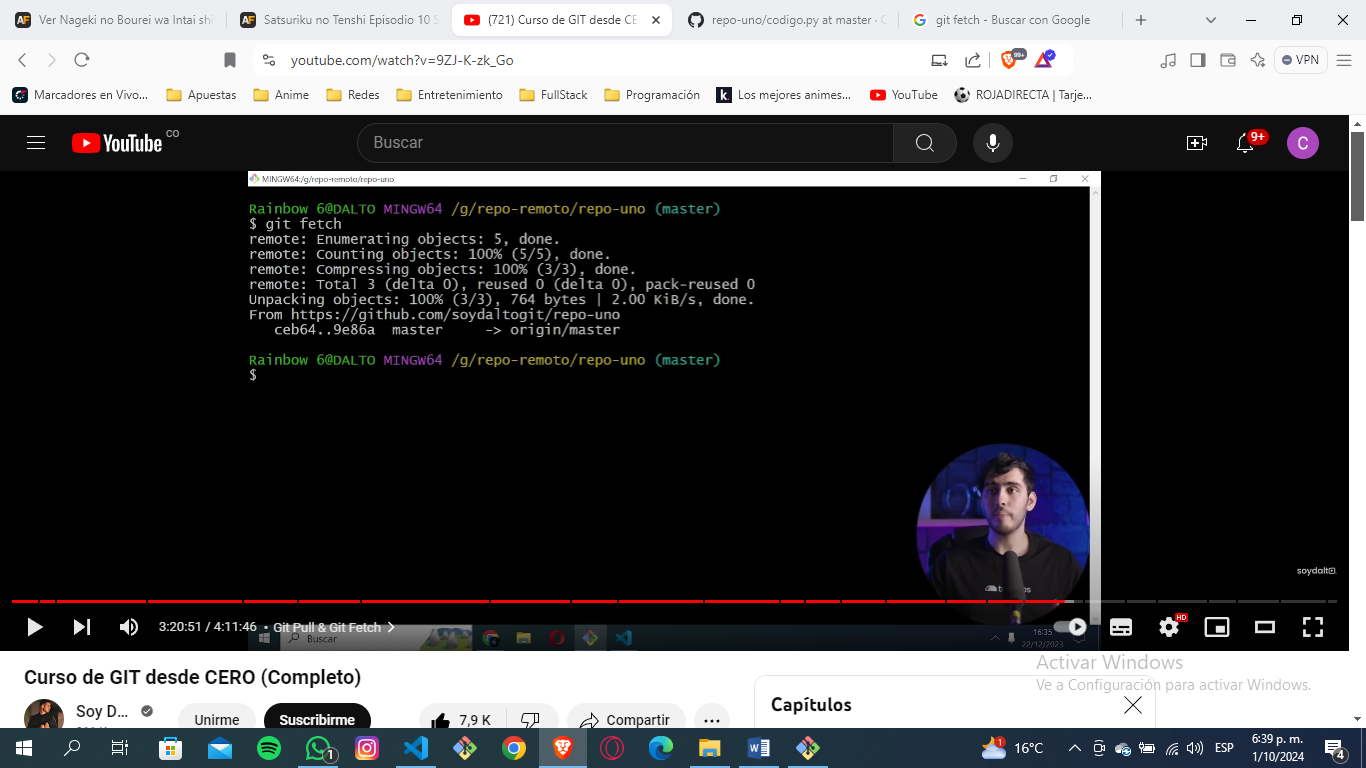
* Subimos los cambios del repositorio local con ‘git add .’, hacemos un commit (git commit), por ultimo hacemos ‘git pull’ para bajar los cambios desde repositorio remoto, esto nos va a generar un conflicto entre lo que bajamos del repositorio remoto y lo que tenemos en el repositorio local. **Resolvemos el merge conflict.**





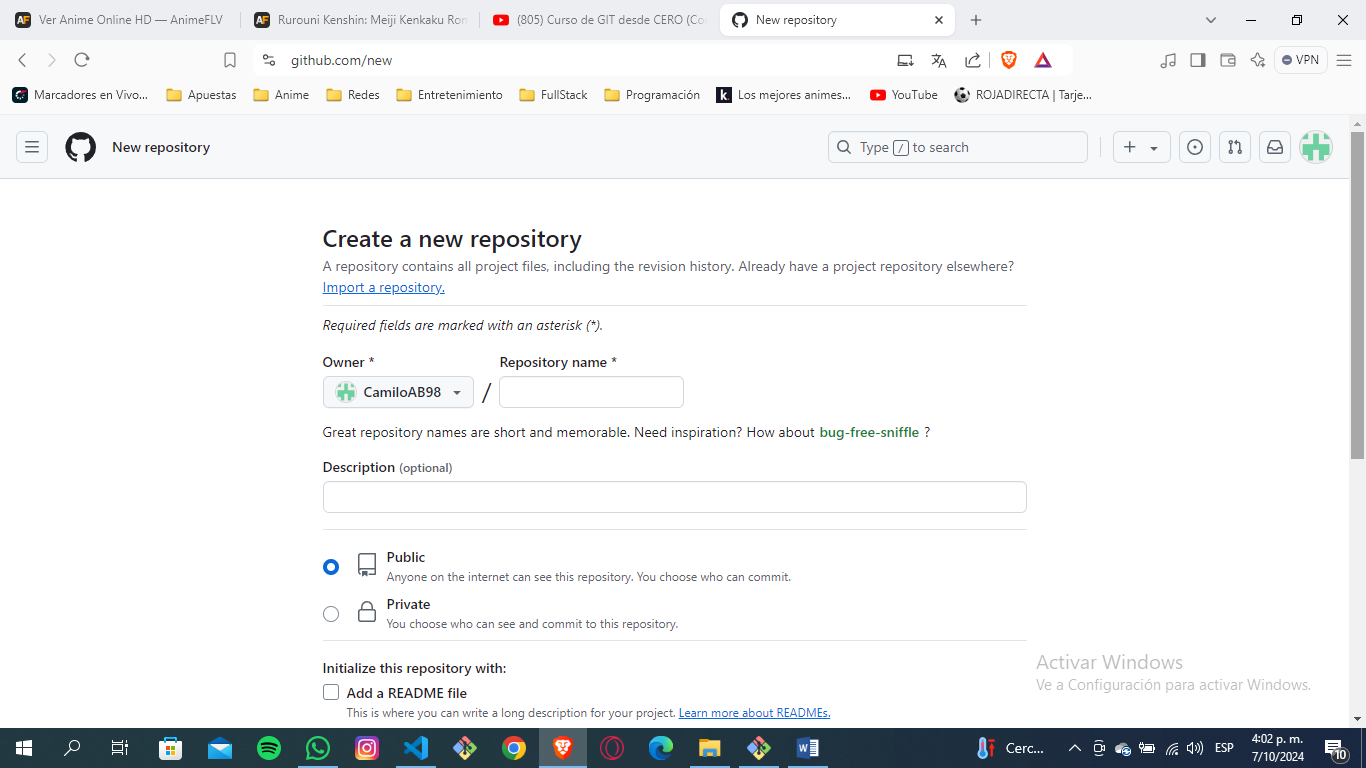
## **GIT FETCH**

El comando git fetch descarga commits, archivos y refs desde un repositorio remoto a su repositorio local. Fetching es lo que haces cuando quieres ver en lo que todos los demás han estado trabajando.

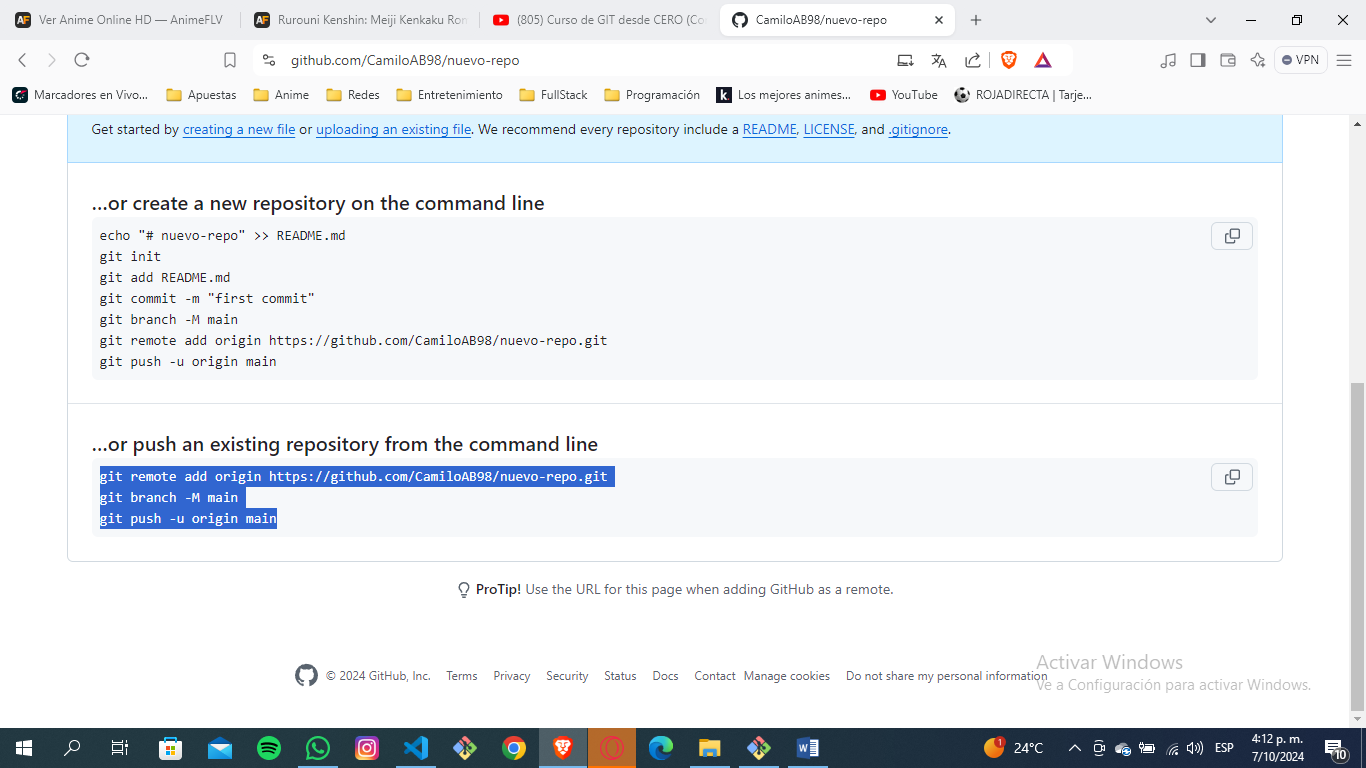


## **Migrar repositorio local a repositorio remoto**

* Creamos un nuevo repositorio



* Copiamos git remote, git branch, git push, y los pegamos en git uno por uno



* Por ultimo hacemos git push

## **Forks**

Una bifurcación es un nuevo repositorio que comparte la configuración de visibilidad y código con el repositorio “ascendente” original.   
  
Un Fork en Git consiste en clonar cualquier repositorio remoto de otro usuario en GitHub a mi repositorio remoto en GitHub, sin que se aloje en la computadora

