**Link del curso de Git.**

[https://www.google.com/search?q=curso+git&sca\_esv=ce2bee5ef594bfe7&sca\_upv=1&sxsrf=ADLYWIK9-M5iHu4ENyiA-FRdQsHfnPFZWg%3A1722870318744&source=hp&ei=LuqwZvGiK8Gxur8P75\_iyQc&iflsig=AL9hbdgAAAAAZrD4Ps0Zo6EeglwgbiJr6QFuxsGiDWC2&ved=0ahUKEwjx8dyKkN6HAxXBmO4BHe-POHkQ4dUDCBc&uact=5&oq=curso+git&gs\_lp=Egdnd3Mtd2l6GgIYAiIJY3Vyc28gZ2l0MgUQABiABDIFEAAYgAQyBRAAGIAEMgUQABiABDIFEAAYgAQyBRAAGIAEMgUQABiABDIFEAAYgAQyBRAAGIAEMgUQABiABEi6T1AAWKIXcAB4AJABAJgB\_QKgAccLqgEHMi41LjEuMbgBA8gBAPgBAZgCCaACrAzCAhAQLhiABBjRAxjHARgnGIoFwgITEC4YgAQYxwEYJxiKBRiOBRivAcICChAjGIAEGCcYigXCAgQQIxgnwgILEC4YgAQYsQMYgwHCAhQQLhiABBixAxiDARioAxiLAxibA8ICCxAAGIAEGLEDGIsDwgIOEAAYgAQYsQMYgwEYiwPCAhcQLhiABBixAxjRAxjSAxjHARioAxiLA8ICCxAAGIAEGLEDGIoFwgIIEAAYgAQYsQPCAgsQABiABBixAxiDAcICDhAuGIAEGLEDGNEDGMcBwgIIEC4YgAQYsQPCAgsQABiABBixAxjJA8ICCxAAGIAEGJIDGIoFmAMAkgcHMS41LjIuMaAHmUY&sclient=gws-wiz#fpstate=ive&vld=cid:38c3657e,vid:9ZJ-K-zk\_Go,st:0](https://www.google.com/search?q=curso+git&sca_esv=ce2bee5ef594bfe7&sca_upv=1&sxsrf=ADLYWIK9-M5iHu4ENyiA-FRdQsHfnPFZWg%3A1722870318744&source=hp&ei=LuqwZvGiK8Gxur8P75_iyQc&iflsig=AL9hbdgAAAAAZrD4Ps0Zo6EeglwgbiJr6QFuxsGiDWC2&ved=0ahUKEwjx8dyKkN6HAxXBmO4BHe-POHkQ4dUDCBc&uact=5&oq=curso+git&gs_lp=Egdnd3Mtd2l6GgIYAiIJY3Vyc28gZ2l0MgUQABiABDIFEAAYgAQyBRAAGIAEMgUQABiABDIFEAAYgAQyBRAAGIAEMgUQABiABDIFEAAYgAQyBRAAGIAEMgUQABiABEi6T1AAWKIXcAB4AJABAJgB_QKgAccLqgEHMi41LjEuMbgBA8gBAPgBAZgCCaACrAzCAhAQLhiABBjRAxjHARgnGIoFwgITEC4YgAQYxwEYJxiKBRiOBRivAcICChAjGIAEGCcYigXCAgQQIxgnwgILEC4YgAQYsQMYgwHCAhQQLhiABBixAxiDARioAxiLAxibA8ICCxAAGIAEGLEDGIsDwgIOEAAYgAQYsQMYgwEYiwPCAhcQLhiABBixAxjRAxjSAxjHARioAxiLA8ICCxAAGIAEGLEDGIoFwgIIEAAYgAQYsQPCAgsQABiABBixAxiDAcICDhAuGIAEGLEDGNEDGMcBwgIIEC4YgAQYsQPCAgsQABiABBixAxjJA8ICCxAAGIAEGJIDGIoFmAMAkgcHMS41LjIuMaAHmUY&sclient=gws-wiz" \l "fpstate=ive&vld=cid:38c3657e,vid:9ZJ-K-zk_Go,st:0)

**Link de descarga de Git**

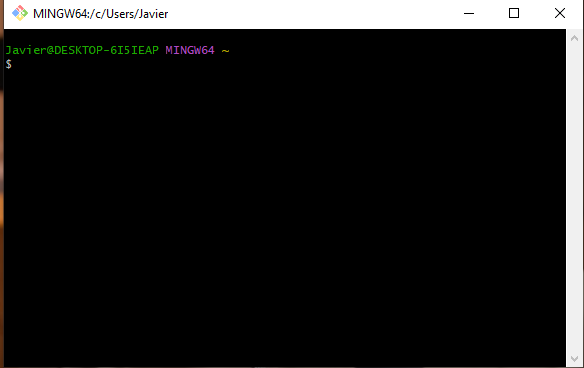
<https://git-scm.com/download/win>

**Link de git-config**

[**https://www.atlassian.com/git/tutorials/setting-up-a-repository/****git-config**](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config)

**Que es Git**

Git es un sistema de control de versiones distribuido, lo que significa que un clon local del proyecto es un repositorio de control de versiones completo. Estos repositorios locales plenamente funcionales permiten trabajar sin conexión o de forma remota con facilidad. Los desarrolladores confirman su trabajo localmente y, a continuación, sincronizan la copia del repositorio con la del servidor.

**Configuración inicial de Git**

Lo primero que necesitamos es definir nuestro alias y nuestro correo esto nos sirve para cuando hagamos cambios en archivos, hacer commits, crear una nueva rama etc. y los subamos a GitHub se sepa quien hizo que cosa para eso nos sirve la firma.

El alias y el correo deben ser mismos en mi repositorio local en la nube GitHub.

**Niveles y archivos de configuración de git**

Antes de analizar más a fondo git config el uso, dediquemos un momento a analizar los niveles de configuración. El comando git config puede aceptar argumentos para especificar en qué nivel de configuración operar. Están disponibles los siguientes niveles de configuración:

* --local

De manera predeterminada, git config se escribirá en un nivel local si no se pasa ninguna opción de configuración. La configuración de nivel local se aplica al contexto git config en el que se invoca el repositorio. Los valores de configuración local se almacenan en un archivo que se puede encontrar en el directorio .git del repositorio:.git/config.

Otras palabras en el directorio donde se invoca git config los valores de configuración local se almacenan en un archivo que se puede encontrar en el directorio .git del repositorio:.git/config.

Tener en cuenta que en este contexto puedo tener diferentes firmas en cada repositorio ya que cada directorio tiene su archivo .git/config

* --global

La configuración de nivel global es específica del usuario (Se Aplica a todos los repositorios de un usuario.), lo que significa que se aplica a un usuario del sistema operativo. Los valores de configuración global se almacenan en un archivo ubicado en el directorio de inicio del usuario ~ /.gitconfig en sistemas Unix y C:\Users\\.gitconfigWindows

* --system

La configuración a nivel de sistema se aplica a toda la máquina. Esto cubre a todos los usuarios de un sistema operativo y todos los repositorios. El archivo de configuración a nivel de sistema se encuentra en un gitconfig archivo fuera de la ruta raíz del sistema $(prefix)/etc/gitconfigen los sistemas Unix. En Windows, este archivo se puede encontrar en C:\Documents and Settings\All Users\Application Data\Git\config en Windows XP y en C:\ProgramData\Git\config en Windows Vista y versiones posteriores.

Por lo tanto, el orden de prioridad de los niveles de configuración es: local, global y sistema. Esto significa que, cuando se busca un valor de configuración, Git comenzará en el nivel local y avanzará hasta el nivel del sistema.

**Escribir un valor**

Ampliando lo que ya sabemos sobre git config, veamos un ejemplo en el que escribimos un valor:

git config --global user.email "your\_email@example.com"

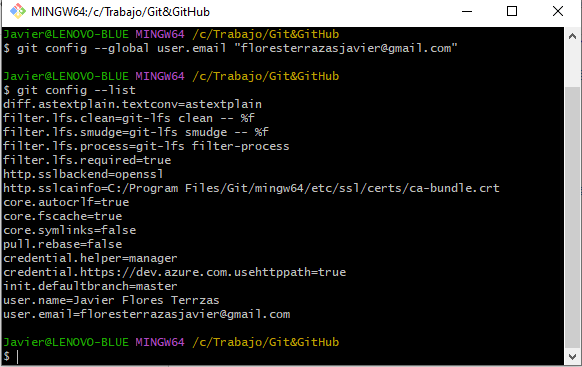
Este ejemplo escribe el valor en el nombre de configuración . Utiliza el indicador para que este valor se establezca para el usuario del sistema operativo [actual.your\_email@example.comuser.email--global](mailto:actual.your_email@example.comuser.email--global)

**Aplico mi configuración a nivel global ( user.name y user.email)**

git config -- global user.name “Javier Flores”

git config --global user.email [floresterrazasjavier@gmail.com](mailto:floresterrazasjavier@gmail.com)

git config --list



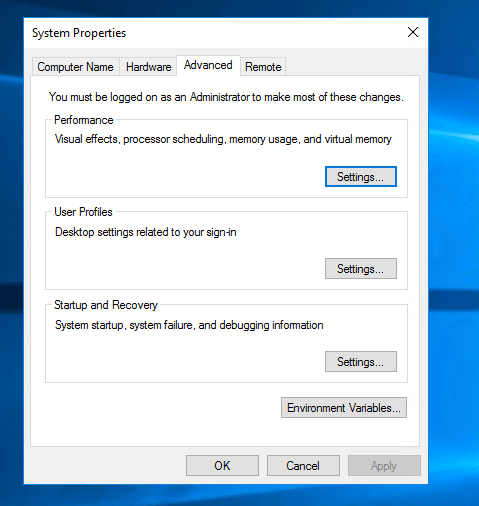
**Configuro el editor de código Visual Studio Code**

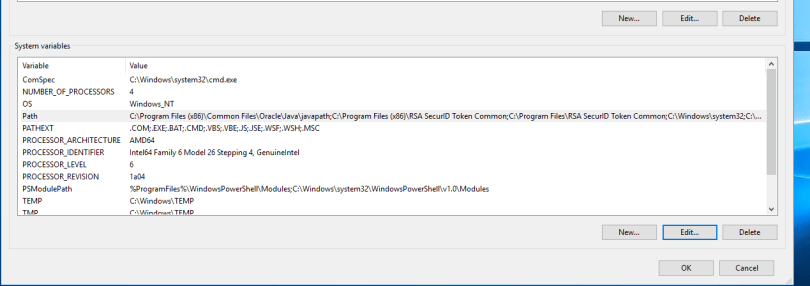
A veces, al implementar Git, es posible que olvides seleccionar Visual Studio Code como editor predeterminado. Pensé que sería bueno tomarme un momento para mostrarles a todos lo rápido y fácil que puedes cambiar el editor global predeterminado en la consola Bash de Git.

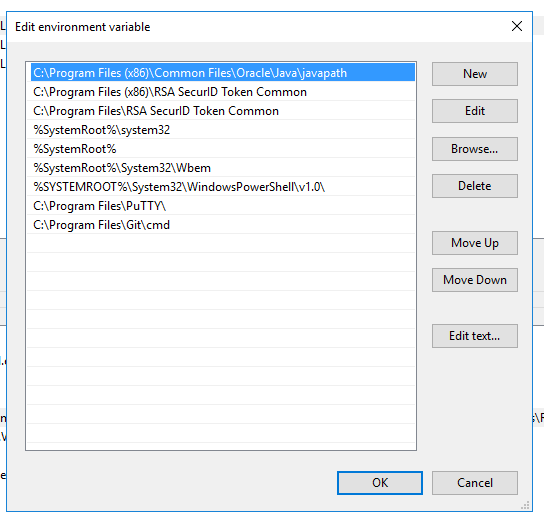
Primero debemos asegurarnos de que VS Code esté en la ruta.

<https://digitalthoughtdisruption.com/2020/09/08/make-vs-code-default-editor-in-git-post-install/>

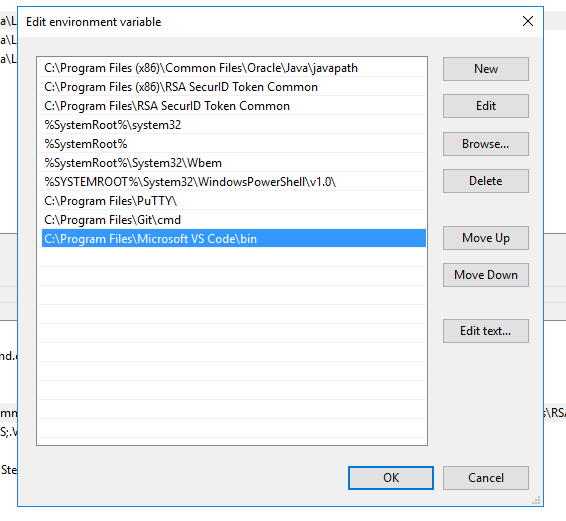
**Configuraciòn avansada del sistema**







Como puede ver, falta la ruta de VS Code, así que haga clic en Nuevo



En Mí maquina la ruta fue: C:\Users\Javier\AppData\Local\Programs\Microsoft VS Code

Una forma sencilla de confirmar que funcionó es simplemente escribir $ code en tu Git Bash y debería abrir VS Code.

En tu Git Bash escribe el siguiente comando:

git config --global core.editor "code -wait -new-window"

git config --global core.editor "code -w -n "

tecleando en git bash code –help para ver las opciones del editor.

VS Code ahora debería ser tu editor global

–wait – se asegura de que Git bash espere a que se cierre el editor de código

–new-window garantizará que se abra una nueva ventana cada vez

**16:33**

**Configurar los colores**

git config --global color.ui true

Revisar este comando \*\*\* Terminar \*\*\*

git config –global autocrlf

**Creando nuestro primer Repositorio**

Un repositorio es un espacio que se utiliza para almacenar archivos, un repositorio tiene 3 áreas un lugar de trabajo (Working Directory. Directorio donde tenemos nuestros archivos).

El área de preparación (Staging Área) es un archivo, generalmente contenido en el directorio de Git, que almacena información sobre lo que se incluirá en la próxima confirmación. Su nombre técnico en el lenguaje de Git es “índice”, pero la frase “área de preparación” funciona igual de bien.

El directorio Git (.git directory Repository) es donde Git almacena los metadatos y la base de datos de objetos de tu proyecto. Esta es la parte más importante de Git y es lo que se copia cuando clonas un repositorio desde otra computadora.

**28:29**

Hago un parénesis a otra liga para comprender mejor lo anterior:

<https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F#:~:text=The%20staging%20area%20is%20a,area%E2%80%9D%20works%20just%20as%20well>.

Tres diferencias en otros cvs y git:

1. Git Instantáneas, Otros cvs no diferencias
2. Casi todas las operaciones son locales
3. Git tiene integridad
4. Git generalmente solo agrega datos

**Los 3 Estados de Git**

Git tiene tres estados principales en los que pueden residir tus archivos: modified , staging y commited :

1. Modificado significa que has cambiado el archivo pero aún no lo has confirmado en tu base de datos.
2. Preparado significa que has marcado un archivo modificado en su versión actual para incluirlo en tu próxima instantánea de confirmación.
3. Comprometido significa que los datos están almacenados de forma segura en su base de datos local.

Esto nos lleva a las tres secciones principales de un proyecto Git: el árbol de trabajo, el área de preparación y el directorio Git.



Árbol de trabajo, área de preparación y directorio Git.

1. El árbol de trabajo es una única versión del proyecto. Estos archivos se extraen de la base de datos comprimida en el directorio Git y se colocan en el disco para que los uses o los modifiques.
2. El área de preparación es un archivo, generalmente contenido en el directorio de Git, que almacena información sobre lo que se incluirá en la próxima confirmación. Su nombre técnico en el lenguaje de Git es “índice”, pero la frase “área de preparación” funciona igual de bien.
3. El directorio Git es donde Git almacena los metadatos y la base de datos de objetos de tu proyecto. Esta es la parte más importante de Git y es lo que se copia cuando clonas un repositorio desde otra computadora.

El flujo de trabajo básico de Git es más o menos así:

1. Modifica archivos en tu árbol de trabajo.
2. Prepara de forma selectiva solo aquellos cambios que desea que formen parte de su próxima confirmación, lo que agrega solo esos cambios al área de preparación.
3. Realiza una confirmación, que toma los archivos tal como están en el área de preparación y almacena esa instantánea de forma permanente en su directorio Git.

Si una versión particular de un archivo está en el directorio de Git, se considera que está comprometida . Si se modificó y se agregó al área de preparación, se considera que está preparada . Y si se modificó desde que se extrajo pero no se preparó, se considera que está modificada.

**Continúo** con el video **28:29**

C:\Trabajo\Git&GitHub

git init -> **Este comando inicializa un repositorio.**

master es la rama principal.

De ahora en adelante trabajo con mi archivo Resumen.doc para los ejercicios.

git add -> Sube in archivo a el área de reparación ( Staging)

git status

El comando git status de Git muestra el estado de los archivos en el directorio de trabajo y en el área de preparación. Con este comando se puede ver:

1. Qué cambios se han preparado
2. Qué cambios no se han preparado
3. Qué archivos no están siendo rastreados por Git
4. Qué archivos están modificados y sin seguimiento
5. Qué archivos están con seguimiento pero no confirmados aún

git commit

El comando git commit de Git sirve para crear una instantánea de los cambios que se han realizado en un proyecto. Estas instantáneas se pueden considerar como versiones "seguras" del proyecto, ya que Git no las modifica a menos que se solicite explícitamente.

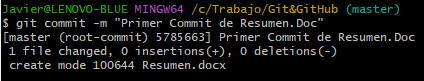
En otras palabras compromete los archivos que están en al área de preparaciòm.

Algunas características de los commits son:

1. Permiten recordar los cambios realizados en una fecha determinada y revertir el proyecto a esa versión.
2. Si varios commits modifican diferentes partes del proyecto, no se sobrescriben entre sí.
3. Los commits semánticos permiten leer el historial de commits para publicar nuevas versiones de un paquete o aplicación.

git commit -m

Compromete todos los archivos los archivos que están en el área de preparación.



Modifique Resumen.Doc

1. git add Resumen.Doc lo subo al área de preparación
2. git commit hago el commit pero sin –m esperando que se habrá le editor.

Todo salió bien en la prueba.

**SESSIÒN RESTORE, CHECKOUT Y MAS DE COMMITS**

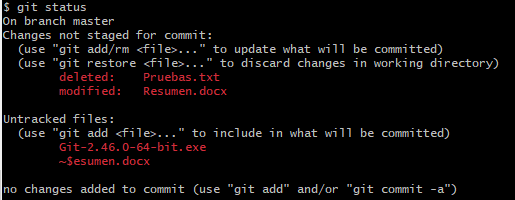
Hacer commit sin pasar por el área de preparación?

**git commit –a**

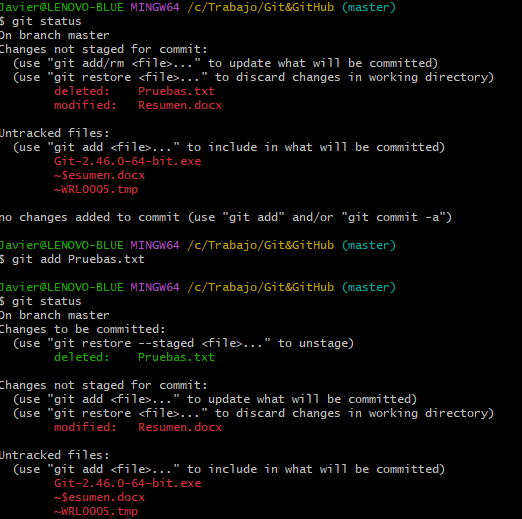
Indique al comando que prepare automáticamente los archivos que han sido modificados y eliminados, pero los archivos nuevos sobre los que no le ha informado a Git no se verán afectados.

Ejercicio:

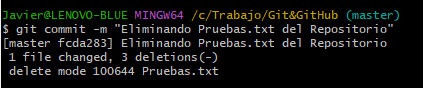
Un borre el archivo Pruebas.txt del espacio de trabajo y ya que había comprometido el Pruebas.txt.



Se Observa que git status me informa que el archivo Pruebas.txt fue borrado.



Posteriormente hago un git add al archivo borrado si me permite pasarlo de nuevo a el área de preparación aun cuando borre, ver la secuencia.



Ahora cuando hago git commit -m "Eliminando Pruebas.txt del Repositorio" -a

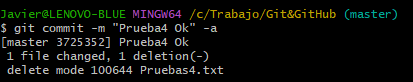
En realidad le estamos diciendo a git haga un commit del archivo eliminado y estaba en el área de preparación pero como borrado entonces no elimina de su base de datos.

Ejercicio:

**Removiendo archivos del repositorio**

Haciendo pruebas4 de ejercicio:

1. Creo archivo Pruebas4.txt
2. git add Pruebas4.txt
3. git commit -m "Pruebas 4"
4. borro archivo del area de trabajo
5. git add Pruebas4.txt lo agrega de nuevo a el area de preparacion
6. git commit -m "Prueba4 Ok" -a



Delete mode 100644 Indica que el archivo a ha sido removido el repositorio.

**Restaurando archivos desde el área de reparación**

Haciendo pruebas5 de ejercicio:

1. Creo archivo Prueba5.txt
2. git add Prueba5.txt
3. git commit -m "Prueba 5"
4. borro archivo del área de trabajo
5. git add Prueba5.txt lo agrega de nuevo a el área de preparación
6. git restore --staged Prueba5.txt

**¿Qué es git checkout?**

El comando git checkout cambia entre ramas o restaura los archivos del árbol de trabajo (working tree).

**Los conceptos de "main", "origin" y "HEAD"**

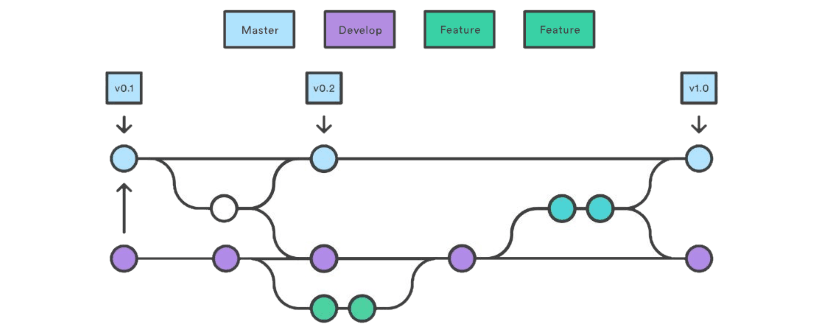
**Main**

Como sabes, los sistemas de control de código fuente como Git trabajan con el concepto de ramas (o branches en inglés). Dicho de manera rápida y básica, una rama no es más que un nombre que se da a un commit, a partir del cual se empieza a trabajar de manera independiente y con el que se van a enlazar nuevos commits de esa misma rama. Las ramas pueden mezclarse de modo que todo el trabajo hecho en una de ellas pase a formar parte de otra.

*Por si no lo tienes claro, un commit es un conjunto de cambios en los archivos que hemos dado por buenos y que queremos almacenar como una instantánea de cara al futuro. Los commits se relacionan unos con otros en una o varias secuencias para poder ir viendo la historia de un determinado archivo a lo largo del tiempo.*

Existe una rama predeterminada que se crea automáticamente cuando se crea un repositorio que se llama rama main.

Por regla general a main se la considera la rama principal y la raíz de la mayoría de las demás ramas. Lo más habitual es que en main se encuentre el "código definitivo", que luego va a producción, y es la rama en la que se mezclan todas las demás tarde o temprano para dar por finalizada una tarea e incorporarla al producto final:



Esta manera de trabajar con ramas nos permite llevar en paralelo varios desarrollos relacionados sin importar que cada uno de ellos termine en momentos muy diferentes, ya que no interfieren, pudiendo mezclarlos todos al final.

Lo más habitual es que para poder mezclar otra rama cualquiera con main haya que pedir permiso y que alguien lo revise todo antes de permitirlo. Es lo que se denomina un **"pull request**". O simplemente que la rama main se encuentre protegida de modo que solo si se pasan todos los test y aseguramos que el producto funciona, sea posible mezclarse con ella. De este modo impedimos que cualquiera pueda llevar al producto final código que no cumple unos mínimos de calidad.

**Origin**

Como seguramente sabrás Git es un sistema de control de código distribuido. Esto quiere decir que, aunque todos los desarrolladores tienen una copia exacta del mismo repositorio en su disco duro, existen uno o más repositorios remotos contra los que trabajamos, y que son los que almacenan el estado final del producto. Estos repositorios remotos se suelen llamar simplemente "remotos", y no todo el mundo tiene permiso para enviar commits hacia ellos (lo que se llama hacer un push).

Bien, pues origin es simplemente el nombre predeterminado que recibe el repositorio remoto principal contra el que trabajamos. Cuando clonamos un repositorio por primera vez desde GitHub o cualquier otro sistema remoto, el nombre que se le da a ese repositorio "maestro" es precisamente origin.

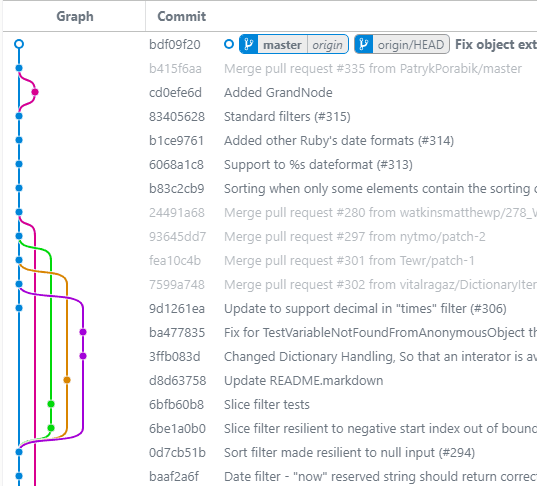
De todos modos ese nombre se puede cambiar, y se pueden crear más remotos contra los que hacer push. Pero en un porcentaje muy alto de los casos, como ese nombre por omisión no se cambia, puedes asumir que vas a enviar la información a origin.

**Head**

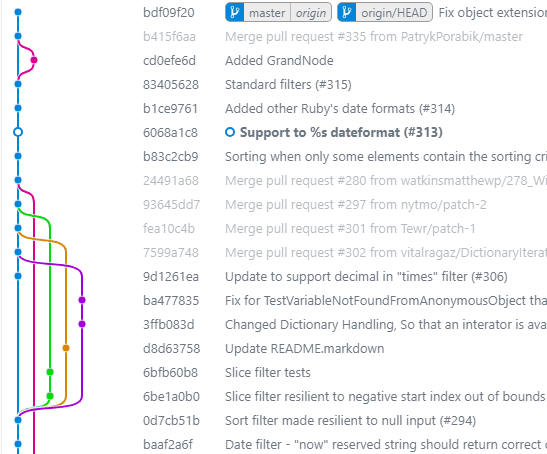
El concepto de HEAD es muy simple: se refiere al commit en el que está tu repositorio posicionado en cada momento. Por regla general HEAD suele coincidir con el último commit de la rama en la que estés, ya que habitualmente estás trabajando en lo último. Pero si te mueves hacia cualquier otro commit anterior entonces el HEAD estará más atrás.

De hecho, si tienes el repositorio actualizado (te has traído los últimos cambios de origin) y estás trabajando en la rama main lo más habitual es que coincidan las tres cosas.

Por ejemplo, este es el log, mostrando las ramas, del repositorio de un proyecto Open Source en el que participo. El HEAD local se muestra como un punto con borde azul. Como se puede observar, en el momento en el que saqué la captura tenía en local la rama master (hoy en día sería main) y estaba ubicado en el último commit (cuyo hash identificador comienza por bdf09f20) y coincide con el último commit en el origen:



Si por el contrario me muevo a otra rama o a un commit anterior, o si el repositorio de origen en la misma rama va por delante de lo que yo tengo en local, ninguno de los tres tiene por qué coincidir. Por ejemplo, me he movido a un commit anterior (el 6068a1c8): y ahora mismo mi HEAD local está 5 commits por detrás (es la que está en negrita con un punto de borde azul), mientras que la rama máster (que replica a la misma en el origen) está 5 commits por delante. El HEAD del repositorio remoto (origen) está en el último commit:



En esta página, se analiza el comando git checkout. Se abordarán ejemplos de uso y casos extremos. En términos de Git, "checkout" (extraer) es el acto de cambiar entre diferentes versiones de una entidad objetivo. El comando git checkout opera sobre tres entidades distintas: **archivos, confirmaciones y ramas**. Además de esta definición, se utiliza habitualmente la expresión "checking out" o "extraer" para referirse al acto de ejecutar el comando git checkout. En el tema sobre cómo deshacer cambios, explicamos cómo puede usarse git checkout para ver confirmaciones antiguas. El eje central de la mayor parte de este documento serán las operaciones de extracción en las ramas.

**Cambio de Rama**

El comando git checkout te permite desplazarte entre las ramas creadas por git branch. Al extraer una rama, se actualizan los archivos en el directorio de trabajo para reflejar la versión almacenada en esa rama y se indica a Git que registre todas las confirmaciones nuevas en dicha rama. Puedes contemplar todo esto como una forma de seleccionar la línea de desarrollo en la que trabajas.

En ocasiones, el comando git checkout puede confundirse con git clone. La diferencia entre ambos comandos estriba en que el segundo recupera código de un repositorio remoto, mientras que el primero cambia entre versiones de código que ya se encuentran en el sistema local.

**Uso: ramas existentes (git branch)**

Si damos por sentado que el repositorio en el que trabajas contiene ramas preexistentes, puedes cambiar de una de estas ramas a otra con git checkout. Para saber qué ramas están disponibles y cuál es el nombre de la rama actual, ejecuta git branch.

$＞ git branch

main

another\_branch

feature\_inprogress\_branch

$＞ git checkout feature\_inprogress\_branch

El comando git checkout se usa habitualmente junto con git branch. El comando git branch permite crear una rama nueva. Si quieres empezar a trabajar en una nueva función, puedes crear una rama nueva a partir de la rama main con git branch new\_branch. Una vez creada, puedes usar git checkout new\_branch para cambiar a esa rama. Además, el comando git checkout acepta el argumento -b, que actúa como un práctico método que creará la nueva rama y cambiará a ella al instante. Puedes trabajar en varias funciones en un solo repositorio alternando de una a otra con git checkout.

git checkout -b ＜new-branch＞

En el ejemplo anterior, se crea y se extrae la rama ＜new-branch＞ simultáneamente. La opción -b es una marca muy útil con la que Git ejecuta git branch antes de hacer lo propio con git checkout ＜new-branch＞.

git checkout -b ＜new-branch＞ ＜existing-branch＞

De manera predeterminada, git checkout -b basará la rama new-branch en el HEAD actual. No obstante, git checkout puede combinarse con un parámetro opcional para ramas adicionales. En el ejemplo anterior, se añade ＜existing-branch＞, que basa new-branch en existing-branch y no en el HEAD actual.

\*\*\* Terminar \*\*\*

<https://www.atlassian.com/es/git/tutorials/using-branches/git-checkout>

\*\*\* Terminar \*\*\*

<https://www.atlassian.com/es/git/tutorials/undoing-changes>

\*\*\* Terminar \*\*\*

<https://www.atlassian.com/es/git/tutorials/resetting-checking-out-and-reverting>

**Recuperar un archivo desde el último commit. Ejercicio:**

1. creo un archivo.txt
2. git add archivo.txt
3. git commit -m archivo.txt “Primera subida de archivo archivo.txt”
4. borro archivo archivo.txt
5. git checkout archivo.txt ( Extrajo mi archivo desde el ultimo commit)
6. agrego nuevo texto mi perro se llama peluda
7. se encuentra en stating
8. git checkout archivo.txt

**Otro ejercicio de Checkout**

Git Checkout

1. creo checkout.txt
2. git add checkout.txt
3. git commit -m checkout.txt
4. agrego esta frase a checkout.txt "Dios esta conmigo"
5. Con hago git add
6. git checkout checkout.txt
7. Me regreso la version anterior
8. agrego frase checkout.txt "Hoy Volvi a Nacer 19 Sep"
9. Con hago git add
10. git checkout checkout.txt
11. Me regreso al version original.
12. agrego frase checkout.txt " Mi Esposa es Luz Divina"
13. Con hago git add
14. git checkout checkout.txt
15. Me regreso al version original.

Entendido

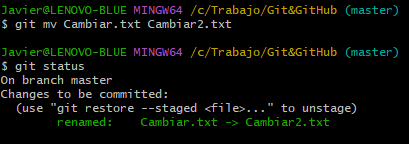
Si ya subi los cambios y ya estan en el area de preparacion, se aplica a todos los

archivos que esten en el area de preparacion.

**git reset –hard**

**Como cambiar el nombre de un archivo**

1. Creo archivo Cambiar.txt
2. git add Cambiar.txt
3. git commit -m "Cambiar nombre de un archivo"
4. git mv Cambiar.txt Cambiar2.txt
5. git add Cambiar2.txt No hace nada porque esta ya renombrado y esta ya en el area de preparacion

****

**Git status sin tanta información.**

git status –s

**Algunos otros comandos de Git son:**

1. git ls-files Este comando es útil para ver la lista completa de archivos que Git tiene en cuenta en el directorio de trabajo y en el área de preparación.
2. git status Muestra la lista de archivos que se han modificado, así como los que están por ser preparados o confirmados.
3. git push Envía confirmaciones locales a la rama maestra del repositorio remoto.
4. git log Muestra los logs de los commits.
5. git diff Muestra los cambios realizados a un archivo.
6. git tag Muestra una lista de todos los tags.
7. git branch Crea un branch.
8. git rebase Se usa para poner al día los branches con el master sin afectar a este último

**git Diff y git show**

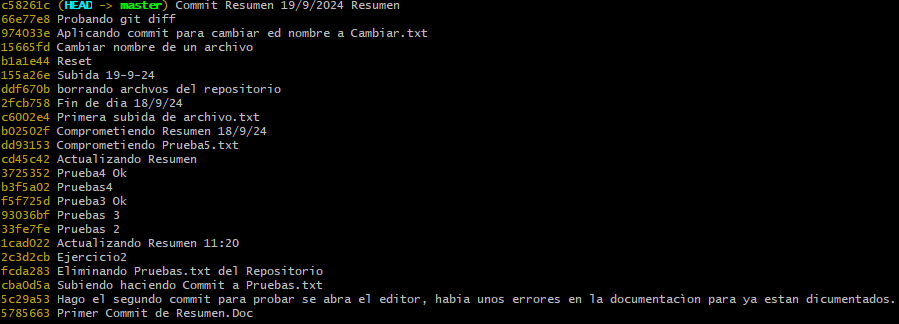
git show < objeto> es una utilidad de línea de comandos que se utiliza para ver detalles ampliados sobre objetos de Git, como blobs, árboles, etiquetas y confirmaciones. Git show tiene un comportamiento específico por tipo de objeto.

<https://www.atlassian.com/git/tutorials/git-show>

**git log --online**

Muestra los commits abreviados de los primeros 7 caracteres su identificador, se puede cambiar la longitud de la abreviatura de los 7 caracteres tecleando

git config --global core.abbrev 10



git diff

Comparando un archivo archivo.txt que esta en el area de preparcion vs ultimo commit archivo.txt

1. creo archivo.txt
2. git add archivo.txt
3. git commit -m "Probando git diff"
4. Si queremos comparar un archivo que es el último commit vs el que está en el area de preparación
5. git diff –staged

**Comparando archivo de diferentes commit**s

git diff <identificador1> <identificador2> 271b050 945254a

en otro formato solo me muestra los archivos que cambiaron

git diff –name-only <identificador1> <identificador2> 271b050 945254a

en otro formato muestra las palabras que cambiaron

git diff --word-diff <identificador1> <identificador2>

**Modificar y deshacer Commits**

Ejercicio

1. Crear un archivo mas\_commits.txt
2. Colocar el texto primer commit
3. git add mas\_commits.txt
4. git commit -m “primer comit”
5. Colocar el texto segundo commit
6. git add mas\_commits.txt
7. git commit -m “segundo comit”
8. Colocar el texto tercer commit
9. git add mas\_commits.txt
10. git commit -m “tercer comit”
11. Colocar el texto cuarto commit
12. git add mas\_commits.txt
13. git commit -m “cuarto comit”
14. Colocar el texto quinto commit
15. git add mas\_commits.txt
16. git commit -m “quinto comit”

Sobre el ultimo Commit voy a modificar el archivo mas\_commits.txt agregándole el texto soy Iron Man.

**git commit –amend**

Este momento se abre el editor de texto con el contenido del archivo modifico y al momento de grabarlo se hace el commit y genera un nuevo Hash.

Nota: Si en el área de preparación existen más archivos se fusionan con mas\_commit.txt con un nuevo hash.

Entedido pero revisar a fondo su documentación para modifica no el ultimo commit si no uno más atrasado. \*\*\*TERMINAR\*\*\*

git rebase -i -head-3 -> indica 3 commits para atrás

git rebase --continue

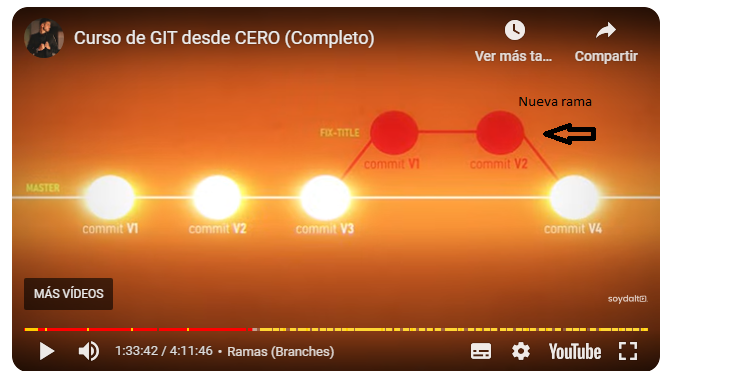
Cuando modificamos un commit que no es el ultimo se borran todos commit que le siguen.

**Como consejo nunca modificar un commit que no es el último es un proceso arduro y difícil.**

**Ramas**

**1:32**

Una rama en esencia un puntero que apunta a conjunto de commits si creamos una rama es una copia del proyecto para trabajar de forma independente sin afectar a la rama Master.



Si todo funciona bien con los cambios en la nueva rama entonces en entonces fusionamos los cambios en nuestra rama Mater es ya llama Merge.

1. Creo un archivo ramas.txt -> en master
2. git add ramas.txt
3. git commit -m “ramas.txt”
4. git branch -> no muestra todas las ramas que tenemos creadas
5. git branch modificaciones-dev
6. git checkout modificaciones-dev
7. otra forma de cambiarnos de rama es usando el comando **git switch** <nombre de la rama>
8. git switch -c modificaciones-dev -> al igual que git checkout modificaciones-dev
9. git switch -c modificaciones-dev -> diferencia crea la nueva rama se cambia a la nueva rama.

**1:40**

**Borrar rama**

**git** **branch** **-d** <nombre de rama>

para borrar la rama no tenemos que estar dentro de la rama.

**Modifcar nombre de la rama**

**git branch -m**  <nombre actual de rama> <nuevo nombre de rama>

para modificar la rama no tenemos que estar dentro de la rama.

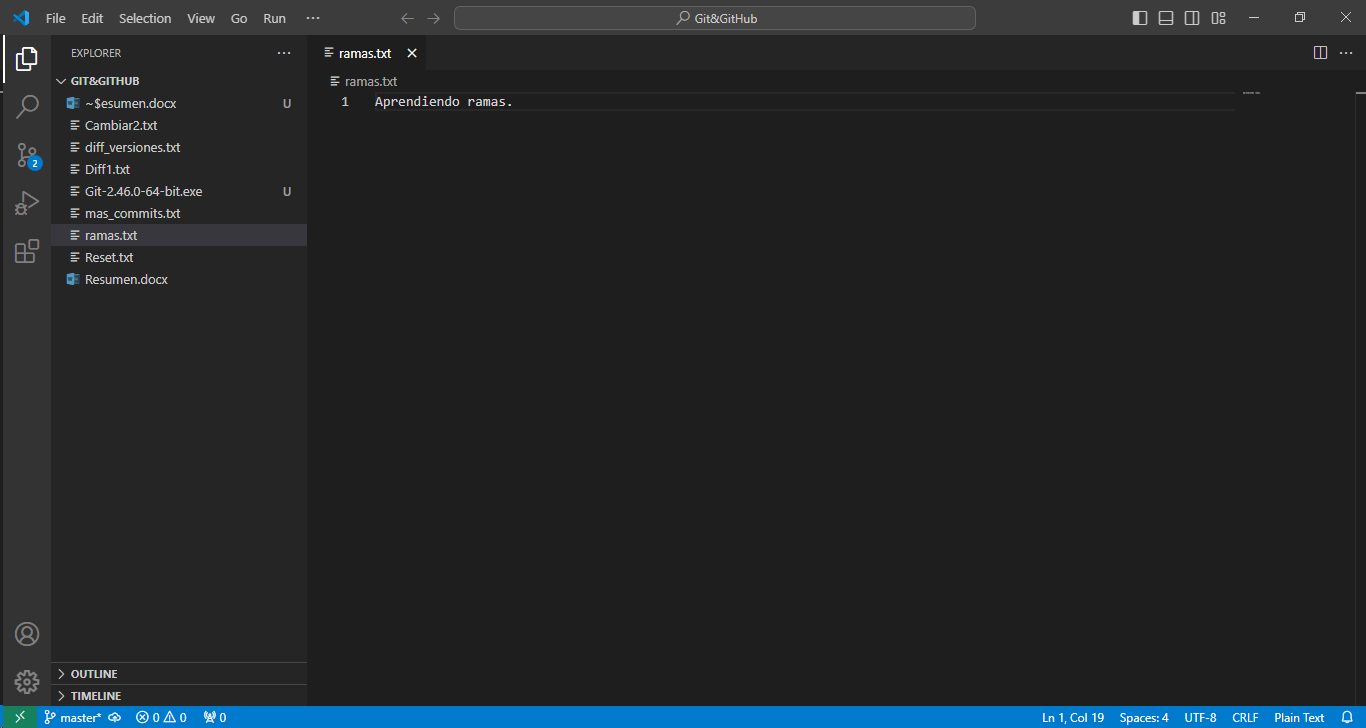
Nota: Situación que no sabía o tal vez por la nueva versión de git. Ve que estoy en rama modificaciones-dev le doy un ls – l veo los mismo archivos que están en la rama master.

Suponia que solo me daba los archivos en la rama en la que estoy, al parcer no es asi, tener mucho cuidado si modifico un archivo cambiándome de rama se modifica el archivo. Voy hacer ejemplos. (Javier)

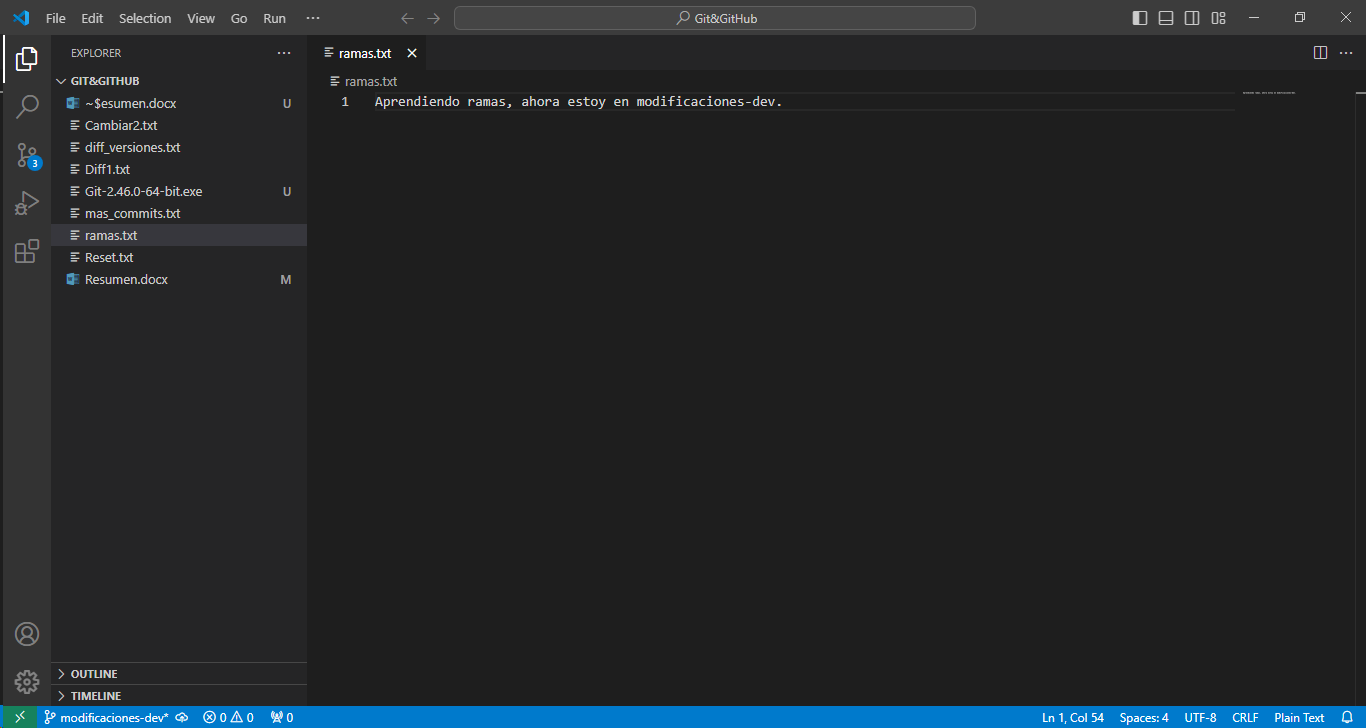
Ejercicio.

1. Si abro un git bash en una ventana1 y estoy en master y abro otra ventana2 y también estoy en master y me cambio en la segunda ventana y me cambio a moficaciones-dev se actualiza también ventana1 a moficaciones-dev.
2. Estoy en master abro ramas.txt tengo este texto "Aprendiendo ramas."
3. No hago git add
4. Abro VCode abro ramas.txt estando en rama master.
5. Me cambio en Bash a modificaciones-dev
6. Me doy cuenta que el archivo ramas.txt en realidad contiene el texto "Aprendiendo ramas, ahora estoy en modificaciones-dev." pero si me cambio de rama a master tengo el texto "Aprendiendo ramas."
7. Mi conclusión es que efectivamente son 2 archivos diferentes. Si mi git bash está en la rama master VCode muestra "Aprendiendo ramas." Si me cambio de rama a moficaciones-dev en vscode muestra "Aprendiendo ramas, ahora estoy en modificaciones-dev."
8. Independientemente que en rama estoy el directorio en Windows muestra todos los archivos del directorio. El archivo que edito depende en que rama estoy Térnelo muy presente.

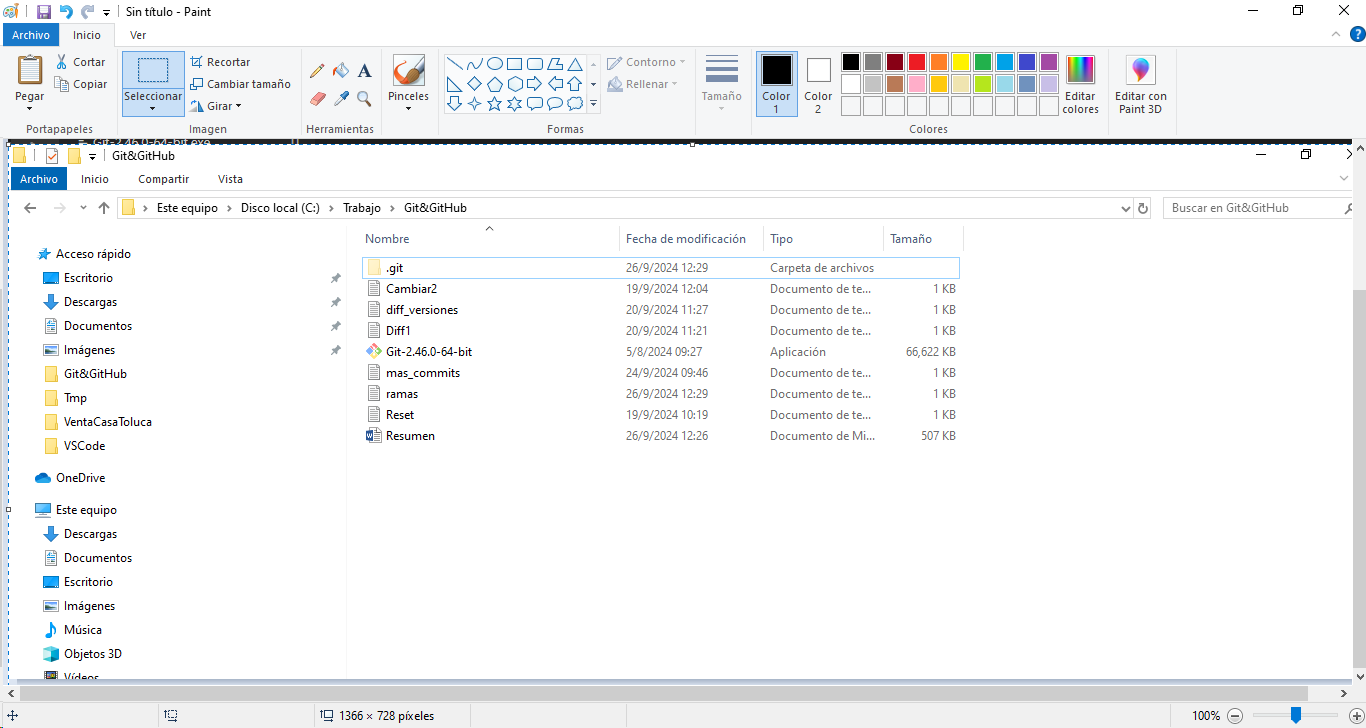
Todo este ejercicio es mi propio equipo.



RAMA MASTER



RAMA MODIFICACIONES-DEV



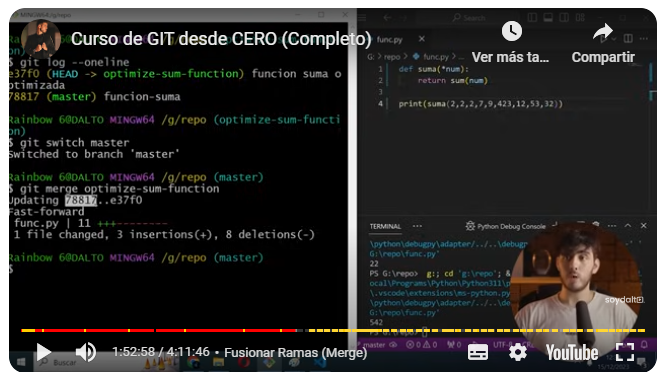
MISMOS ARCHIVOS

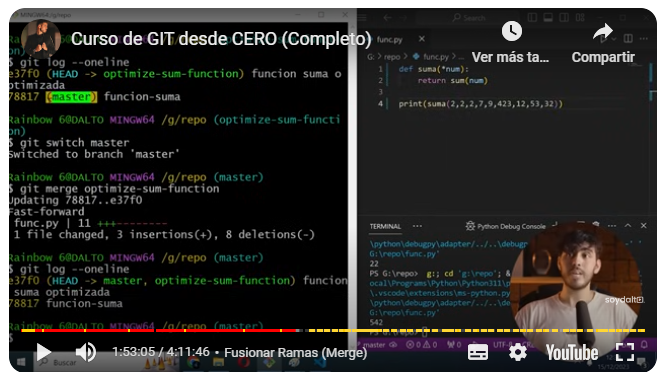
**FUSIONAR RAMAS**

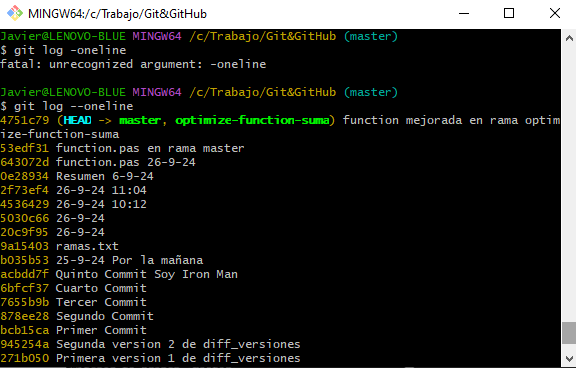
Ejercicio.

/\* Javier flores terrazas \*/

1. Creo archivo function.pas en rama master.
2. git add function.pas
3. git commit -m function.pas "function.pas en rama master"
4. git branch -c optimize-function-suma
5. cambio la implememtacion de function.pas
6. git add function.pas -> estando en rama optimize-function-suma
7. git commit -m "function modificada en rama optimeze-function-suma"
8. estando en rama optimize-function-suma
9. para fusionar rama optimize-function-suma vs rama master
10. debo estar en la rama que recibira la fusion.
11. nos cambiamos a master
12. git switch master
13. git merge optimize-function-suma



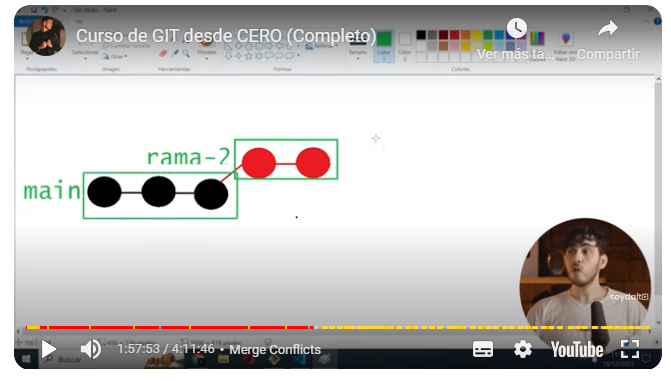




Ahora suponiendo que nos equivocamos.

1. git reset –hard <id de rama que recibió la fusion> bien lo probe y todo bien.

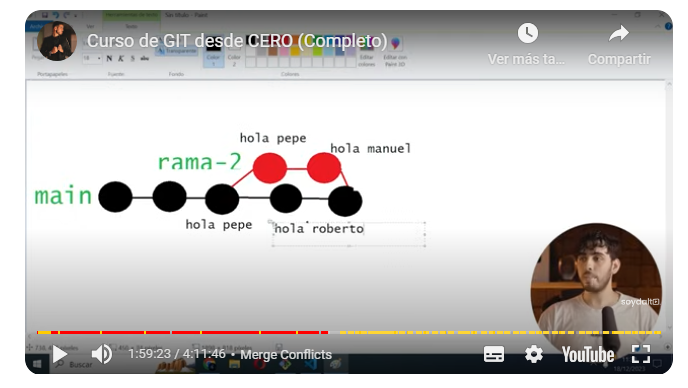
**Merge Conflits**



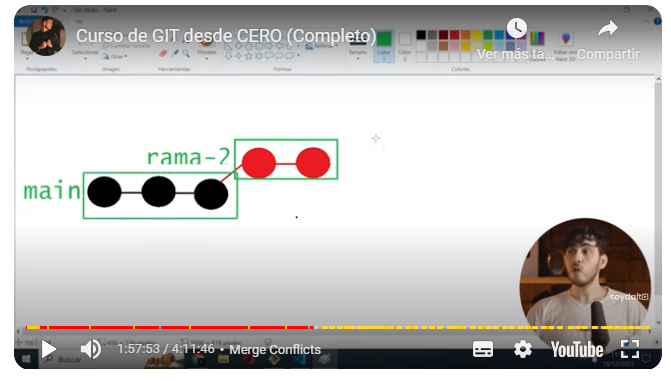
Recordar que rama2 es una copia de rama master y en este ejemplo rama2 tiene 2 commits mas.

Aquí no hay conflictos porque rama 2 es la copia de rama master y al fusionar todo queda claro.

Ahora es este escenario aquí si abría conflictos.



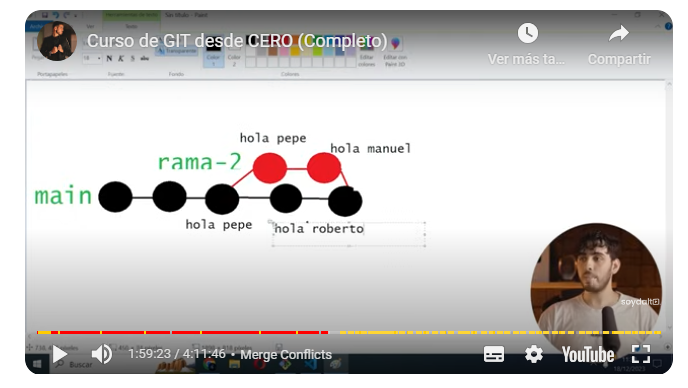
El cuarto commit de rama master no está en el rama2 ademas que los commits de rama2 agrego hola manuel al fusionar y main tiene ahora **hola Roberto** mientras que rama2 que quedo con **hola manuel** con cual nos quedamos con hola manuel o con hola Roberto ¿



Recordar que rama2 es una copia de rama master y en este ejemplo rama2 tiene 2 commits mas.

Aquí no hay conflictos porque rama 2 es la copia de rama master y al fusionar todo queda claro.

Ahora es este escenario aquí si abría conflictos.



El cuarto commit de rama master no está en el rama2 ademas que los commits de rama2 agrego hola manuel al fusionar y main tiene ahora **hola Roberto** mientras que rama2 que quedo con **hola manuel** con cual nos quedamos con hola manuel o con hola Roberto ¿

Ejercicio.

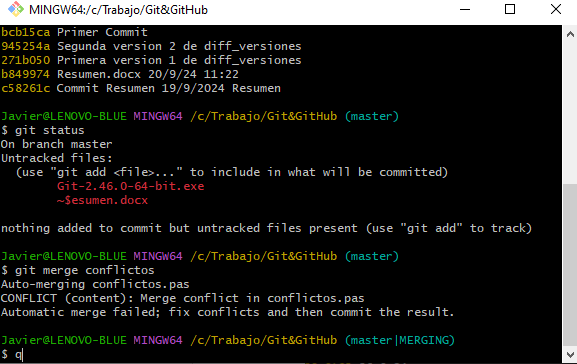
1. Crea archivo en master conflictos.pas
2. git add conflictos.pas
3. git commit –m “confictos.pas en master.”
4. git brach –c conflictos
5. Estando en rama conflictos modifcar el archivo conflictos regrese la funcion “Javier Flores terrazas”
6. git switch master
7. **git log –online –all ->Muestra todos lo commits.**
8. Estando en master agragmos algo en nuestro archivo y creamos otro commit.
9. git add conflictos.pas
10. git commit –m “agregue texto a conflictos .pas estando en master”



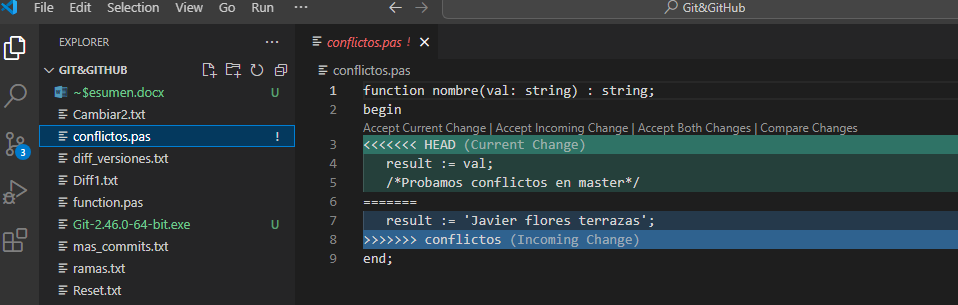
Secuencia:

1. conflictos.pas en master -> master
2. Conflictos.pas con texto Javier Flores Terrazas ->conflictos
3. agregue texto a conflictos .pas estando en master ->master
4. Aquí sucede aqui el conflicto ultimo commit de master es diferente al ultimo commit de conflictos.

Ahora estando en master git merge conflictos ->muestra los conflictos:



En vscode abrimos la unidad conflictos.pas y muestra los conflictos.



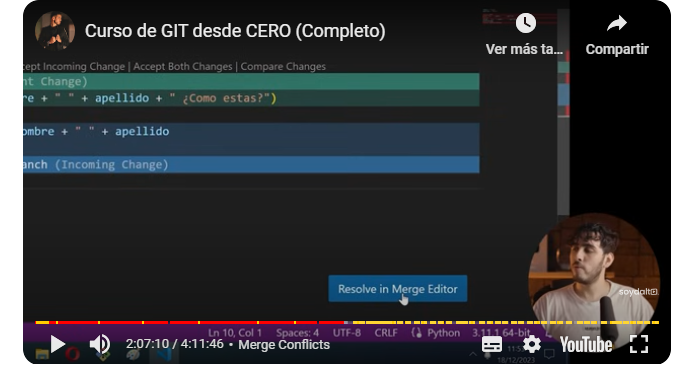
Ahora hay 3 opciones para resolver el conflicto:

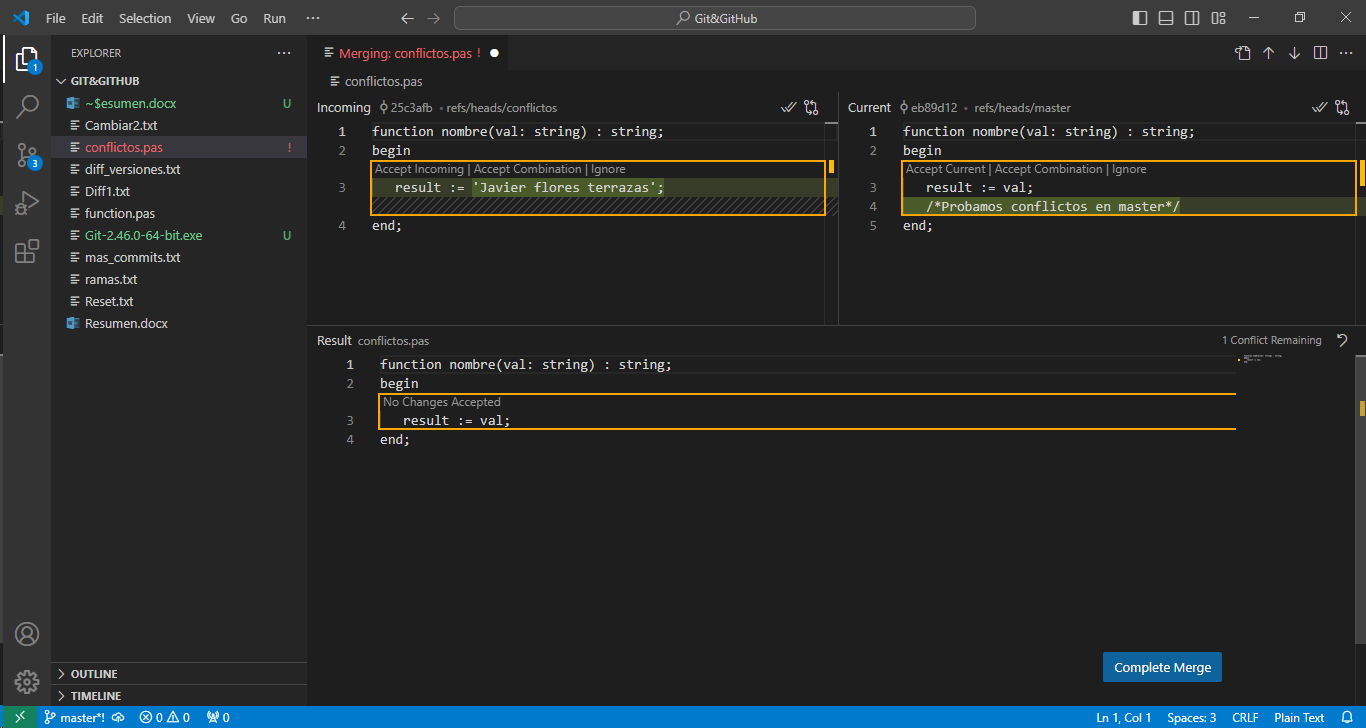
Accept Current Change

Accept Incoming Change

Accept Both Changes

Ahora pulsamos botón Resolver in Merge Editor





En los paneles izquierda y derecha tenemos 3 opciones, en el panel inferior nos muestra como quedaría el resultado y finamente se oprime el botón Complete Merge

**GIT IGNORE**

El archivo .gitignore, es un archivo de texto que le dice a Git qué archivos o carpetas ignorar en un proyecto.

Un archivo local .gitignore generalmente se coloca en el directorio raíz de un proyecto. También puedes crear un archivo global .gitignore, y cualquier entrada en ese archivo se ignorará en todos tus repositorios de Git.

Para crear un archivo .gitignore local, crea un archivo de texto y asígnale el nombre ".gitignore" (recuerda incluir el . al principio). Luego, edita este archivo según sea necesario.

Cada nueva línea debe incluir un archivo o carpeta adicional que quieras que Git lo ignore.

Las entradas de este archivo también pueden seguir un patrón coincidente:

\* se utiliza como una coincidencia comodín.

/ se usa para ignorar las rutas relativas al archivo .gitignore.

# es usado para agregar comentarios

Este es un ejemplo de cómo puede lucir el archivo .gitignore :

# Ignora archivos del sistema Mac

.DS\_store

# Ignora la carpeta node\_modules

node\_modules

# Ignora todos los archivos de texto

\*.txt

# Ignora los archivos relacionados a API keys

.env

# Ignora archivos de configuración SASS

.sass-cache

Para agregar o cambiar tu .gitignore global, ejecuta el siguiente comando en la terminal:

git config --global core.excludesfile ~/.gitignore\_global

Esto creará el archivo ~/.gitignore\_global. Ahora puedes editar ese archivo de la misma manera que un archivo .gitignore local. Todos tus repositorios Git ignorarán los archivos y carpetas listadas en el .gitignore global.

**¿Cómo evitar el rastreo de archivos a los que previamente les has hecho "commit" desde el nuevo Gitignore?**

Para evitar el rastreo de un solo archivo, es decir, detener el rastreo del archivo, pero no borrarlo del sistema, utiliza:

**Para evitar el rastreo de todos los archivos en .gitignore:**

Primero haces "commit" de cualquier cambio de código pendiente, y luego ejecuta:

git rm -r –cached

Esto elimina los archivos modificados del índice (área de montaje), y luego ejecuta:

Git add .

Haces "commit":

git commit -m ".gitignore funciona correctamente"

Para deshacer git rm --cached nombre-del-archivo, usa git add nombre-del-archivo

Link -> <https://www.freecodecamp.org/espanol/news/gitignore-explicado-que-es-y-como-agregar-a-tu-repositorio/>

**Git Alias**

Los alias se usan para crear comandos más cortos que se asignan a comandos más largos.

Es importante tener en cuenta que no existe un comando git alias tal cual.

Los alias se crean mediante el comando git config y los archivos de configuración de Git.

Al igual que sucede con otros valores de configuración, los alias se pueden crear con un

alcance local o global.

Los alias se pueden crear desde la línea de comandos con git config --global alias.[alias] "[comando]"

Dónde [alias] es el alias que queremos usar y [comando] el comando a sustituir.

Ejemplos:

git lg ->Muestralos commits en otro formato

git config --global alias.lg "log --pretty=format:'%h - %an, %ar : %s' --graph"

git alias -> Me muestra todos lo alias

git config --global alias.alias "config --get-regexp ^alias\."

git lgc....................Muestra el log en formato gráfico con colores.

git config --global alias.lgc "log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)%Creset' --abbrev-commit --date=relative"

git lgb....................Muestra el log en formato gráfico con colores y ramas.

git config --global alias.lgb "log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)%Creset' --abbrev-commit --date=relative --branches"

La otra opción para definir los alias es editar el fichero .gitconfig con un editor de texto y añadirlos tú mismo.

En windows el fichero .gitconfig global suele estar en el directorio del perfil de usuario. Puedes teclear %userprofile% en la barra de direcciones del explorador de windows o en Inicio, Ejecutar o Inicio, Buscar para acceder a él.

Crear los alias en el fichero gitconfig

Puedes copiar y pegar directamente estos alias en tu fichero .gitconfig:

[alias]

b = branch

ci = commit

co = checkout

cob = checkout -b

d = diff

l = log -> omite git solo es log

lg = log --pretty=format:'%h - %an, %ar : %s' --graph

lgc = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)%Creset' --abbrev-commit --date=relative

lgb = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)%Creset' --abbrev-commit --date=relative --branches

st = status

En la parte del comando se omite la palabra git

**Git Reflog**

Git guarda todos los cambios (**Comits**)aunque decidas borrarlos, al borrar un cambio lo que estás haciendo sólo es actualizar la punta del branch, para gestionar éstas puntas existe un mecanismo llamado registros de referencia o reflogs…La gestión de estos cambios es mediante los hash’es de referencia (o ref) que son apuntadores a los commits…Los recoges registran cuándo se actualizaron las referencias de Git en el repositorio local (sólo en el local), por lo que si deseas ver cómo has modificado la historia puedes utilizar el comando:

**git reflog**

Muchos comandos de Git aceptan un parámetro para especificar una referencia o “ref”, que es un puntero a una confirmación sobre todo los comandos:

git checkout Puedes moverte sin realizar ningún cambio al commit exacto de la ref por ejemplo:

git checkout eff544f

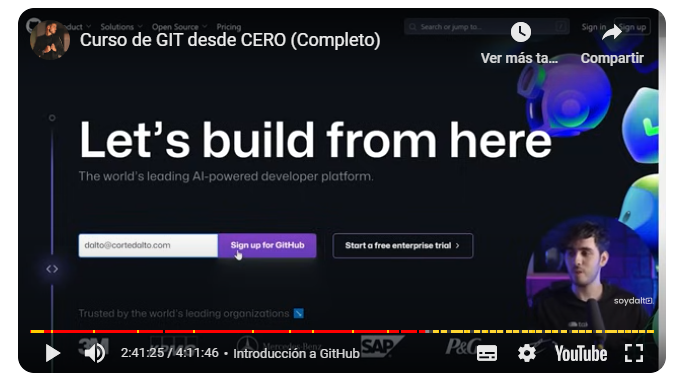
**A menudo se utiliza para con git reset para regresar a un commit que perdemos.**

**git reset es una mala práctica, no deberías usarlo en ningún momento. Debe ser nuestro último recurso.**

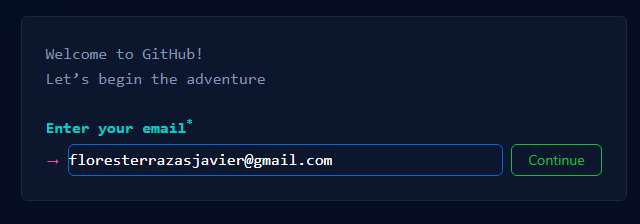
**\*\*\* TERMINAR HACER PRACTICAS\*\*\***

**Git Introducción a GitHub**

Creo una cuenta en GitHub:

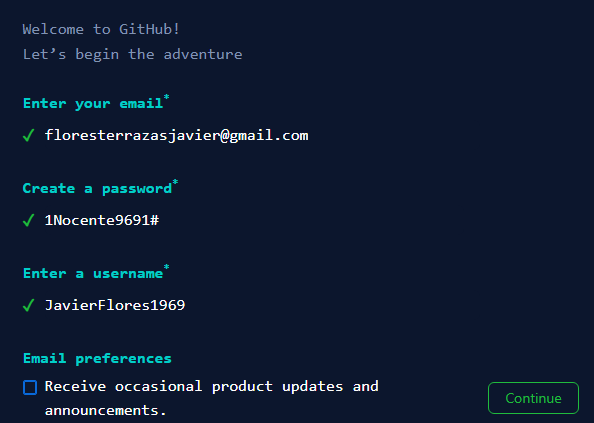


Con este correo: floresterrazasjavier@gmail.com

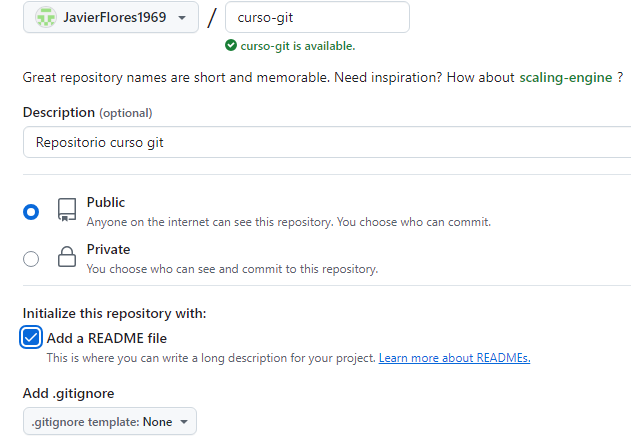


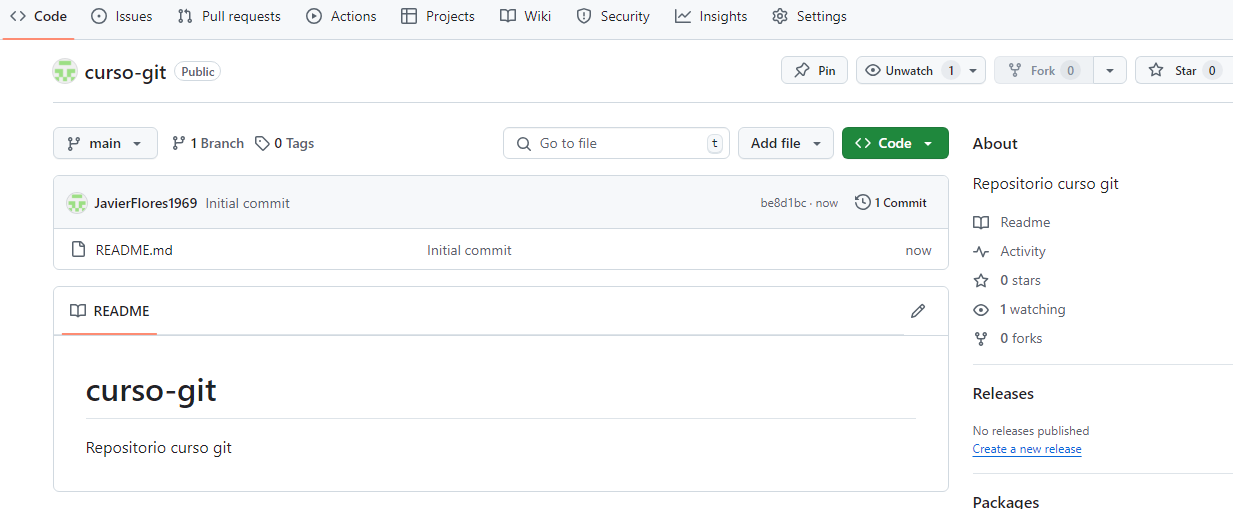
Contrasela: 1Nocente9691#

Username: JavierFlores1969



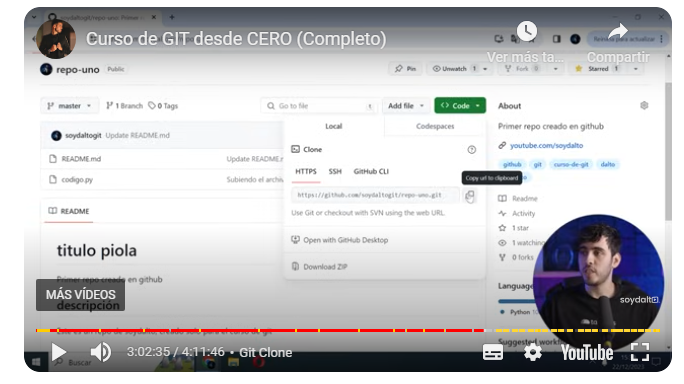
**Crear y configurar Repositorio:**

****

****

**Git Clone**

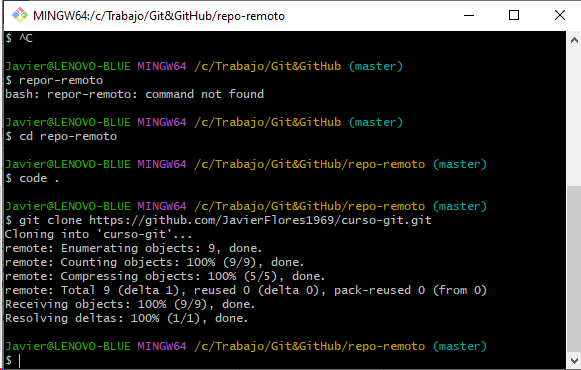
3 Formas de conectarse a un repositorio remoto y clonarlo:



1. HTTPS
2. SSH
3. GITHUB CLI

Ahora para descargar el repositorio copiamos el enlace:

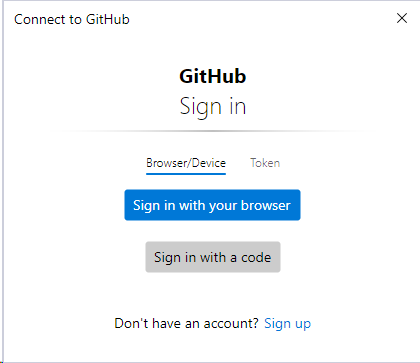
1. Abrirmos git bach
2. mkdir repo-remoto
3. cd repo-remoto
4. cd ..
5. cd .git
6. creo archivo .gitignore
7. Agrego estas 2 lineas a mi archivo:
8. # Ignora la carpeta repo-remoto
9. repo-remoto
10. cd repo-remoto
11. code . Abre VSCode
12. git clone https://github.com/JavierFlores1969/curso-git.git



**Git Push**

Subir archivos al servidor GiHub

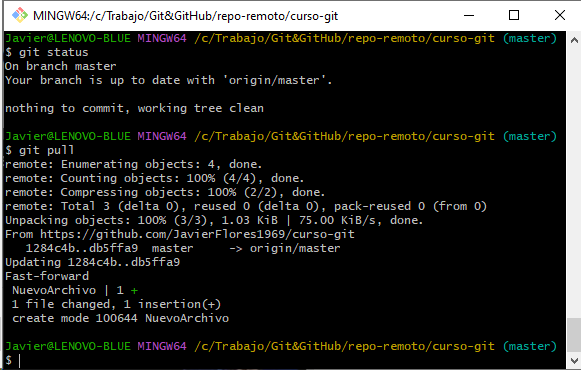
Ejercicio.

1. Estoy en el directorio repo-remoto
2. Creo una archivo de texto
3. git add Subirestearchivo.txt
4. git commit -m “Subo mi archivo a GitHub”
5. git config --global –list Verifico sea el mismo email local y remoto
6. git push origin master
7. Darle enter me muestra esta ventana
8. 
9. Podemo elegir subirlo browser sin embargo a cada vez que necesitemos subir algo pide la clave o podemos generar un token para generar el token
10. Settings
11. Del lado izquierdo al final tenemos “Devolper setting”
12. Personal Access Token
13. Genramos el Token
14. ghp\_PGVqQrKMTKdkiHO1cd1kVD4sT2A8Tu0ZKtKP
15. Seguramente por la versión pide otros datos
16. Termine todo Ok.

**Git Pull – Git Fetch**

Git Pull permite bajar los cambios que ya están en el servidor.

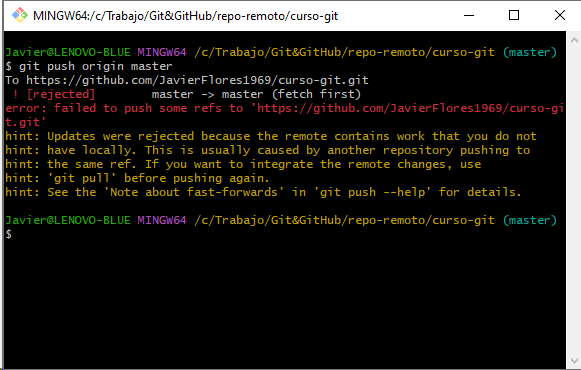
Ejercicio:

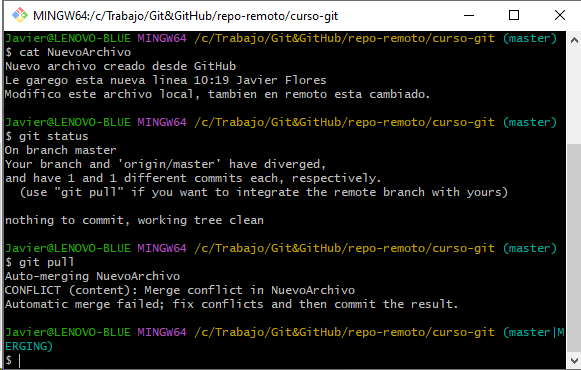
1. Creo un nuevo archivo directamente desde la pagina de github.
2. Ahora git pull
3. 
4. En realidad lo que hacer git pull es traerse los cambios y hacer un git merge con nuestra rama local.

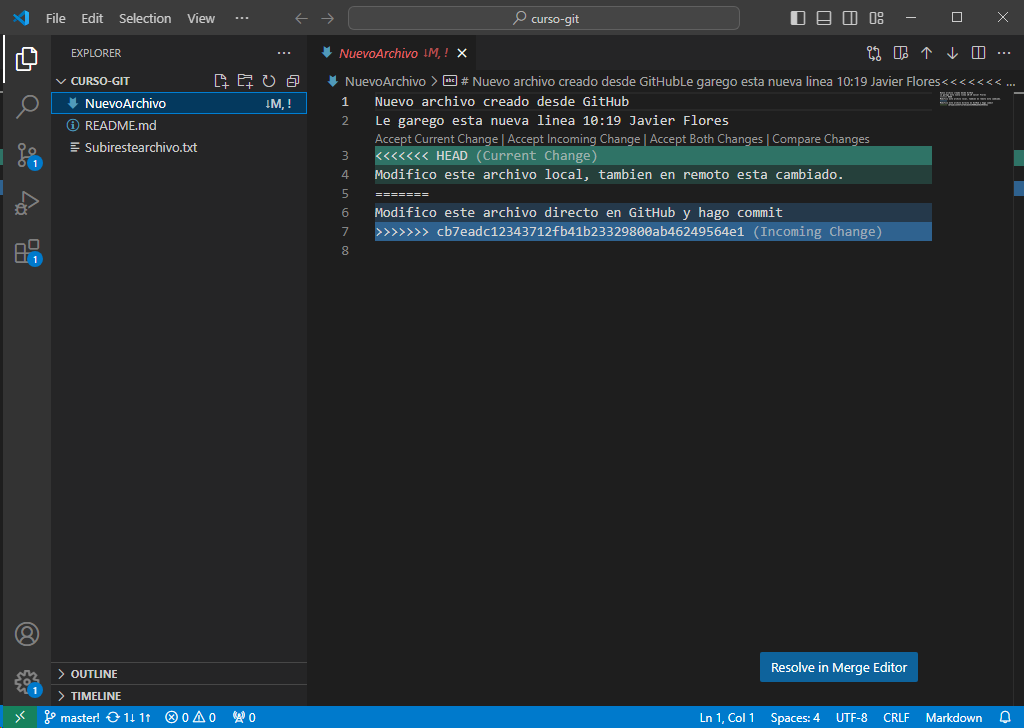
Hago

Otro ejercicio

1. Modifico mi archivo directo en github hago commit
2. Modifico mi arachivo en local y hago commit
3. Hacer git push me muestra conflictos veamos



1. Ahora hago un git fetch
2. No me hizo cambios en el local ahora hago un git pull
3. Muestra que hay conflictos
4. 
5. Tecleo vcode .
6. Me abre Vcode



1. Resuelvo con los conflictos y hago en github
2. Listo

La diferencia entre los comandos git fetch y git pull es que git fetch descarga contenido remoto sin modificar el repositorio local, mientras que git pull descarga el contenido y modifica el repositorio local para reflejarlo.

|  |  |  |
| --- | --- | --- |
|  | **git fetch** | **git pull** |
| Función | Recupere la información más reciente de un repositorio remoto | Descargue el contenido remoto y modifique el repositorio local para reflejarlo |
| Efecto | No modifica el repositorio local | Puede provocar conflictos en el repositorio local |
| Uso | Útil cuando no se quieren traer todos los cambios | Refleja todas las ramas remotas, incluidas las incorrectas o problemáticas, en la rama local |

Después de ejecutar git fetch, se puede comparar la rama local con su rama de seguimiento remoto para ver los cambios en la rama remota. Si se quiere actualizar la rama local con los cambios recuperados, se puede realizar merge o rebase de Git.

**Migrar un repositorio Local a un Remoto**

Ejercicio:

1. Creo una nueva carpeta migrar-repo-remoto
2. Git init
3. Crear Archivo .gitignore con estas líneas:
4. # Ignora la carpeta ignorar
5. Ignorar
6. Reviso mi directorio diga /c/Trabajo/VSCode (master)
7. Creo 3 archivos de 3 Texto
8. git add para los 3 archivos (Resumen.docx)
9. git commit para los 3 archivos
10. En GiHub creamos un nuevo repo
11. git remote add origin <https://github.com/JavierFlores1969/repo-migrado.git>
12. https://github.com/JavierFlores1969/vscode.git
13. Revvisamos que la rama principal sea master
14. git remote show origin
15. git branch -M master
16. git push -u origin master
17. El paso 10 no me funciono tuve que utilizar este comando:
18. git push origin master –forcé
19. git merge origin/master
20. git push --set-upstream origin master
21. Modifique archivo3
22. Git add archivo3.txt
23. Git commit archivo3 –m “Texto”
24. Git push
25. Me funciono tuve que digitar
26. git push --set-upstream origin master
27. Todo subio bien
28. Volvi a modificar archivo 3
29. Git add archvo3.txt
30. Git commit archivo3
31. Git push
32. Ya no marco error
33. Conclusion en un problema del nombre master con main
34. Revisar.

…or create a new repository on the command line

echo "# vscode" >> README.md

git init

git add README.md

git commit -m "first commit"

git branch -M master

git remote add origin https://github.com/JavierFlores1969/vscode.git

https://github.com/JavierFlores1969/vscode.git

git push -u origin master

git push --set-upstream origin master

…or push an existing repository from the command line

git remote add origin https://github.com/JavierFlores1969/vscode.git

git branch -M master

git push -u origin master

1. En Hub Creo https://github.com/JavierFlores1969/vscode.git

2. Agrego un README.md

3. Local creo VSCode

4. git init

5. git branch -M master

6. git remote add origin https://github.com/JavierFlores1969/vscode.git

7. git remote show origin

8. git pull origin master

9. git add Resumen.docx

10. git commit -m "Commit 31/10/24 12:48"

11. git push --set-upstream origin master

12. Todo Bien

1. En Hub Creo https://github.com/JavierFlores1969/git-hub.git

2. Agrego un README.md

3. Local creo Git-Hub

4. git init

5. git branch -M master

6. git remote add origin https://github.com/JavierFlores1969/git-hub.git

7. git remote show origin

8. git pull origin master

9. git add Resumen.docx

10. git commit -m "Commit 31/10/24 12:48"

11. git push --set-upstream origin master

12. Todo Bien

Para revisar a que repositorio remoto esta conectado mi repositorio local:

**git remote –v**

**Que son los FORKS**

Un fork en GitHub es una copia de un repositorio que permite hacer cambios sin afectar al original.

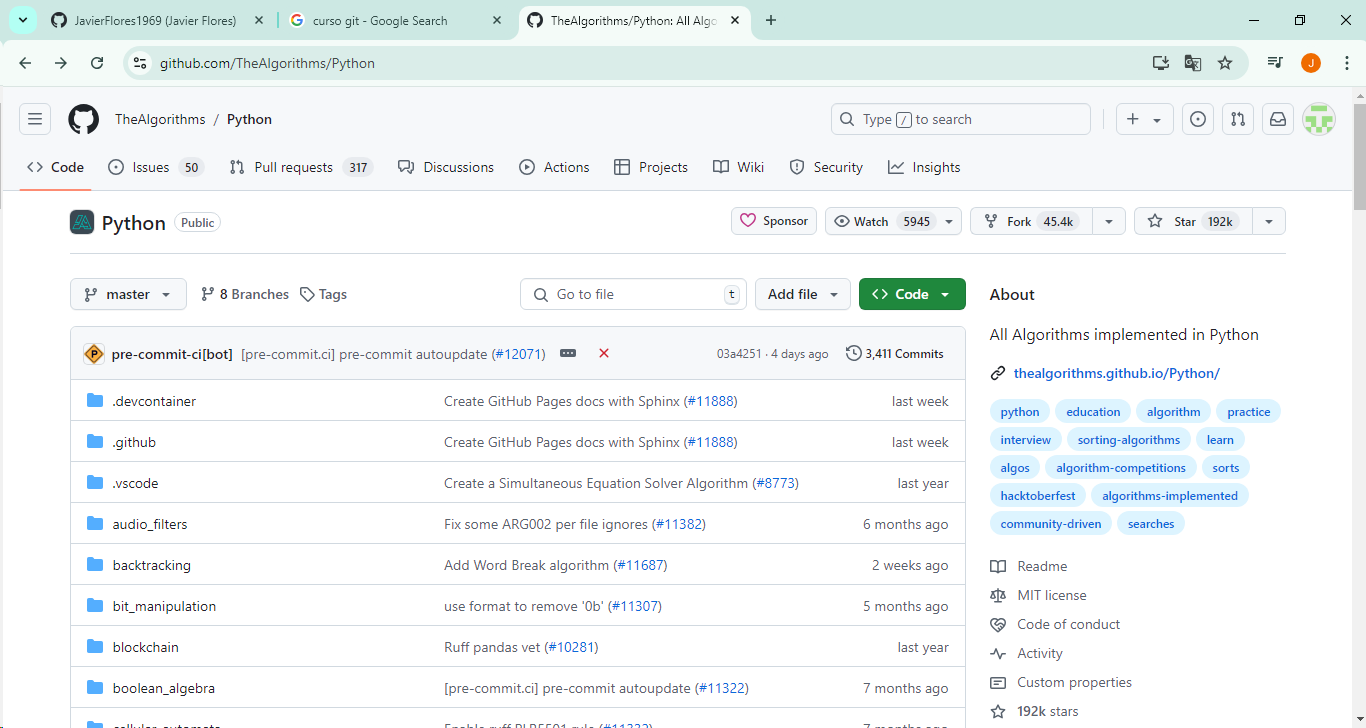
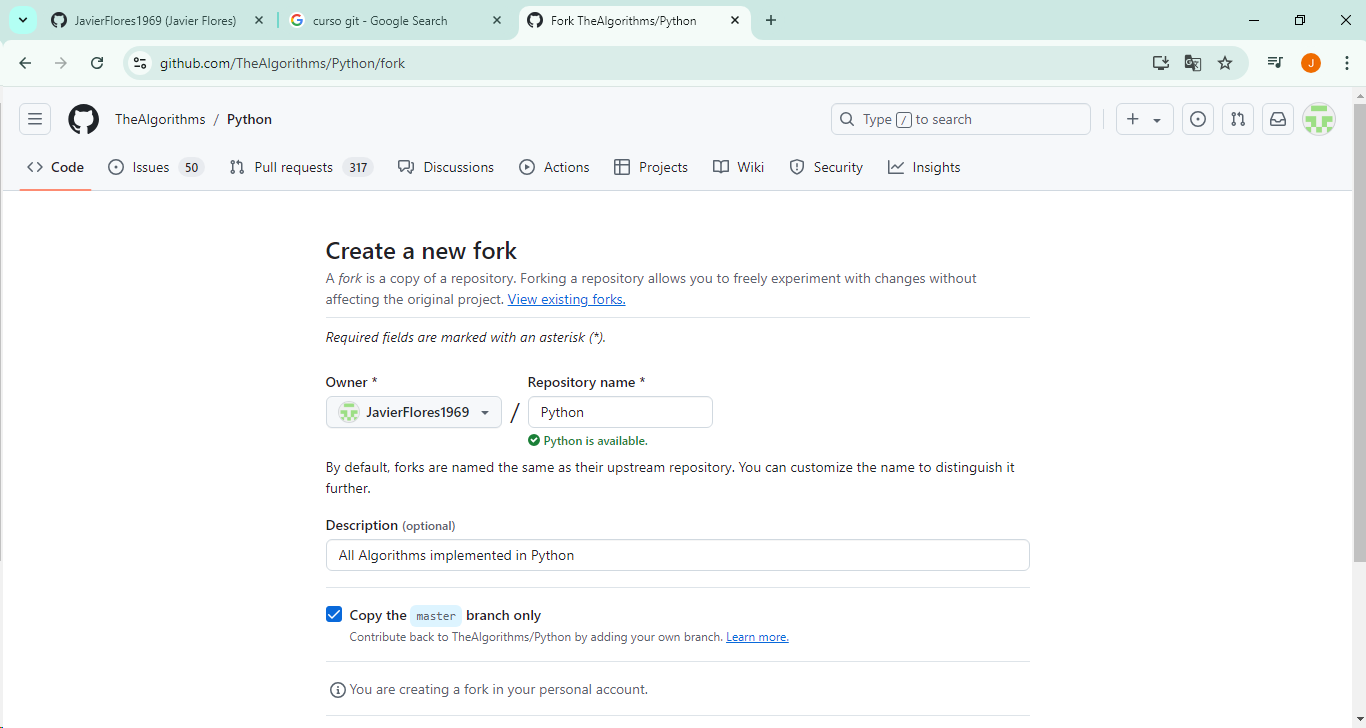
El término fork proviene del inglés y significa "bifurcación".

En GitHub, un fork tiene las siguientes características:

1. Es un repositorio nuevo que comparte la configuración de visibilidad y código con el repositorio original.
2. El repositorio fork es completamente diferente al original, pero comparten alguna historia en común.
3. El creador del fork se convierte en dueño del repositorio y puede trabajar en él con todos los permisos.
4. Los cambios realizados en el fork se pueden hacer públicos.
5. Los forks son una forma de participación abierta en proyectos, ya que no es necesario agregar colaboradores con acceso de escritura.
6. Los forks son comunes en el desarrollo de software de código abierto.

La diferencia entre un fork y un clon es que un fork copia el repositorio entre cuentas de GitHub, mientras que un clon lo copia en la computadora local.

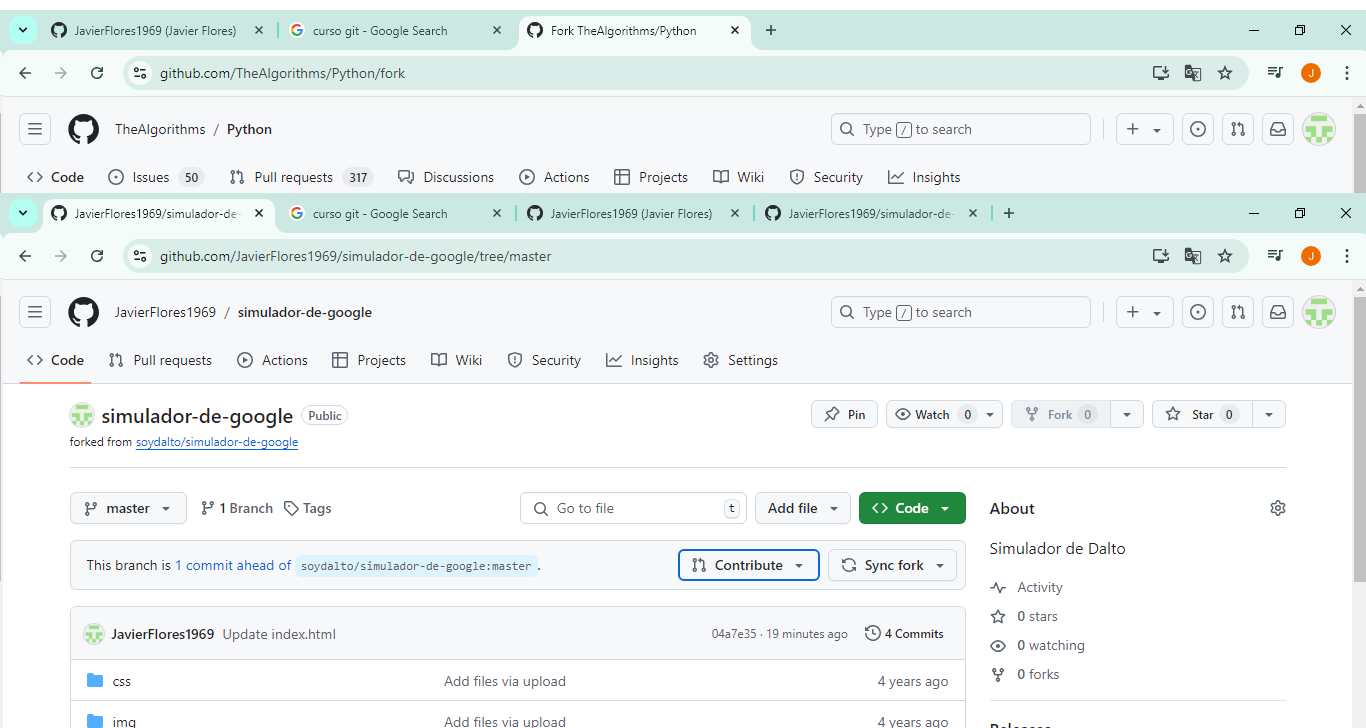
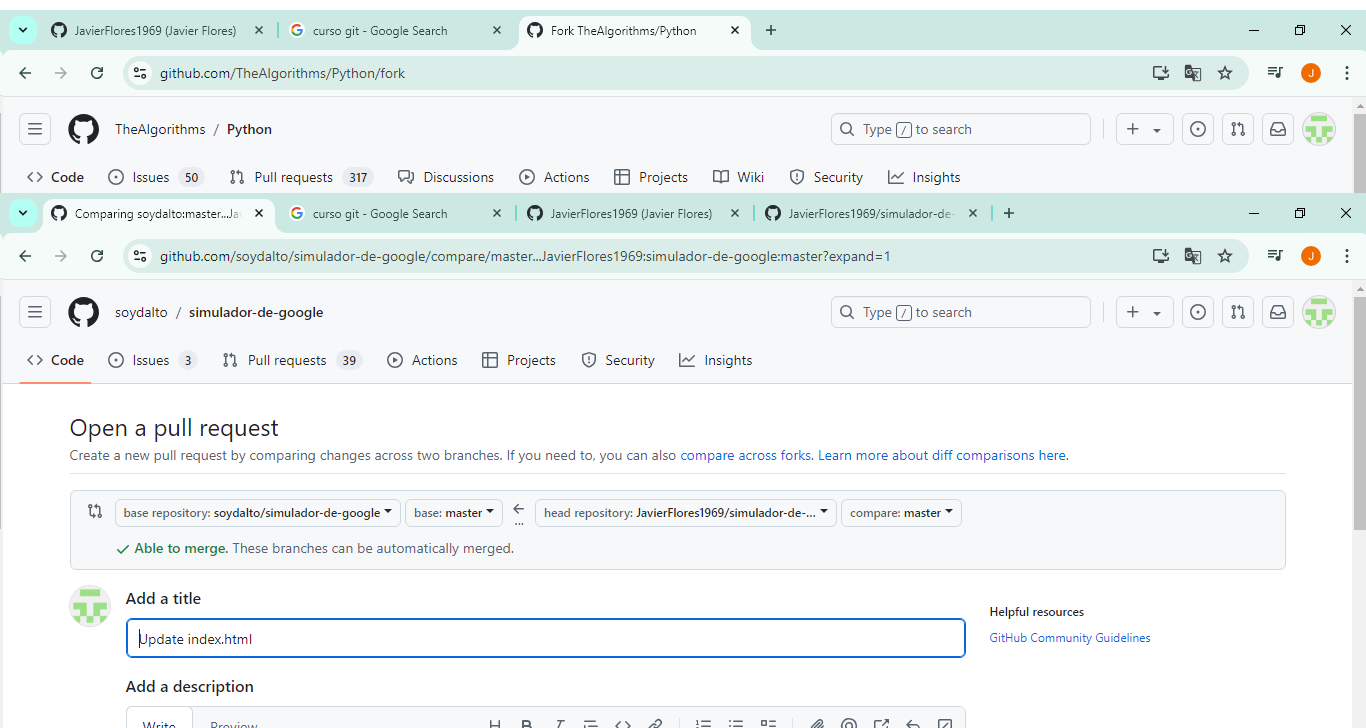
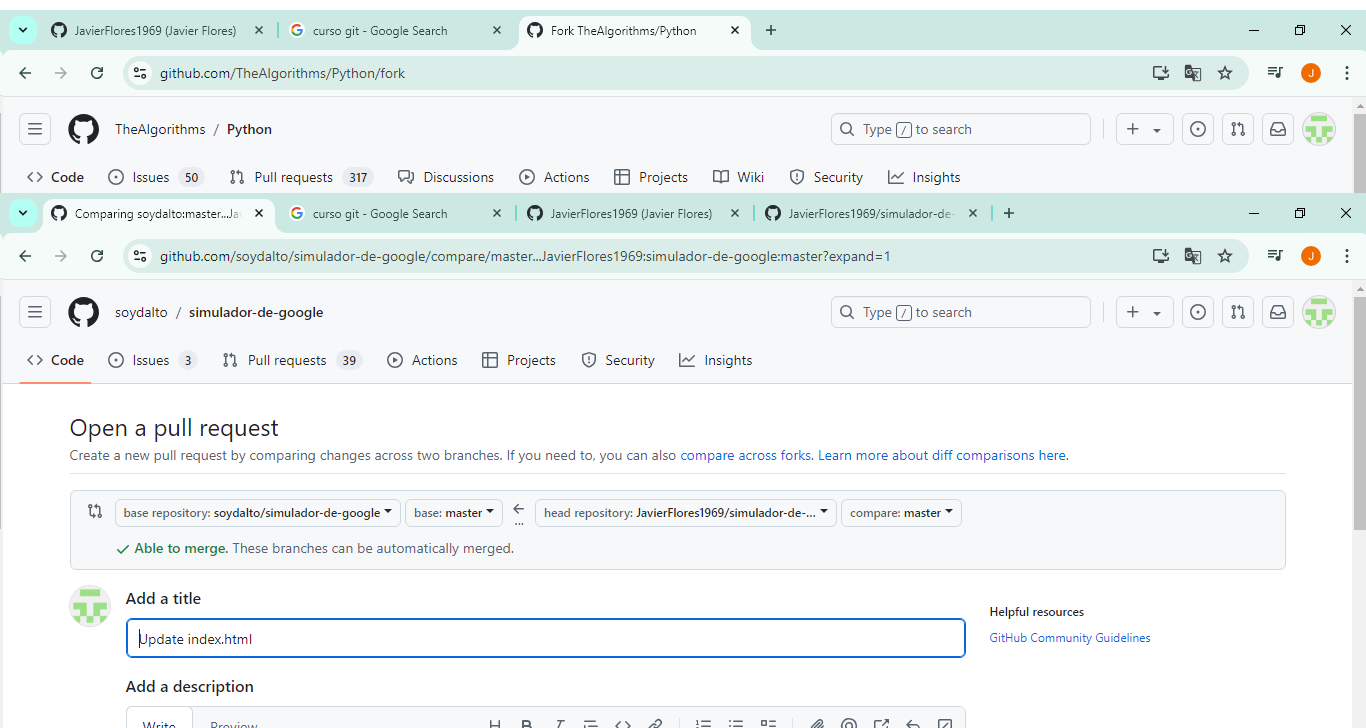
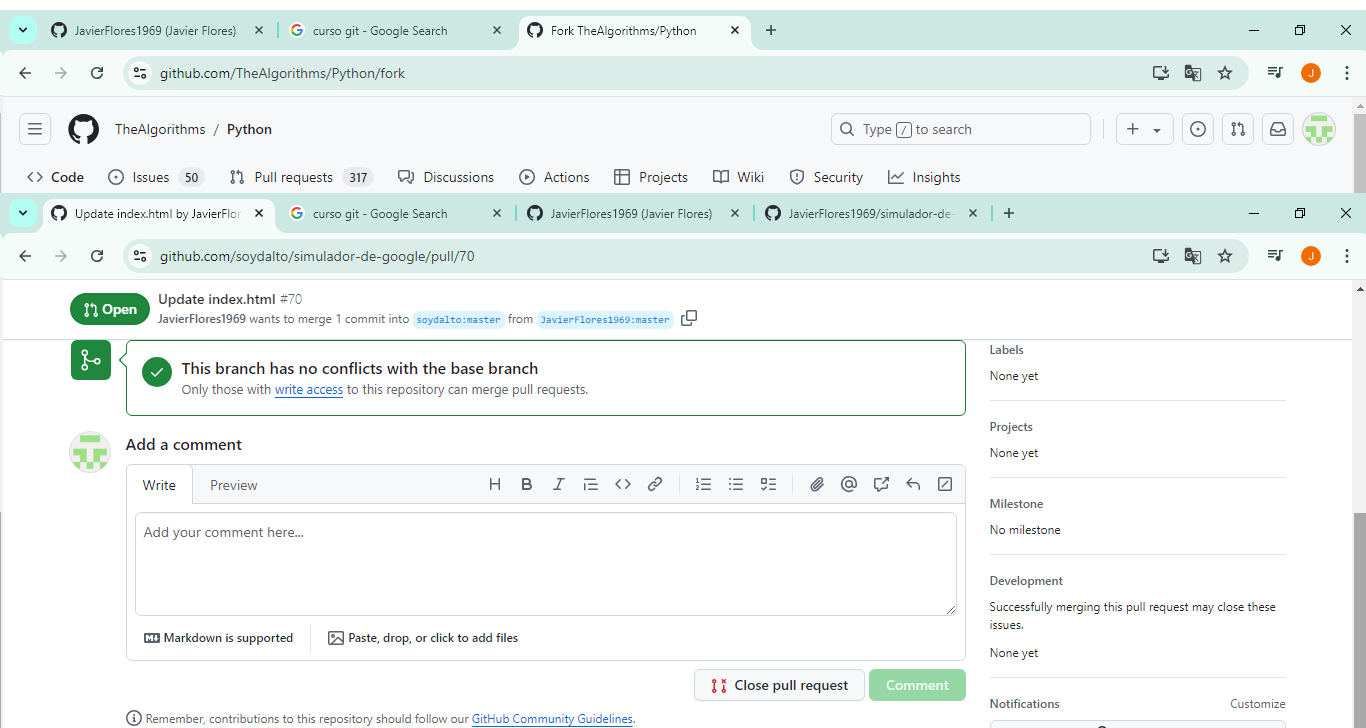
Crear un Fork

1. Buscar el Proyecto que queremos
2. Yo localice este proyecto <https://github.com/TheAlgorithms/Python>
3. 
4. Oprimo el botón y selecciono crear nuevo Fork
5. 
6. Descargamos el Repositorio clonándolo haciendo los cambios necesarios.
7. Despues hacermos un Pull Request enseguida se explica.

**Pull Request.**

En pocas es esto: Me descargue tu repositorio le hice unos cambios y quiero que los integres en el proyecto Original.

Ahora como lo hacemos:

1. A modo de exemplo y hize cambios en un archivo lo comitee muestra que estoy 1 commit mas que el repositorio orginal:
2. 
3. Despues oprimo el botón Contribuir me muestra:
4. 
5. 
6. Ahora mas debajo existe un botón CREATE PULL REQUEST
7. Ahora si el dueño acepta lo cambios hace al mergue con los cambios, sin embargo si no los acepta nosotros podemos cerrar el CLOSE PULL REQUEST.
8. 

**ISSUES**

En computación, el término issue se atribuye a la unidad de trabajo para realizar una mejora en un Sistema informático. Un issue puede ser el arreglo de un fallo, una característica pedida, una tarea, un pedido de Documentación específico y todo tipo de solicitud al equipo de desarrollo.

**Buenas Practicas en Git**

1. Siempre Tener un Git gitnore
2. Siempre configurar user.name y user.email
3. La ramas se deben de crear para un objetivo en especfico
4. Las ramas deben de tener nombres descriptivos
5. Hacer pruebas de manera local
6. No trabajar en la rama principal en el equipo local.
7. Crear ramas cada objetivo no en la rama Main
8. Existen varias formas de fusionar ramas investigar las diferentes opciones \*\*\*TERMINAR\*\*
9. Los Commits deben ser significativos, es decir si me voy a comer hago un commit o me voy dormir hago un commit esto esta mal los commits deben ser significativos.
10. Mantener actualizado el repositorio, hacer un pull request con frecuencia.
11. Mantener un Historial limpio las ramas que ya se usan elimnarlas.

**Proximos Pasos.**

**Git stash**

[**https://www.freecodecamp.org/espanol/news/git-stash-explicado/**](https://www.freecodecamp.org/espanol/news/git-stash-explicado/)

Git tiene un área llamada "stash" donde puedes almacenar temporalmente una captura de tus cambios sin enviarlos al repositorio. Está separada del directorio de trabajo (working directory), del área de preparación (staging area), o del repositorio.

Esta funcionalidad es útil cuando has hecho cambios en una rama que no estás listo para realizarle commit, pero necesitas cambiar a otra rama.

Guardar cambios en el stash

Para guardar tus cambios en el stash, ejecuta el comando:

git stash save "mensaje opcional para ti"

Esto guarda los cambios y revierte el directorio de trabajo a como se veía en tu último commit. Los cambios guardados están disponibles en cualquier rama de ese repositorio.

Ten en cuenta que los cambios que quieras guardar deben estar en los archivos rastreados. Si has creado un nuevo archivo e intentas guardar tus cambios, puede que obtengas el error No local changes to save (No hay cambios locales que guardar).

**Git Cherry-Pick**

**MilesTones**

**Tags**

**Rebase - Squash**

**SubModulos**

**Hooks**

**Bisect**

**GitFlow**

**GitHub Flow**

**JFT2**

**https://ed.team/blog/trabaja-en-dos-o-mas-ramas-de-git-al-mismo-tiempo**

**https://www.atlassian.com/git/tutorials/rewriting-history#:~:text=The%20git%20commit%20%2D%2Damend,message%20without%20changing%20its%20snapshot.**

**El sistema de archivos de Git**

<https://apuntes.de/git/el-sistema-de-archivos-de-git/#gsc.tab=0>