

22 de abril de 2020

David Martin Vergues

Curso Git y GitHub

UDEMY course

[Tema 1 2](#_Toc1223215323)

[¿Cómo funciona GIT? 2](#_Toc1492686274)

[Primeros pasos 2](#_Toc730356522)

[Tipos de add 7](#_Toc925355447)

[Como revisar el log 8](#_Toc1223774518)

[Creando alias 9](#_Toc572744655)

[Tema 2 10](#_Toc831858485)

[Más allá de los fundamentos 10](#_Toc1788937057)

[Uso de Diff 10](#_Toc1019428568)

[Reset, sacando del stage 11](#_Toc872785884)

[Checkout – para deshacer cambios 11](#_Toc186381187)

[Modificar mensaje del commit (--amend) 11](#_Toc47985178)

[Reset - -soft modificando un commit 12](#_Toc2091680053)

[Reset - -mixed - -hard - - reflog 17](#_Toc581716825)

[Renombrar y eliminar archivos con GIT 20](#_Toc1690459012)

[Con terminal 20](#_Toc1727960354)

[Ignorar archivos en git - .gitignore 22](#_Toc1091589345)

[TEMA 3 - RAMAS CONFLICTOS Y TAGS 23](#_Toc922696069)

[Definición 23](#_Toc531711276)

[Fast-forwars ejemplo 23](#_Toc2120344743)

[Merge automático 25](#_Toc1993583338)

[Merge Manual o con conflictos 26](#_Toc559117073)

[Tags o etiquetas 27](#_Toc1659835349)

[TEMA 4 - GIT stash y rebase 29](#_Toc431831764)

[Apendices 30](#_Toc1395546916)

[Evitar que GIT pida credenciales en cada push a GitHub 30](#_Toc1610021439)

# Tema 1

## ¿Cómo funciona GIT?

Básicamente con git conseguimos tener un proyecto distribuido. A ese proyecto de le denomina repositorio. Distribuido significa que cada integrante del proyecto tiene una copia de ese repositorio en local.

Git establece una línea del tiempo desde que se crea el repo hasta el día de hoy. Durante esa línea del tiempo vamos tomando fotografías de como se encuentra el proyecto, los llamados commits, de tal manera que siempre podremos volver a alguno de esos momentos.

## Primeros pasos

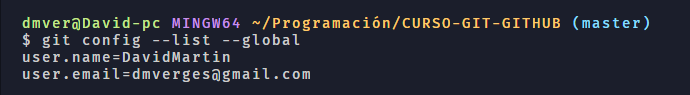
Lo primero que debemos hacer es guardar un nombre de usuario y un mail en nuestra máquina para presentarnos quienes somos en GIT, estas credenciales serán las usadas en todos los repositorios que tengamos en local. Podemos optar por tener un nombre y un mail específicos por repositorio. Todo depende si añadimos al comando la palabra global.

Usamos: config global





Para saber si se ha guardado bien hacemos



Podemos ir directamente al archivo q almacena esta información en .gitconfig y alterarla(no recomendable)





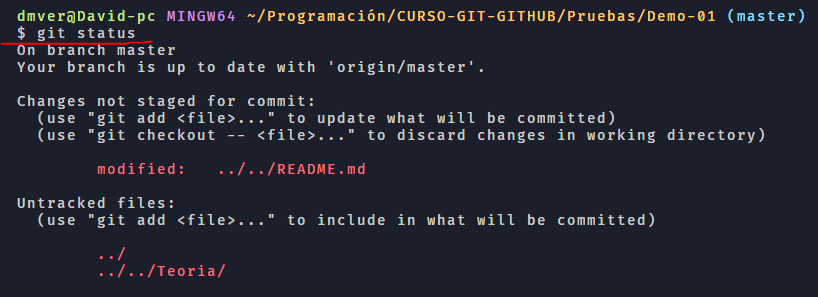
Usamos: init status add commit .gitignore

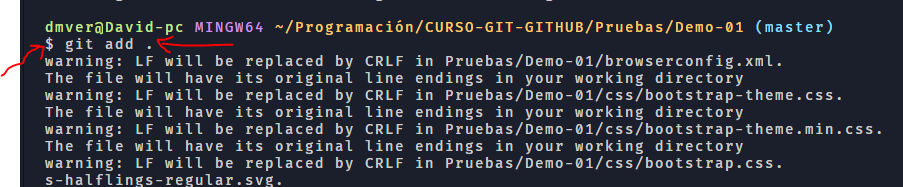
Cuando hacemos un init en nuestro repositorio local se crea la carpeta git donde estará todo lo necesario para guardar nuestros cambios.



Luego creamos nuestros archivos

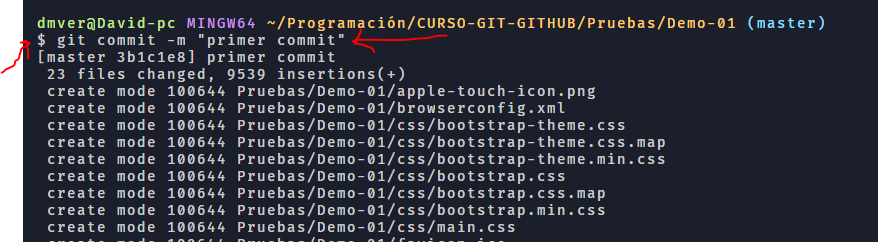
Cuando queremos ver el estado de git usamos git status y si hay cambios que hay que incluir usamos git add.





Al importar un repositorio hecho en Linux en un entorno Windows el final de las líneas en diferente, en Linux usan LF (line feed) y en Windows CR (carriage return).

Para tomar la foto del proyecto actual usamos commit -m “mensaje”



Con git init inicializamos el repositorio. Creamos los archivos necesarios. Con git add . ó nombre del archivo añadimos al stage esos archivos, después hacemos un commit que toma una foto de todos los archivos que hay colocados en el stage (escenario) para que git pueda registrar como se encuentran.

Si queremos descartar algún tipo de archivo, para q no sea seguido por git creamos un archivo .gitignore con la ruta a los archivos que no queremos controlar.

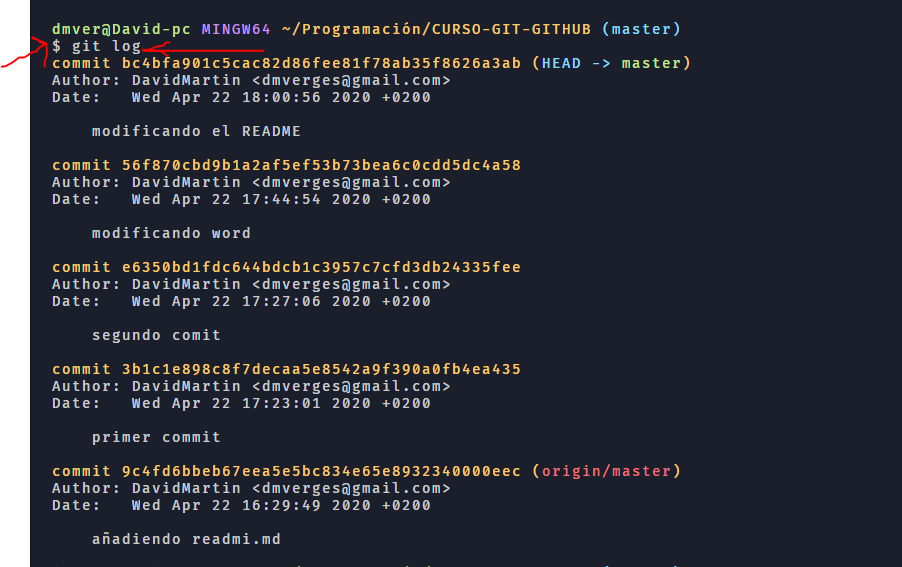
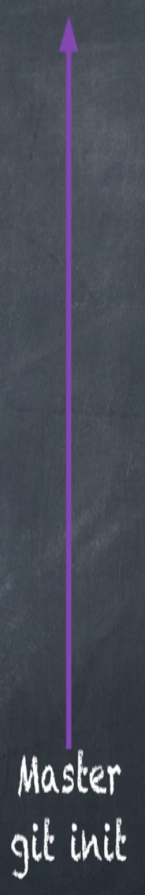


Usamos: checkout log

Imaginemos que por error modificamos un archivo o eliminamos su contenido. Con el comando checkout recuperamos el proyecto como estaba en el último commit. Esto también sirve para recuperar carpetas borradas

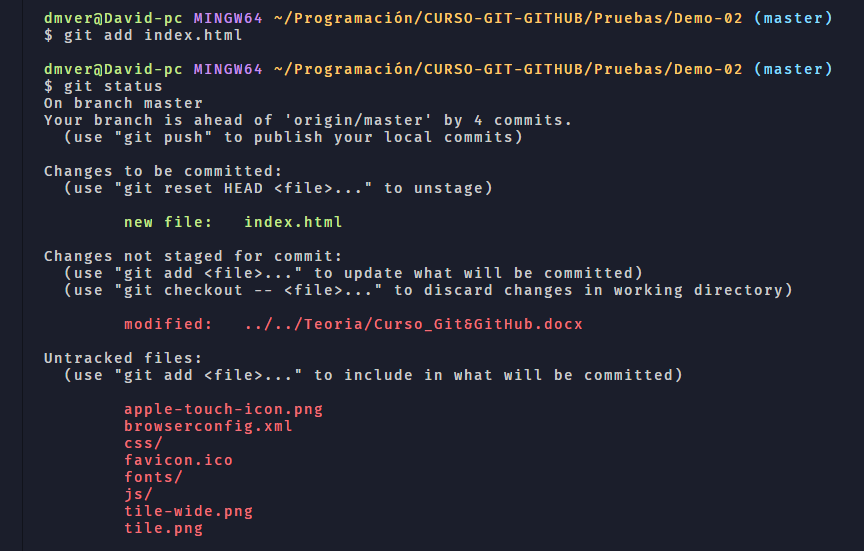


A medida que vamos haciendo commits se va creando un registro con toda la información. Para acceder a este registro usamos log



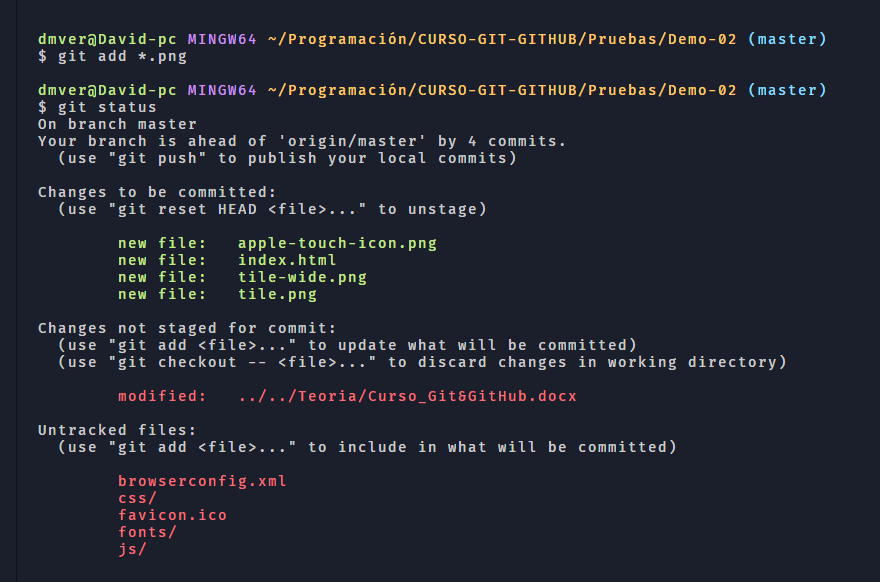
Lo correcto sería hacer commits por archivos, es decir si tenemos varios archivos modificados incluir cada uno de los archivos por separado y hacer el commit. Por jemplo hago el commit solo del index.

Usamos: add \*.png add css/ git add -A reset

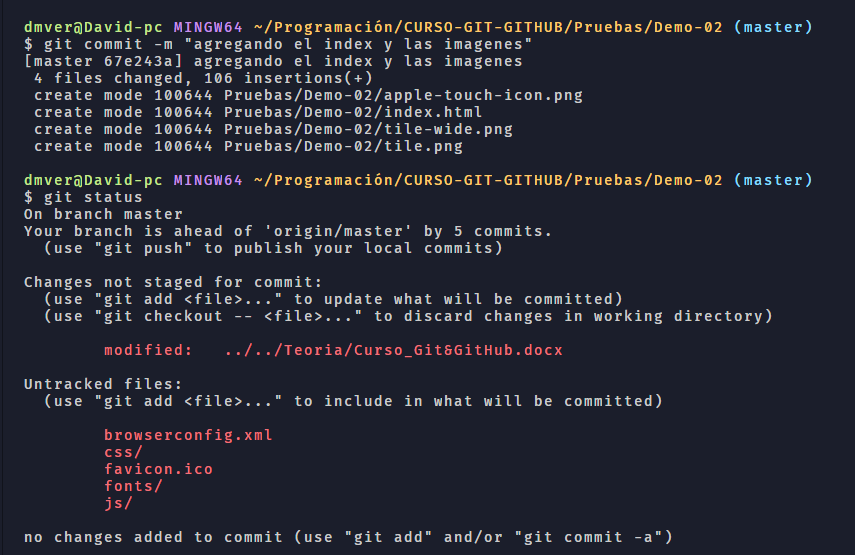


En la imagen vemos como index está en el stage y el resto de archivos en rojo todavía no.

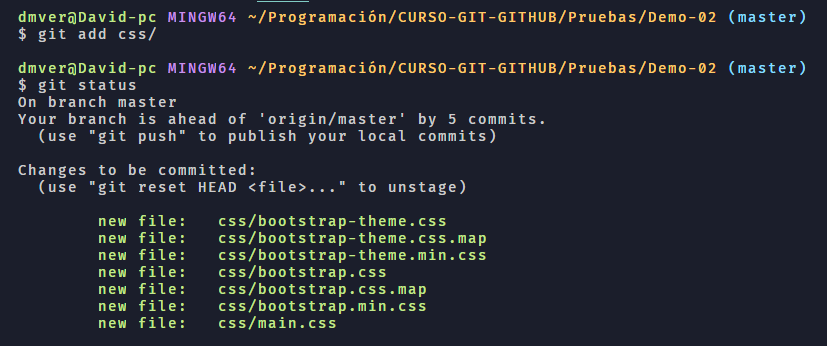
Para añadir todos los archivos de un mismo tipo git add \*.png



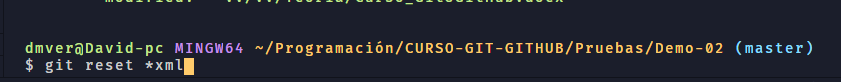
Hacemos el commit indicando que trackeamos el index y las imágenes.

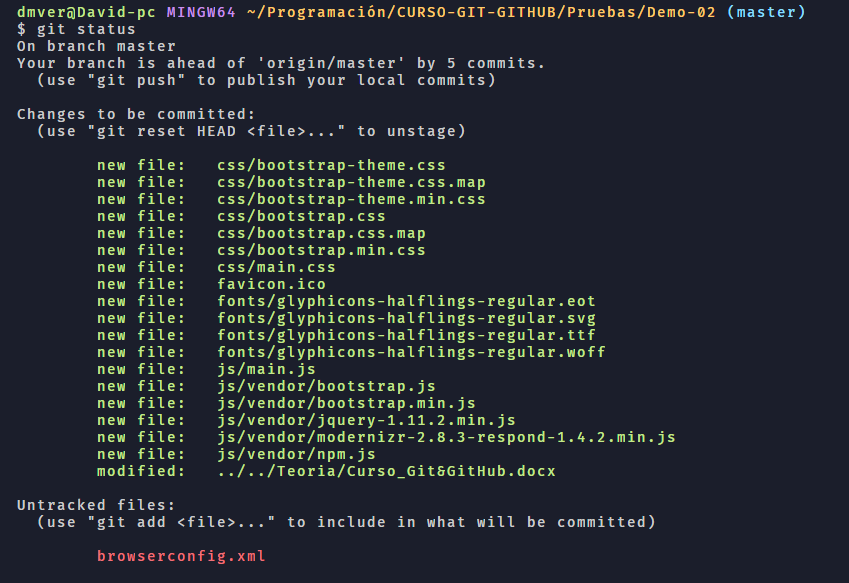


Agregar toda una carpeta el stage



Con git add -A añadimos todos los archivos con cambios al stage pero si queremos descartar uno antes del commit hacemos un reset





Y lo podemos añadir a parte. Si hacemos un log veremos todo el regustro



### Tipos de add

Git add “\*.txt” => añade al stage los archivos txt modificados en todo el proyecto

Git add \*.txt => añade al stage los archivos txt modificados sólo en el directorio actual

Git add . => agrega todos los archivos modificados del mismo directorio

Git add - -all

agrega todos los archivos modificados en todo el proyecto

Git add -A

Git add <file1.txt, file2.js> => agrega una lista de archivos.

Git add carpeta/\*.pdf => agrega todos los archivos de un tipo dentro de una carpeta

Git add carpeta/ => agrega todos los archivos dentro de una carpeta

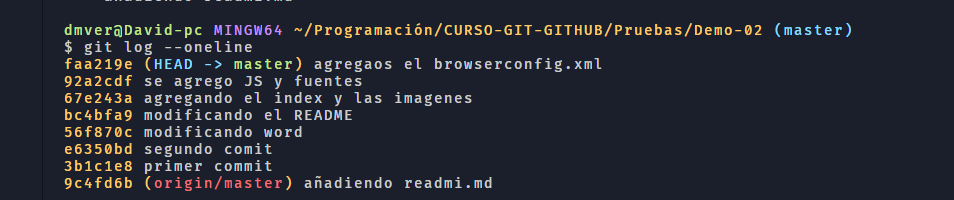
### Como revisar el log

Usamos: log --oneline --decorate --all –graph status -s -b

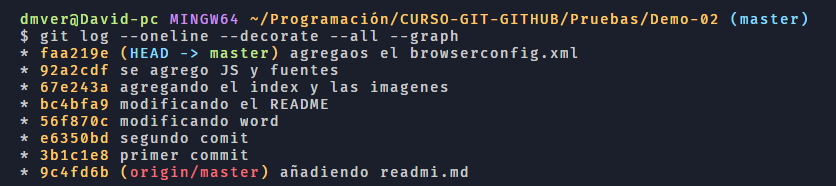
Cuando hacemos un log vemos el HEAD que nos indica el último commit de la rama en la que estamos, en este caso master



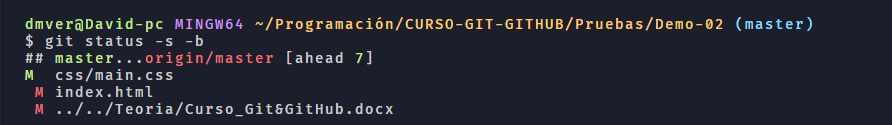
Oneline



Todos estos parámetros me servirán para ver mejor el log cuando tengamos ramas y merges…



status -s (silence) -b (brach) reducir la info del status

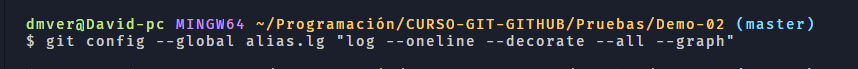


En M rojo modificados pero fuera del stage y M verde modificado pero dentro del stage

### Creando alias

Git s git lg

Generar alias de forma global



Lo que va detrás del punto en alias es cómo se llamará mi alias, en este caso “lg”



Para ver todas las configuraciones que vamos haciendo a nivel global hacemos un

git config - -global -e ó también git config –global -l



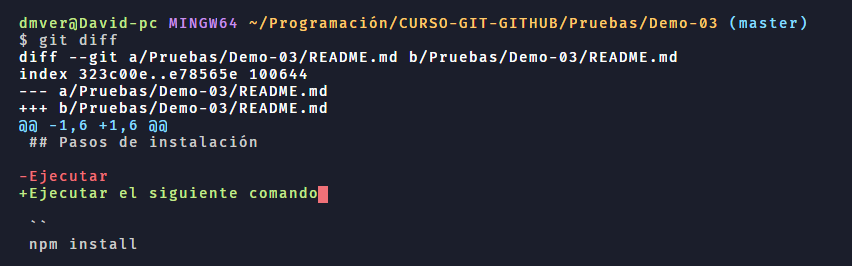
# Tema 2

## Más allá de los fundamentos

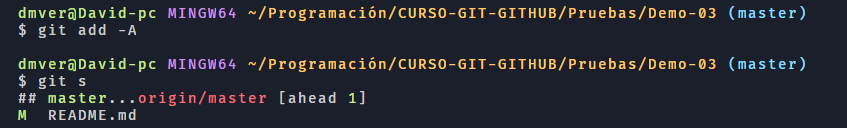
### Uso de Diff

Usamos: git diff –staged checkout – commit -am commit –amend reset --soft

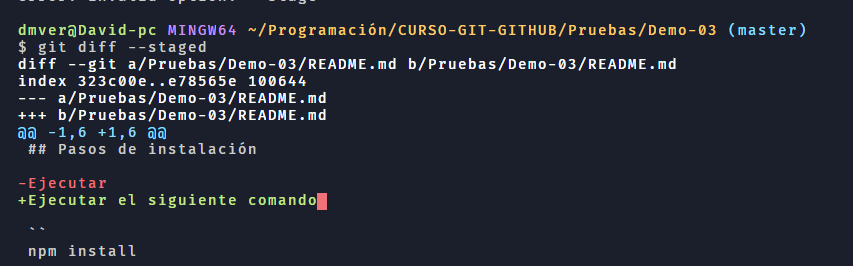
Para saber que modificaciones hicimos en un archivo hacemos un diff, nos da las modificaciones entre el último commit y el momento actual.



Incorporamos el archivo al stage

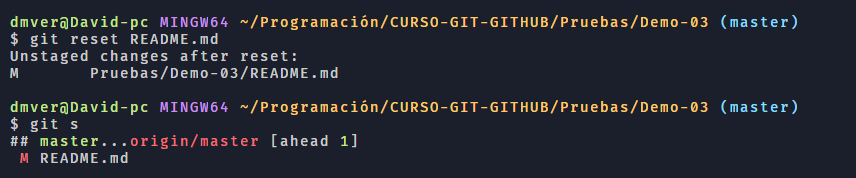


Ahora al ejecutar diff no nos muestra nada para saber las modificaciones de una archivo que ya está en el stage le añadimos –staged



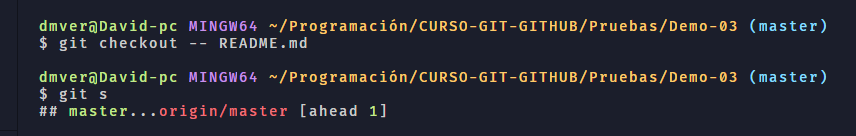
### Reset, sacando del stage

Ahora después de ver las modificaciones consideramos que es mejor la versión anterior así q lo sacamos del stage con reset



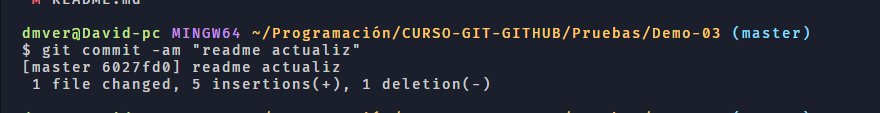
### Checkout – para deshacer cambios

Y lo volvemos a la versión anterior con checkout – nombreArchvo



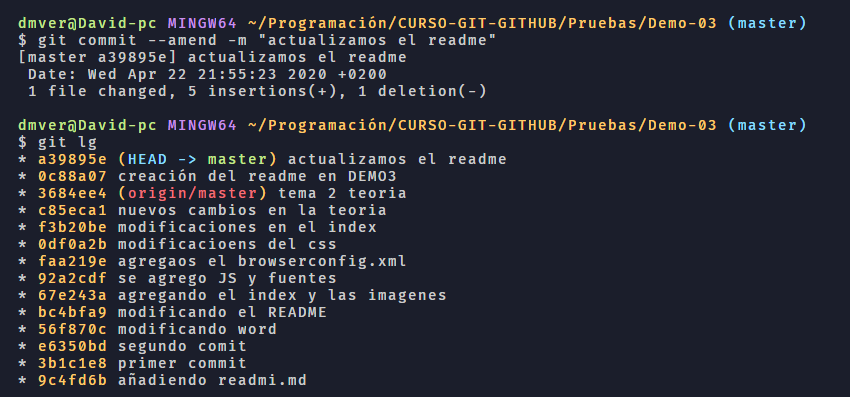
Vemos como al hacer status ya no hay ningún archivo modificado pendiente de poner en el stage.

Si volvemos a modificar el archivo README.md añadiendo texto, como es un archivo ya en seguimiento (tracked) podemos hacer un add + commit al mismo tiempo con commit --am.



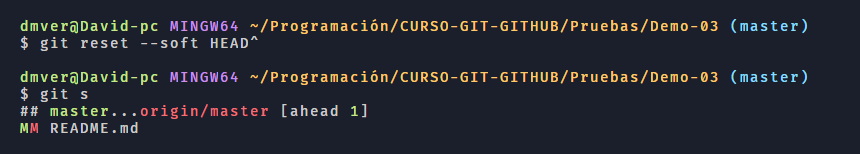
### Modificar mensaje del commit (--amend)

Como vemos el mensaje del commit está mal escrito para poder modificar un mensaje del commit usamos commit –amend. Esto lo que hace es sustituir el mensaje del último commit por el que nosotros introducimos.

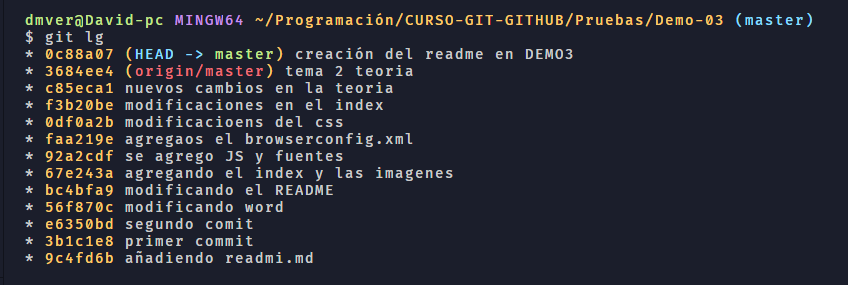


### Reset - -soft modificando un commit

Queremos modificar el archivo README pero esas modificaciones deben estar asociadas a ese commit para ello usamos reset –soft HEAD^ le coloco el ^ porque quiero volver al commit justo anterior a donde apunta HEAD, que es último.



Vemos como el README está a la vez en el stage y modificado eso es porque hemos movido el HEAD



Pero seguimos teniendo el commit

pero HEAD apunta al commit de abajo. Entonces modificamos el archivo y volvemos a hacer un commit lo que llevará los cambios al commit de arriba con las nuevas modificaciones incluidas pero el commit a39895e queda colgado, sin continuidad en nuestra línea y se genera un nuevo commit.

HEAD-> a8d5118

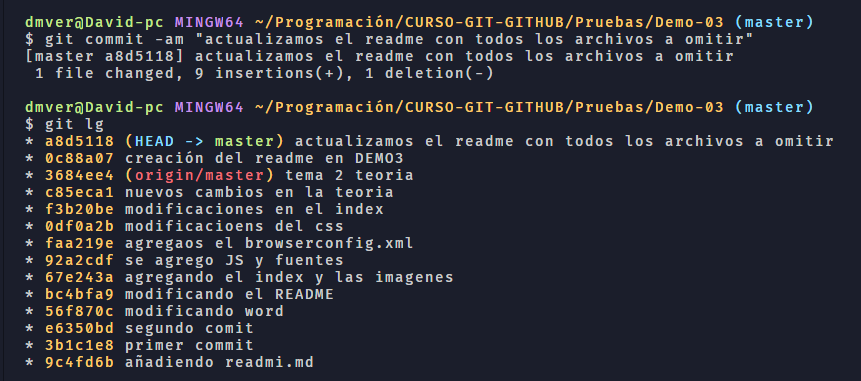
(commit con las modificaciones correctas)

A39895e (commit eliminado)

0c88a07

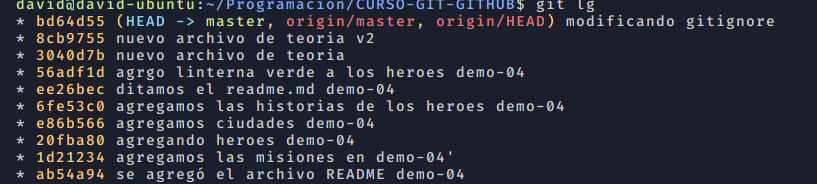
(commit anterior)

Así si hacemos un log queda así



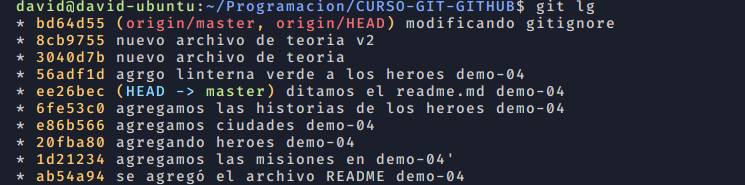
En lugar de usar HEAD^ podemos añadir el id del commit al cual queremos volver

Yo tengo este log:

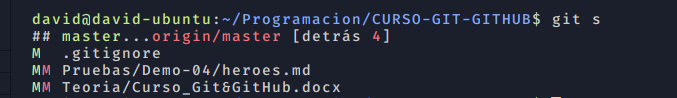
y quiero modificar el commit donde agrego linterna verde entonces voy al commir anterior el ee26bec



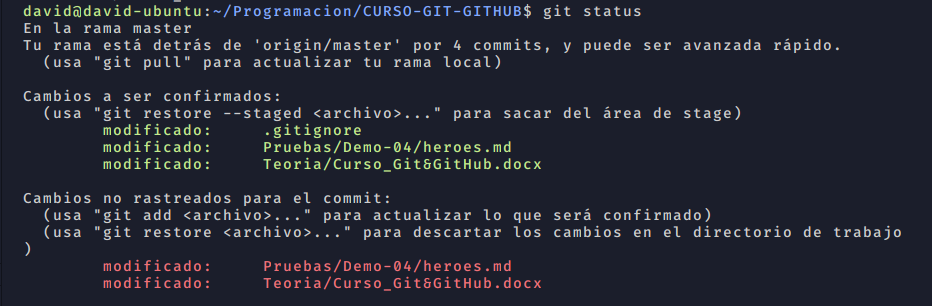
Ahora vemos como el HEAD apunta a este commit



Una vez aquí hago las modificaciones en el archivo del commit (heroes).



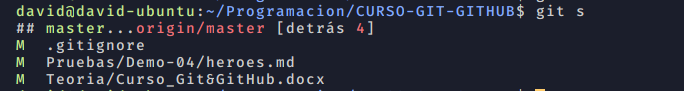
Si miramos la forma extendida se entiende mejor



Tenemos que hay archivos incluidos en el commit pero como hemos vuelto a ese commit y hecho nuevas modificaciones nos lo pone también como pendiente de agregar al commit por eso aparece la doble **MM**

Agregamos los cambios al stage

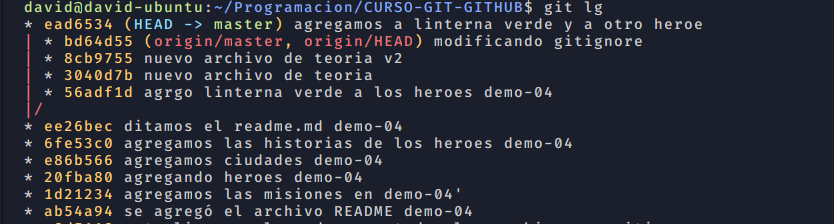




Y hacemos el commit



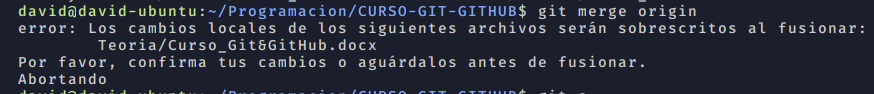
Ahora si hacemos un log vemos el camino



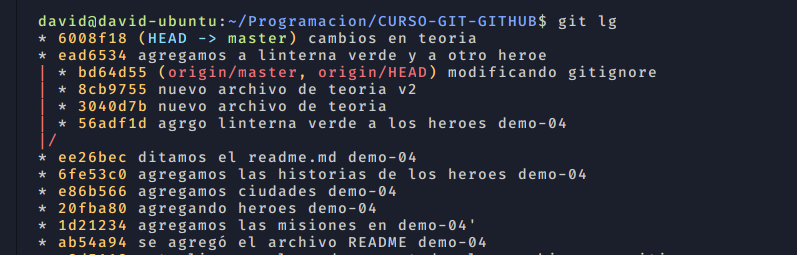
Usamos: merge

Ahora vemos como el origin con sus commits que quiero mantener están en otra línea que el local por lo que hay que mergearlos para incluir todos esos cambios con la nueva línea local.

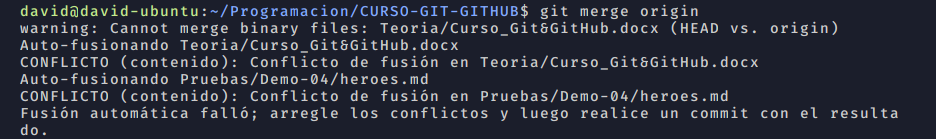
Antes de nada guardo todos los archivos abiertos como por ejemplo el word con la teoría por eso me obliga a hacer añadirlo al stage



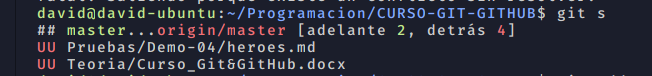
Lo que hago es mirar el status veo que efectivamente teoria esta fuera del stage lo agrego y hago un commit



Ahora en local no tengo archivos con cambios y puedo proceder al merge



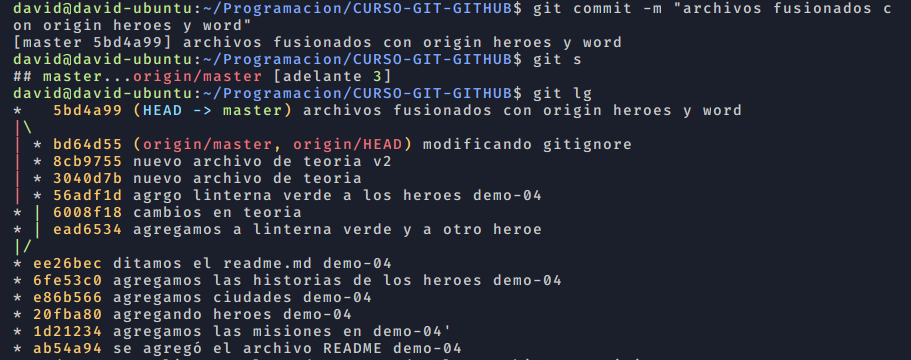
Obviamente cuando intento el merge me dice que hay conflictos por que los archivos modificados en local no concuerdan con los de origin(los de gitHub) estos archivos son el de heroes.md y el word de teoria, si hago un status veo como estan marcados con el flag **UU** que significa updated/unmerged.



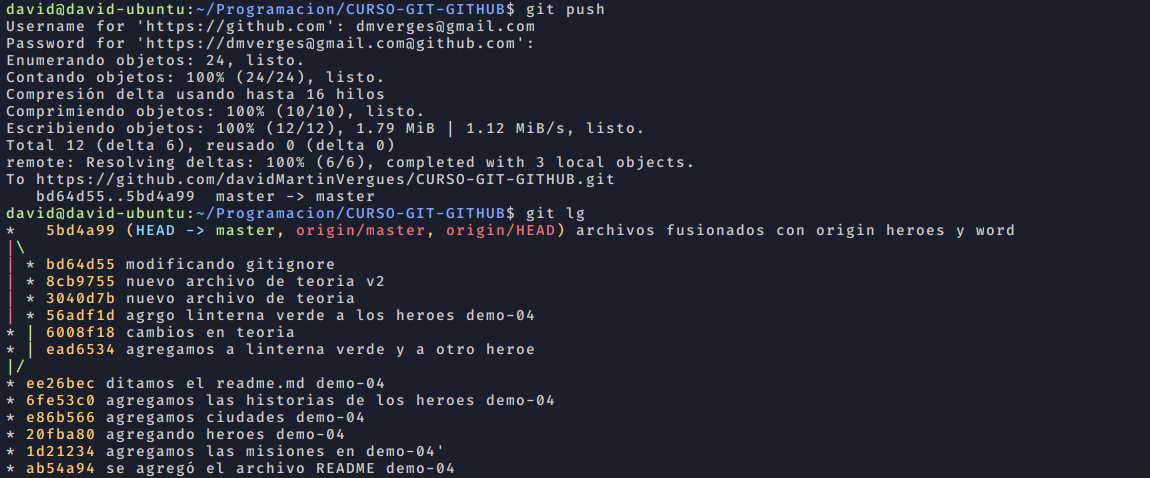
Pero que pasa que el .md el vsCode me muestra las opciones que tengo de manera grafica y me compara ambos archivos así yo puedo elegir con que cambios me quedo así que resulelvo ese comflicto y lo mergea. Así que decido no hacer nada con el word y añado ambos archivos directamente al stage



Aquí ya me dice que master (local) está adelantada 2 commits y origin/master (la de gitHub) atrasada 4. Así como me decía anteriormente q después de solventar los errores hiciese un commit y aunque solo he solventado el de heroes.md hago el commit a ver que pasa Y PARECE QUE FUNCIONA.



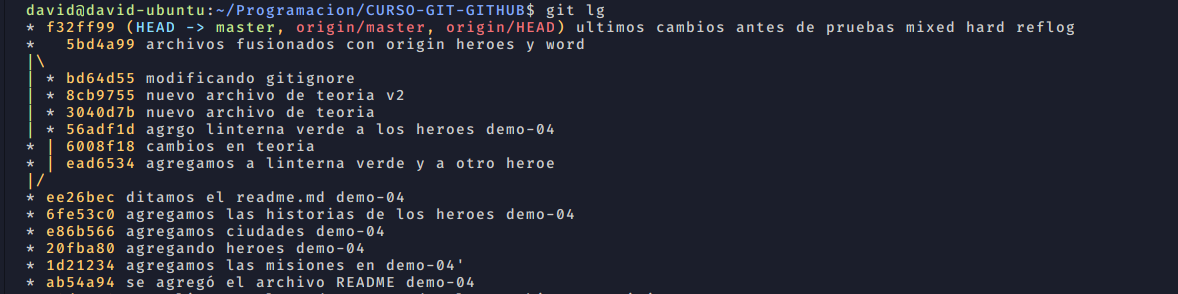
Ha incluido en la misma línea de tiempo los commits del local con los de origin y al hacer el último commit parece que todo acabe allí pero me head local apunta al último commit, lo que es correcto, pero origin está por detrás por lo que tengo q hacer un push para igualarlos.



Ahora ya tengo HEAD/master y origin apuntando al mismo commit.

### Reset - -mixed - -hard - - reflog

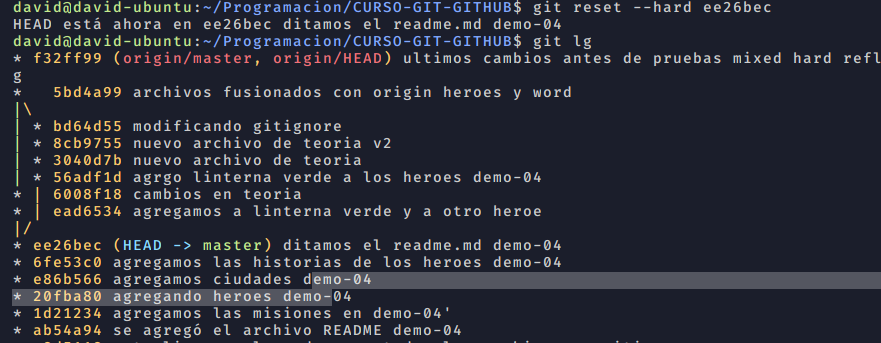
Ahora lo tenemos todo bien y actualizado pero imaginemos que todos esos commits no nos sirven ya y queremos volver a un commit anterior para volver a comenzar desde allí para eso usamos reset mixed



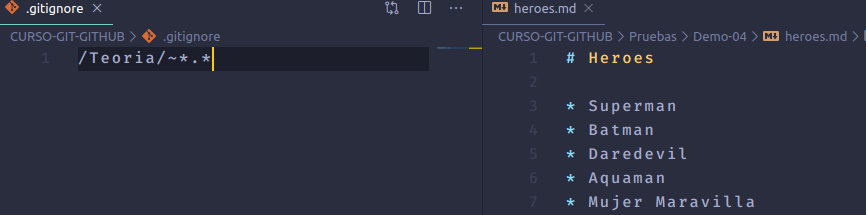
Yo tengo esto y quiero volver al punto dnd empieza la bifurcación ee26becpara ello uso un reset - - mixed

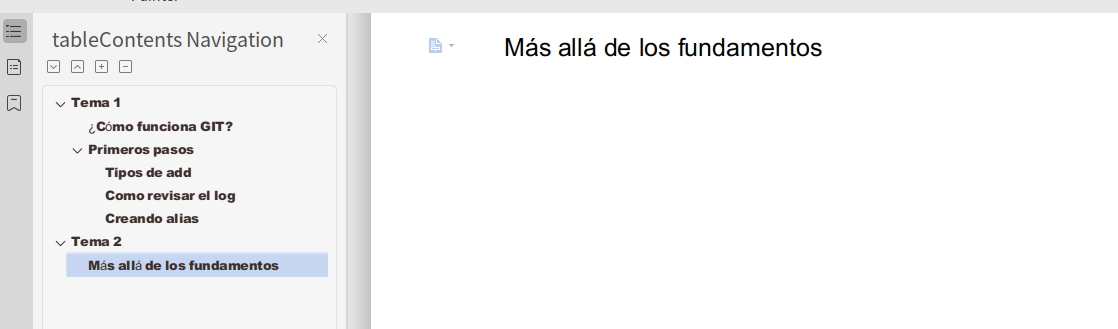


Vemos el head que apunta al commit escogido también aparece los archivos modificados entonces pero en pricipio todos los cambios se mantienen como los teníamos pero lo único es que están fuera del stage. Ahora si realmente queremos eliminar esos cambios y quedarnos en ese commit para trabajar a partir de ahí debemos hacer un reset --hard al mismo commit

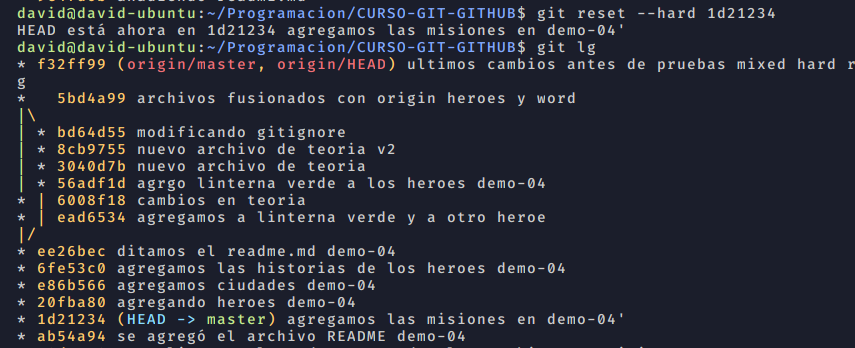


En el árbol sigue igual pero si vamos a los archivos implicados como .gitignore heroes.md y el word se han perdido todos los cambios hechos después de ese commit ee26bec.

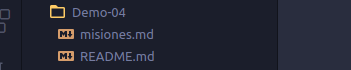




Sigamos bajando en el árbol y veremos como van desapareciendo los archivos, bajaré hasta donde agregamos las misiones

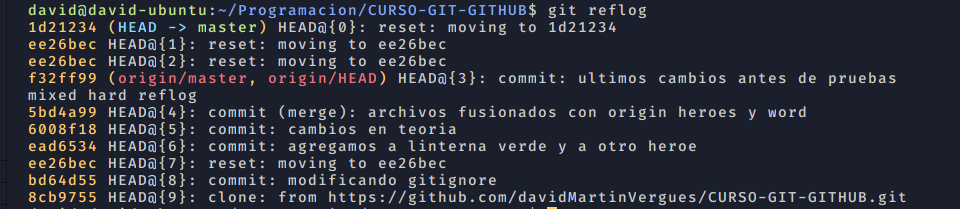


El HEAD lo tenemos casi al principio del proyecto y nos ha desaparecido casi todo

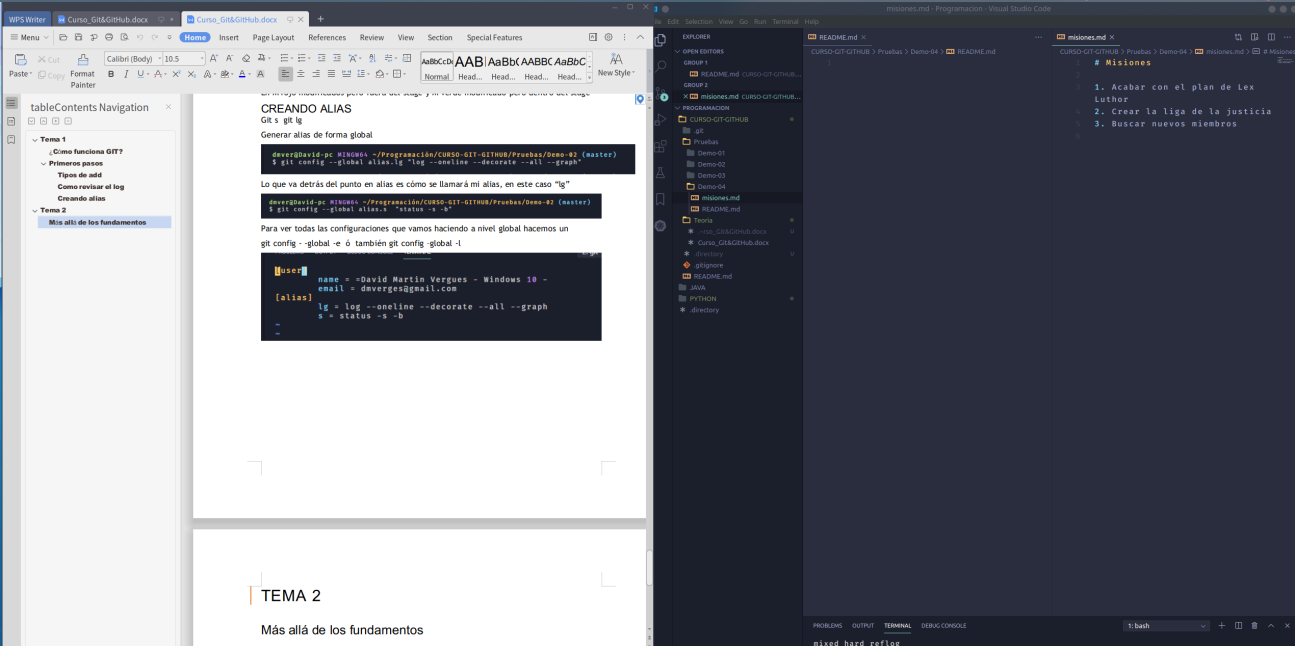


Solo tenemos esos dos archivos en el proyecto y la teoría también se ha perdido.

Pero ahora queremos recuperar todo lo borrado pero en el log ya no aparecen (a mi sí pq trabajo con gitHub y como el origin/master está al principio y como no hice push pues lo mantiene) pero para git mantiene todo un registro de los pasos q vams dando así que aunq en local haya perdido todos los commits y los archivos con la instrucción reflog veoo todo los cambios efectuados en el árbol del tiempo y puedo volver a cualquier punto siempre que quiera



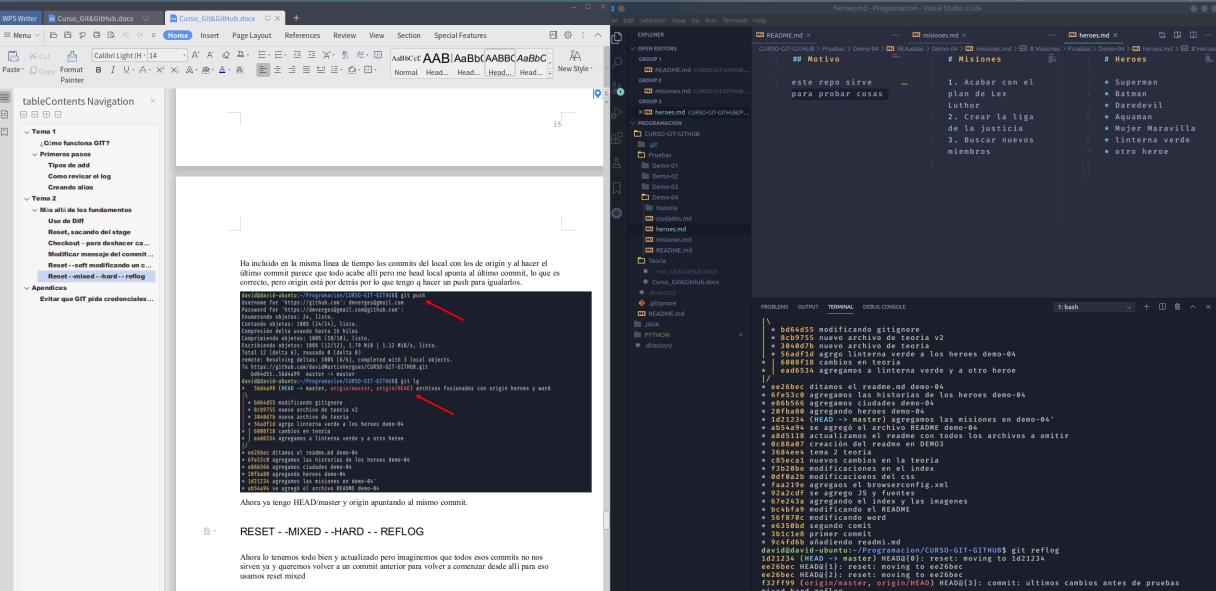
Me indica que mi HEAD está apuntando a ese commit y para abajo todos los cambios que he ido haciendo. Lo que yo quiero es mover a donde apunta origin y revertir todos los cambios hechos desde entonces. Ahora lo tengo así, sin teoría y solo dos archivos en mi proyecto, voy a hacer reset --hard al commit deseado



Hacemos el reset



Y vemos como todo vuelve a como estaba



Recupero los archivos del proyecto y la teoria.

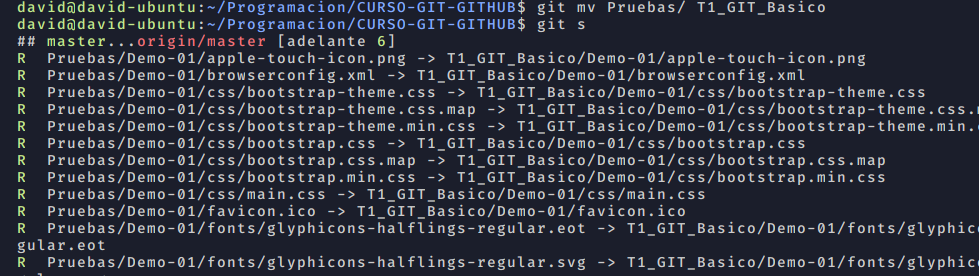
### Renombrar y eliminar archivos con GIT

#### Con terminal

Usamos: mv rm

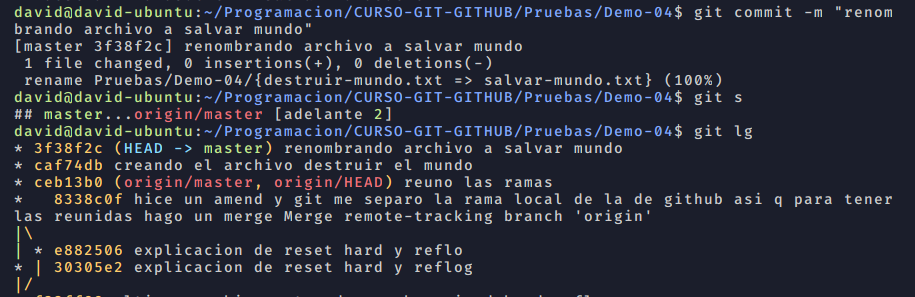
Si usamos la terminal para renombrar conservaremos toda la hstoria de ese archivo, no será considerado un archivo nuevo.

En mi caso tengo un directorio, Pruebas que quiero renombrar a T1\_GIT\_Basico utilizo el comando mv de esta manera (git mv nombre\_archivo\_antiguo nuevo\_nombre):



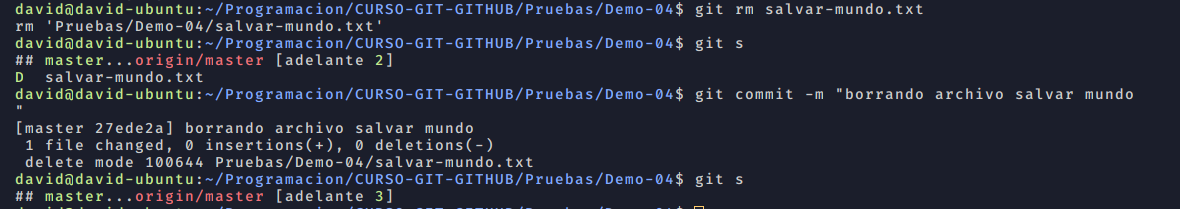
Vemos como todo el contenido del directorio se va renombrando con la nueva ruta

**R** de rename. Ese cambio está en el stage así que para confirmar el cambio hay que hacer un commit.



Vemos como no queda nada en el satge y se ha confirmado el cambio.

Si queremos eliminar el archivo usamos rm y lo mismo se nos quedará e cambio en el stage y luego confirmamos con un commit



**D** de delete.

Si usamos el sistema de archivos para hacer cambios en nuestros proyecto debemos proceder de la siguiente manera para que git entienda los cambios.

1. Hacemos la modificación (cambiar el nombre de un fichero)



Git interpreta que hemos eliminad un arcivo y hemos creado otro nuevo, así que debemos registrar el cambio como un update.

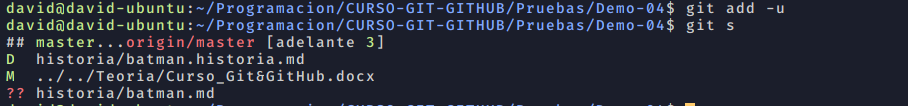
**??** significa que es un archivo sin seguimiento, para remediarlo hay que hacer un add

**D** delete, archivo eliminado

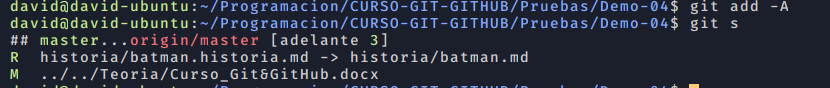
**M** modified, archivo modificado

1. Hacemos un update git add -u

Este comando nos permite añadir al stage solo aquellos archivos modificados/eliminados sin añadir los que no estan en seguimiento.

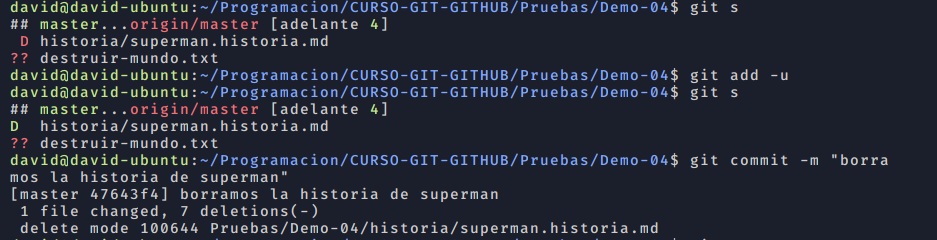


Hemos subido al stage el delete pero no ha añadido el nuevo. Para que lo reconozca como un renombramiento hacemos un -A



Para confirmar el cambio hacemos el commit

Si eliminamos el archivo es lo mismo primero un add -u y después un commit



### Ignorar archivos en git - .gitignore

Este archivo debe ir en la raíz del proyecto.

En cada línea de este archivo debe tener una expresión para gnorar ciertos archivos.



# TEMA 3 - RAMAS CONFLICTOS Y TAGS

## Definición

Una rama no es más que una línea del tiempo alterativa a la línea principal (master). En esa rama podemos hacer nestras modificaciones añadir funcionalidades al proyecto sin alterar el proyecto principal, si posteriormente se aceptan los cambios y todo funciona bien se puede agregar al proyecto principal, rama master, lo que llamamos merge.

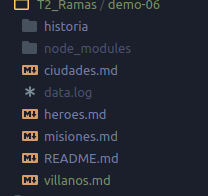
Hay tres tipos de merge:

1. **Fast-forward** => éste se da cuando en la rama master no ha habido ningún cambio desde que se separó la rama secundaria así que simplemente GIT incorpora los cambios a la principal cerrando la secundaria.
2. **Merge automático** => es cuando sí ha habido cambios en la rama principal (master) pero no en archivos comunes entre la rama secundaria y master así que git vuelve a incorporar los cambios sin ningún problema.
3. **Merge manual**=> en este caso git solicita una solución manual ya que que la rama secundaria ha modificado archivos que también se encuentran en la principal. Así una vez resulete el cnflicto se debe realizar un merge commit.

## Fast-forwars ejemplo

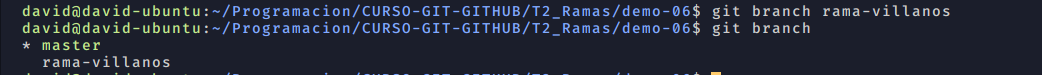
Usamos: branch diff -d

1. En nuestro proyecto Demo-06 creamos un archivo, villanos, pero no estamos seguros si incluirlo en master así que creamos una rama llamada villanos.

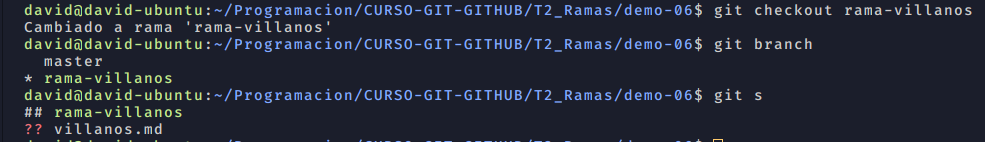




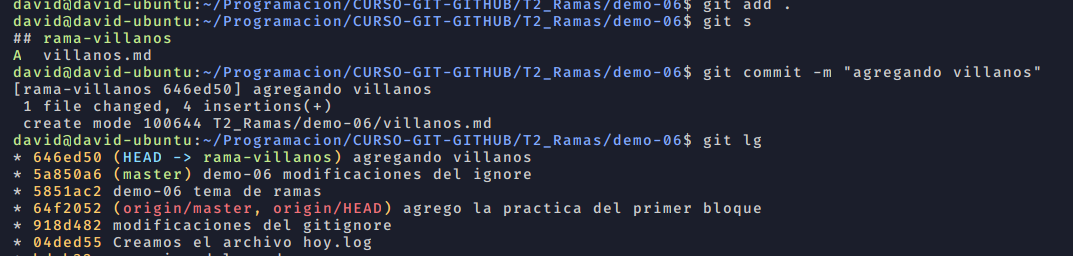
1. Creamos la rama con git banch



1. nos movemos a la rama deseada con un checkout

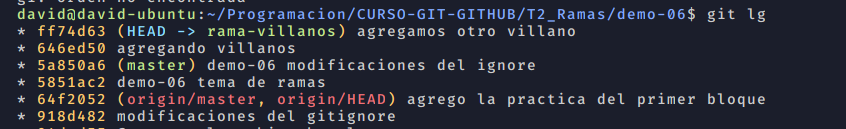


1. Añadimos villanos al stage



Vemos como nuestro HEAD apunta al último commit de la rama donde nos encontramos, master está un commit por debajo.

Editamos de nuevo el fichero y hacemos un nuevo commit en la rama villanos

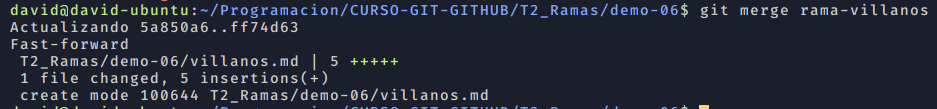


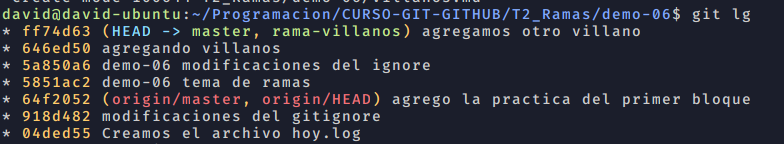
1. Integramos la nueva rama con la master, antes mirareos la diferencia entre las ramas con diff



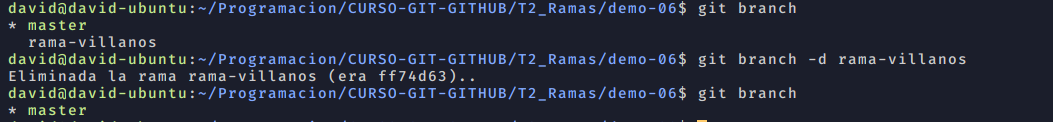
Vemos como indica que en la rama a(villanos) tenemos un archivo villanos que en la rama b (master) no existe.

1. Para fusionar las ramas lo que queremos es fusionar la secundaria con la primaria pero partiendo de master así que volvemos a master y hacemos un merge.





Ahora vemos como master y rama-villanos apuntan al mismo commit. Una vez terminado el trabajo en la rama secundaria es bueno cerrarla para ello git branch -d

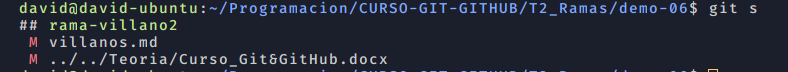


## Merge automático

En esta ocasión creamos una nueva rama y nos movemos automáticamente a ella.

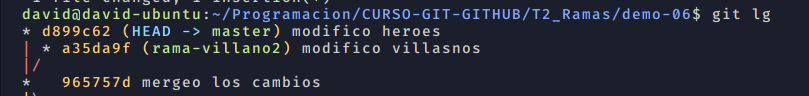


Modificamos el archivo villanos desde la rama nueva.



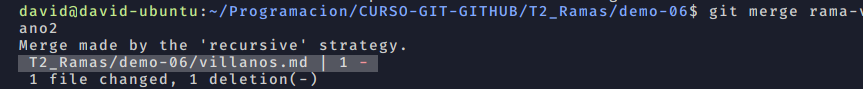
Hago un commit.

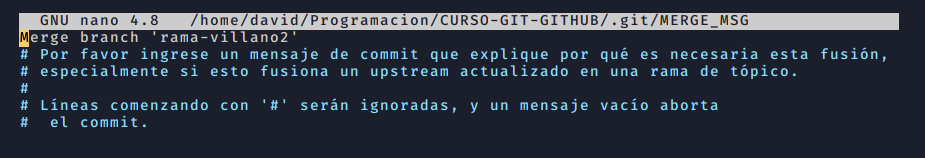
Ahora bien si me muevo a master ese cambio no aparece. Pero una vez en master hacemos una modificación sobre un archivo distinto. Así master modifico archivo heroes y desde rama-vilanos modifico el archivo villanos



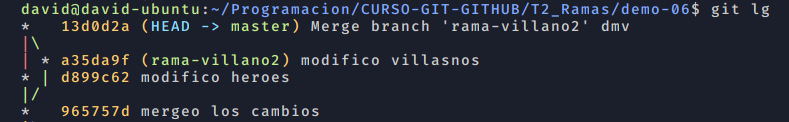
Como ambas ramas hicieron un commit se ven representadas por separado.

Ahora quiero unir las dos ramas estando siempre en master. Como amas ramas han trabajado sore archivos distintos el merge es automático. Me aparecerá para que incluya un texto para el commit.



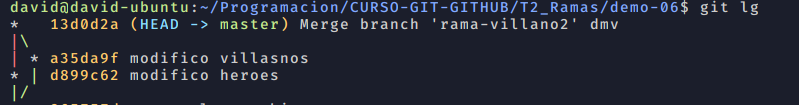


Ahora veo como se han unidos las dos ramas



Los cambios de ambas ramas se reflejan en la rama master.

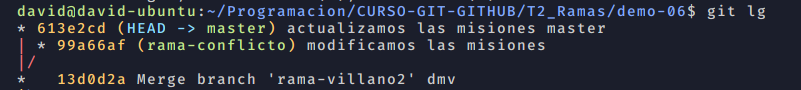
Finalmente borramos la rama secundaria.



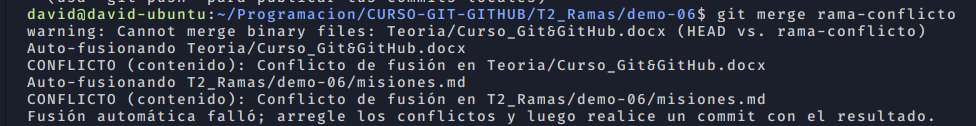
Ya desapareció la rama-villanos2

## Merge Manual o con conflictos

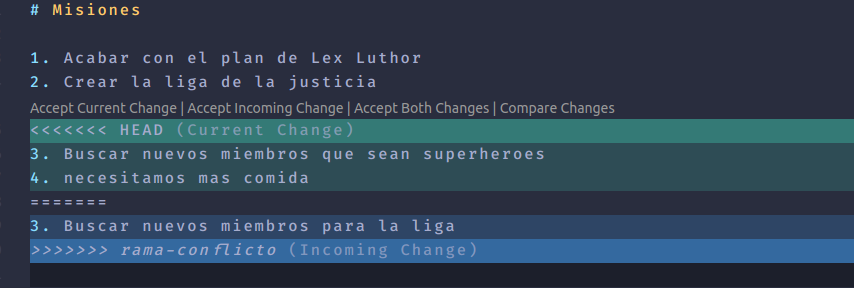
Creamos una rama llamada comflicto. Básicamente lo que haremos es modificar un mismo archivo desde ramas diferentes para crear un conflicto a resolver manualmente.



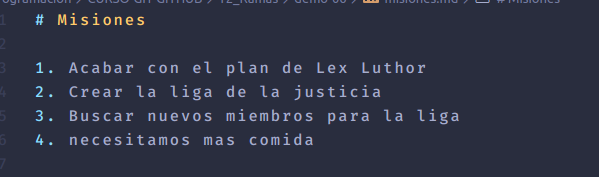
Ahí vemos los cambios realizados en ambas ramas. Cuando haga un merge me saltará el conflicto, tendré q resolverlo y luego hacer un commit. En VSCode me indica los conflictos gráficamente. Pero básiamente lo q no s hace git es escribir en el archivo unas etiquetas para indicar donde están los cambios...dejamos el archivo como queramos y hacemos un commit.

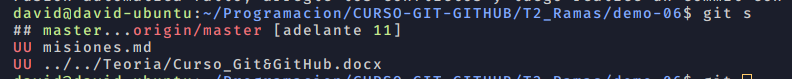


Visión gráfica

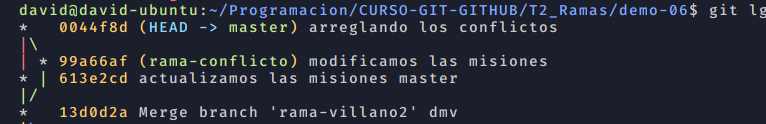


Esos son los dos cambios. Ahora manuelmente puedo arreglar como quiero q quede el archivo (quitando esas etiquetas de head y rama-conflicto) o bien darle a las sugerencias de vscode de aceptar uno u otro. Lo arreglo manualnete y hago el commit





Así es como se ven los archivos modificados pero no mergeados (UU ) y hago el commit.



Ya he fusionado ambas ramas y arreglado los conflictos, ahora borramos la rama secundaria.

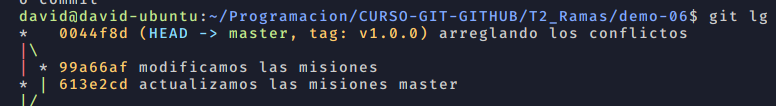
## Tags o etiquetas

Son referencias a un commit en concreto, simplemente les damos un nombre a esos commit pej v 1.0.1. Sirven para marcar versiones o release.

Para crear un tag y añadir un mensaje asociado a ese tag hacemos:



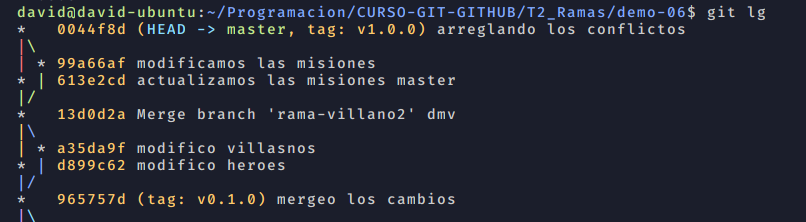
Y en el log ya vemos el tag



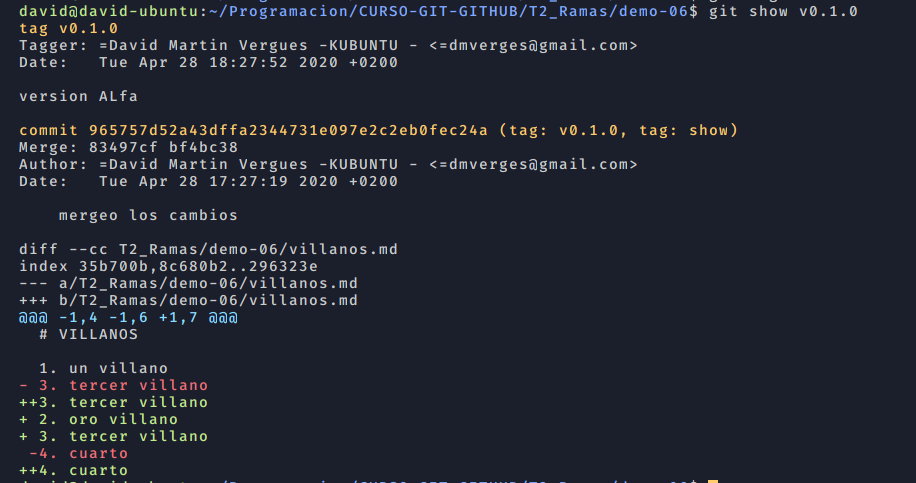


Podemos poner tag en cualquir commit utilizando su ID





Para ver el mensaje del tag usamos show, nos da toda la info relacionada con ese commit.



# TEMA 4 - GIT stash y rebase

## STASH

### Definición

Pongamos la situcación que tenemos un proyecto sobre el qe estamos trabajando en mejoras y añadiendo nuevos archivos pero todos estos cambios todavía no están probados pero hay qel problema que debemos desplegar el proyecto ya! Enntonces lo que hacemos es meter todos esos cambiosno probados en un stash, es como una caja donde poner los cambios y dejar el proyecto como estaba en el último commit. Posteriomente podremos recuperar todos esos archivos y agrgarlo al proyecto. Stash solo guarda archivos que etsán en seguimiento por GIT.

Ejemplo:

Modifico el archivo misiones

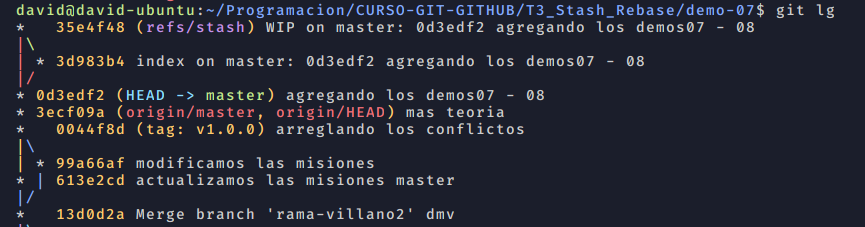


Pero tengo que desplegar master sin añadir ese cambio, pues uso el stash



WIP => work in progress

Ahora ya no tenemos el archivo pendiente de añadir al stage y la modificación realizada desaparece. Hacemos log

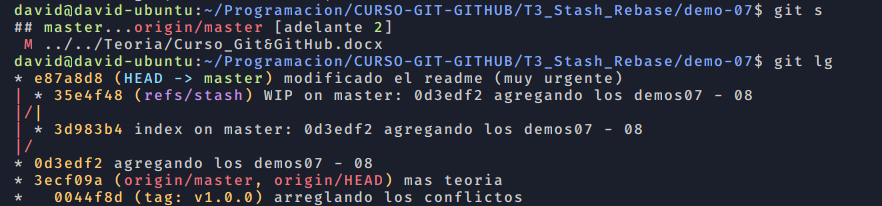


Para ver todos los WIP, trabajos en progreso hacemos list

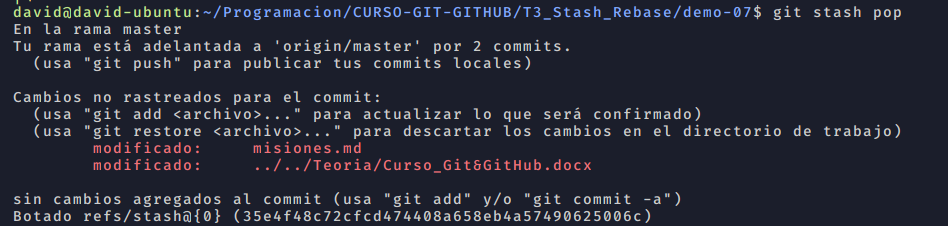


Estamos en stash 0, podemos tener varios, lo bueno es ir eliminando stash a medida q vamos recuperando el trabajo.

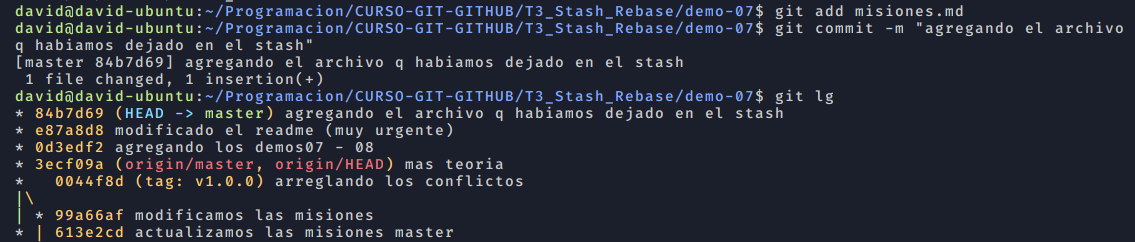
Una vez hecho esto podemos seguir en master y podemos agregar algo q era muy necesario, p ej modificar el archivo README.md y hacemos un commit.



Una vez resuelto esto ya podemos volver a traer los cambios que teníamos en el stash para ello hacemos pop.

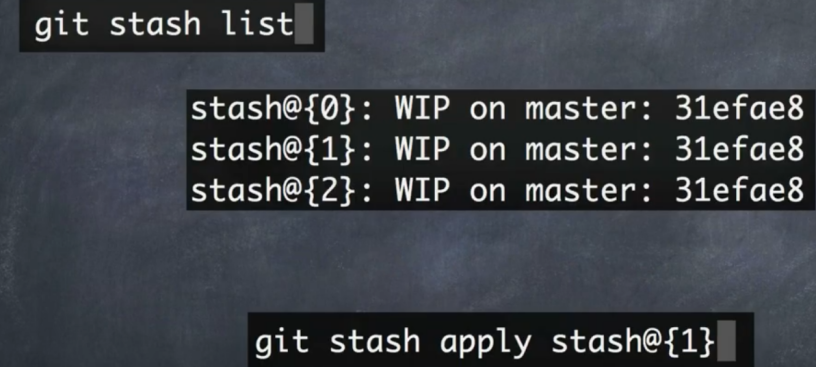


Así cogemos del stash todos los cambios y elimina de la lista de stashes. Sigue apareciendo el archivo como lo teníamos en ese momento y solo tenemos q hacer el commit. Incluso desaparece la referencia al stash en el log.



Cuando hacemos el pop puede que nuestros archivos entren en conflicto, entonces este se resolverá como de un merge se tratase, resolvemos el conflicto y hacemos un add y commit para confirmar los cambios.

Si tuvieramos más de un stash podemos aplicar/ recièrar el que nos interesa mediante git stash apply stash@{id del stash}



Para borarr las entradas del stash list utilizamos git stah drop, si lo dejamos así boraría el primero si nos interesa borrar uno en concreto seguimos con la posición deseada.

## Rebase

### Definición

Imaginemos que de master sacamos una rama y vamos trabajando en ella, hemos hecho unos cuantos commits, pero resulta que en la rama master también hemos ido avanzando y tiene commits, para poder actualizar nuestra rama e incluir los cambios hechos en master utilizamos el rebase. El rebase se tiene que hacer desde la rama secundaria y hacer git rebase master.

Git rebase master

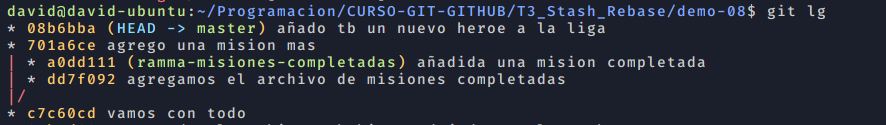
Hay otro tipo de rebase, el llamado interactivo Git rebase -i HEAD~3esto lo que hace es recuperar, en este caso, los últimos 3 commits hechos antes del HEAD pero en lugar de HEAD podemos poner el hash de un commit. Al ejecuta esto lo que hace es guardar estos commit en una érea temporal para luego regresarlos a master en el mismo orden en el q estaban. Casos en los que utilizar el rebase interactivo son :

1. ordenar commits
2. Corregir mensajes de los commits
3. Unir commits
4. Separar commits.

### Rebase para actualizar una rama secundaria

Usamos: git rebase nombreRama

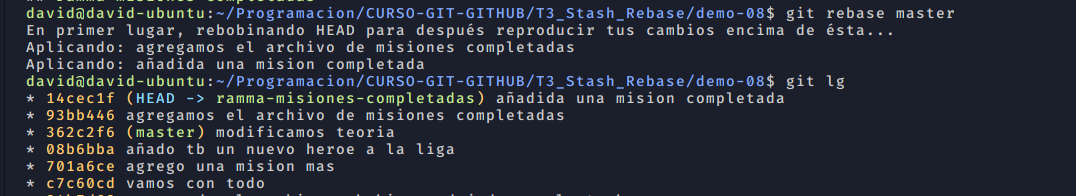
Tenemos esta situación una rama master con dos commits y en paralelo una rama secundaria tb con dos commits, ahora quiero q la rama secundaria tb incluya esos commits de master, es decir actualizar mi rama secundaria.



Este movimiento es como si avanzáramos el incicio de la rama secundaria y esta ponerla al final del último commit de la master así cuando hagamos un merge de ambas nos hará un merge fast-forward y no nos dará conflictos.

1. Vamos a la rama secundaria
2. Hacemos el rebase con master
3. Volvemos a la master
4. Hacemos merge de la 2º a master
5. Borrar la rama secundaria

Aquí vemos como los commits de master ahora están por detras y los commits de la secundaria quedan delante de la línea temporal, ahora no nos diferencia estre ramas pero si nos fijamos tenemos master en un commit y dos por delante tenemos la rama secundaria.



Mergeamos:



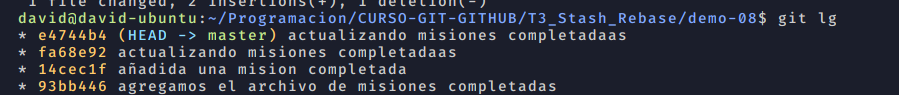
Este merge es automático sin comflictos de ramas xq ya hemos actualizado anteriormente la rama secundaria con los cambios q había en master.

### Uniendo dos commits

Usamos: git rebase -i commit~x

La i significa interactive, rebase interactivo

Pongamos que hacemos dos commits consecutivos pero q los cambios son sobre el mismo archivo y deberían ir en un mismo commit, como se muestra en la foto

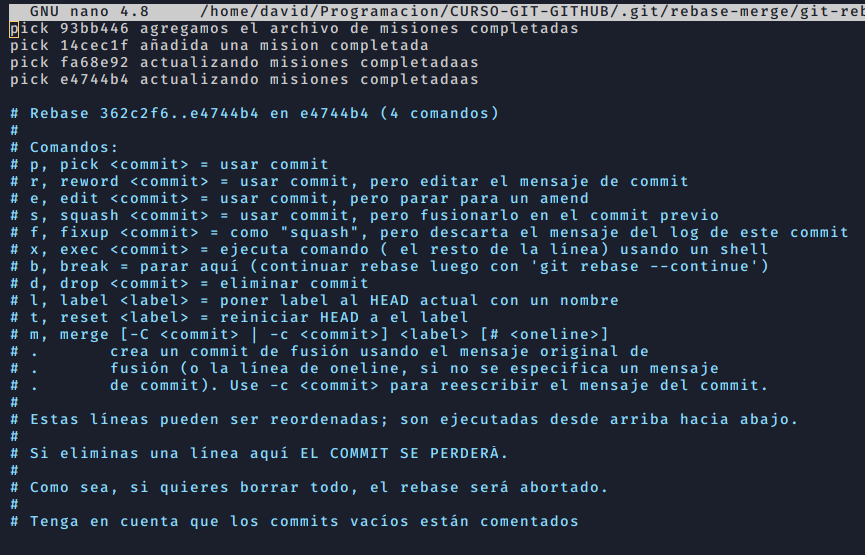


Hacemos un rebase -i

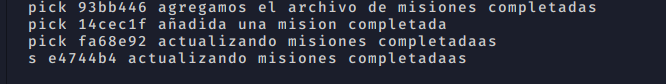


Para coger los últimos 4 commits partiendo de HEAD (este incluido).

Y nos salta esto



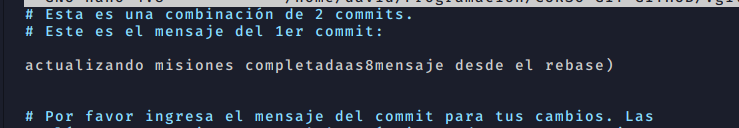
En rojo son los commits que hemos seleccionado con el rebase (el primero es el más antiguo). En azul aparecen las funciones que podemos hacer, la que nos interesa es el squash. Así que ponemos s delante del commit q queremos fusionar y este lo unirá con justo el anterior, de los 4 solo queremos fusionar los dos últimos



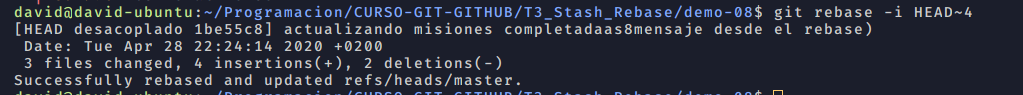
Salimos y nos aparecerá esta otra página para ponerle un mensaje al nuevo commit:

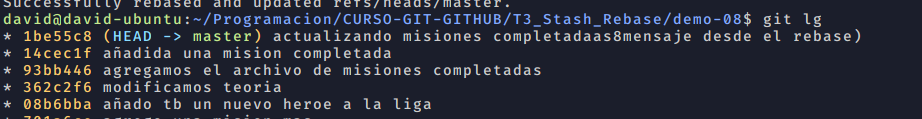


Cambiamos el mensaje del commit.



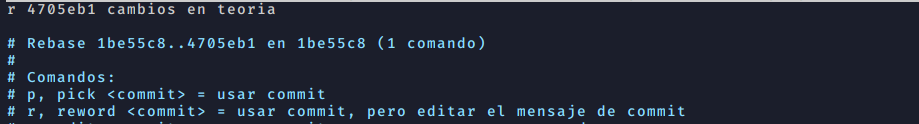
Guardamos y salimos y nos dará el commit:



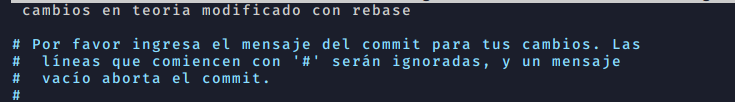


### Modificar el mensaje de un commit con rebase

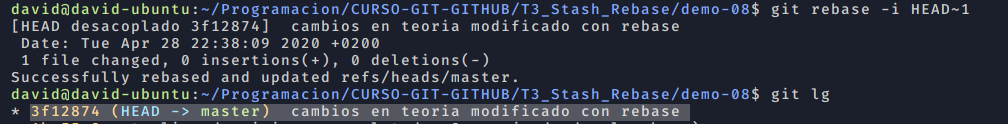
1. Git rebase -i HEAD~1 (cogemos el último commit) y lo marcamos con r



1. Modificamos el mensaje



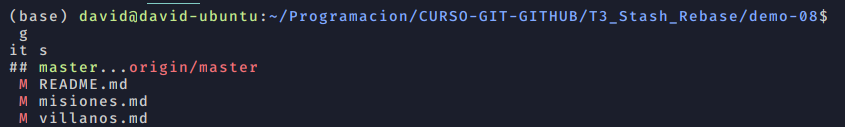
1. Guardamos y salimos



1. Como me pone q HEAD está desacoplado es bueno hacer un checkout a master para asegurarnos q no crea una rama nueva

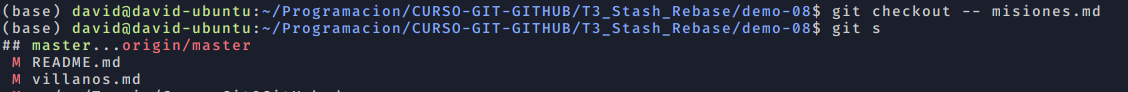
### Rebase edit

Imaginemos que modifico 3 archivos



Ups el readme la fastidié y tengo q revertir los cambios, pues hago un:

git checkout -- misiones.md

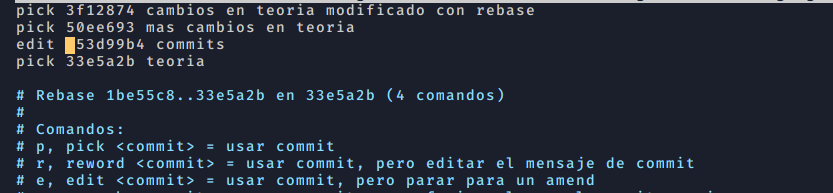


Los cambios en ese archivo se revierten y ya solo me aparece como pendientes 2 archivos.

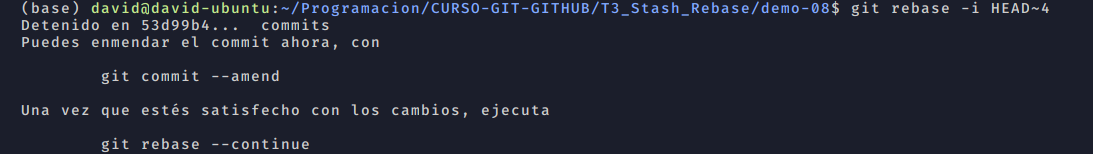
Si ahora hacemos un commit de ambos archivos pero luego me doy cuenta que debería haber hecho un commit para cada archivo puedo usar rebase. Entonces aplicamos un rebase interactivo



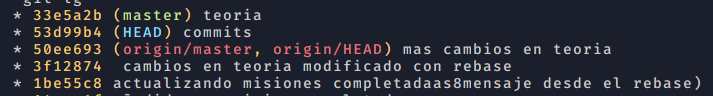
Selecciono los últimos 4 commits y de esos edito el tercero, la intención es separarlo en 2 commits.



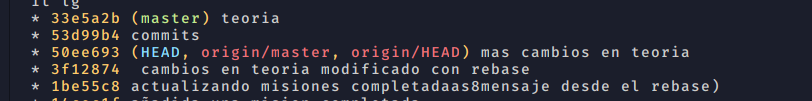
Guardo cambios salgo...y obtengo esta indicación:



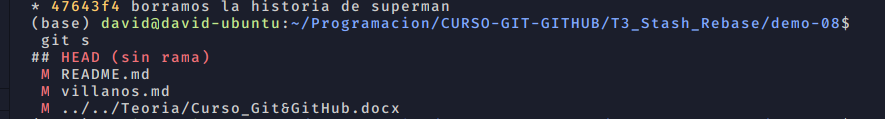
Como yo no quiero cambiar el mensaje del commit paso de la recomendación y llevo el HEAD al commit justo anterior, revierte los cambios del último commit sin destruirlos.



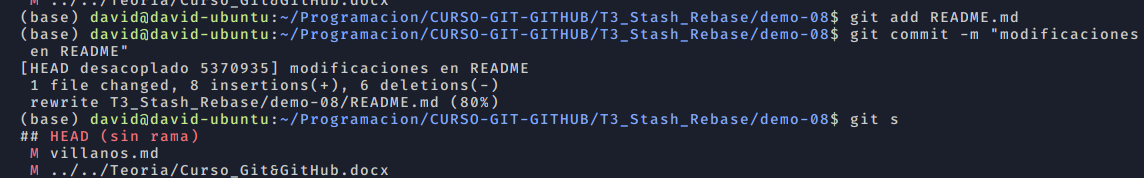
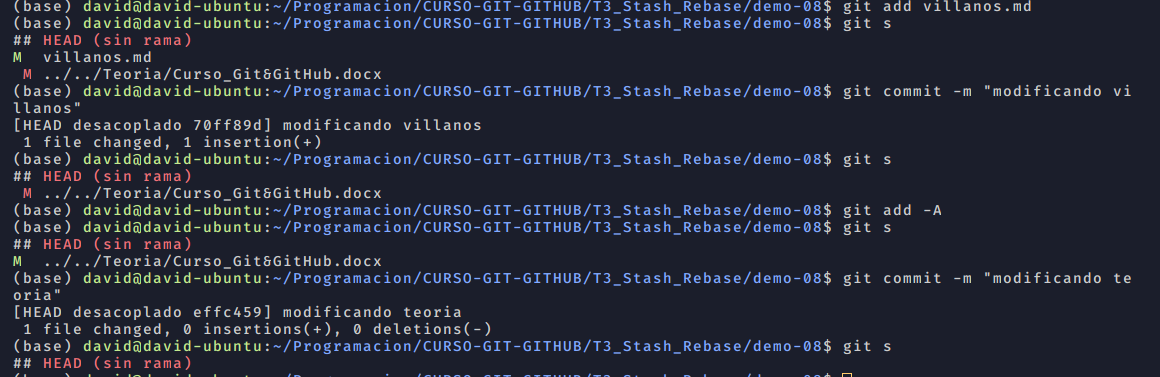
Esto como estaba antes del reset y después veremos el HEAD apuntando dnd esta origin.



Pero sin revertir los cambios, solos los quita del stage. Los archivos siguen teniendo las modificaciones.



Ahora ya podemos poner cada archivo en un commit diferente.

Durante todo este proceso el HEAD no tenía rama pero eso es xq estamos en medio de un rebase, para volver el HEAD a una rama hacemos un rebase --continue.

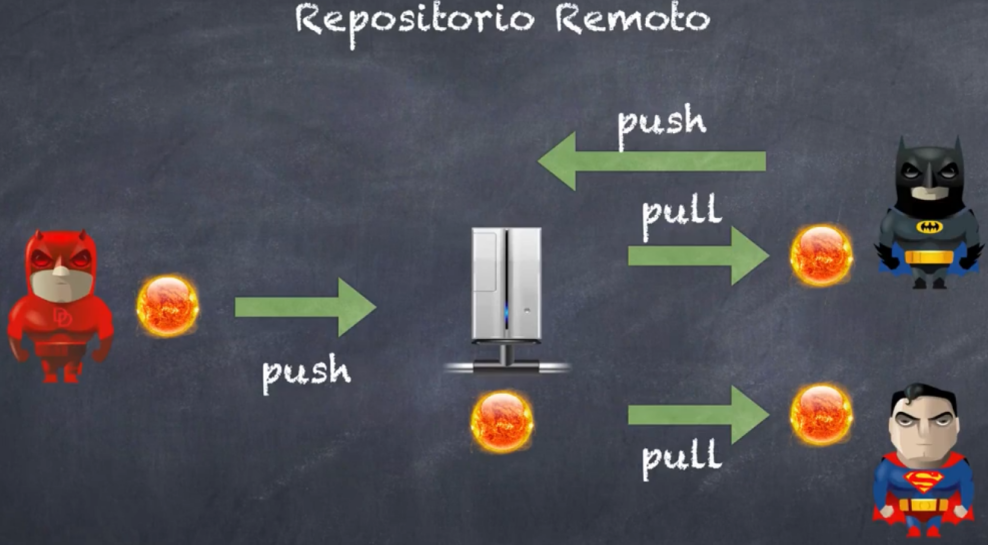


# Tema4-github gitremote push& pull

## GitHub

GitHub nos permite tener nuestro repositorio en la nube para evitar perder el trabajo si nuestro computadora se dañara.

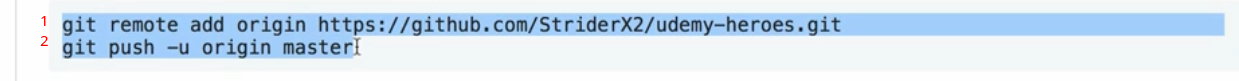
Básicamente cuando trabajamos con repositorios remostos, por ejemplo en GitHub, tendremos que usar comandos como **push**, para subir los cambios o **pull**  para bajar los cambios hechos por otra persona a mi proyecto.



Ahora bien git no maneja el acceso al repositorio, es decir todos los integrantes del equipo podrían acceder a todos los archivos del repo. Para controlar el acceso a un repo remoto podemos usar servicios de hosted como **GitHub** o **BitBucket** o utilizar una herramienta llamada **Gitosis.**

### Flujo de trabajo con GitHub

1. Creamos un repositorio (vacío) en GitHub con el nombre que queramos.
2. Hacemos push desde nuestro repositorio local. Para ello basta con:



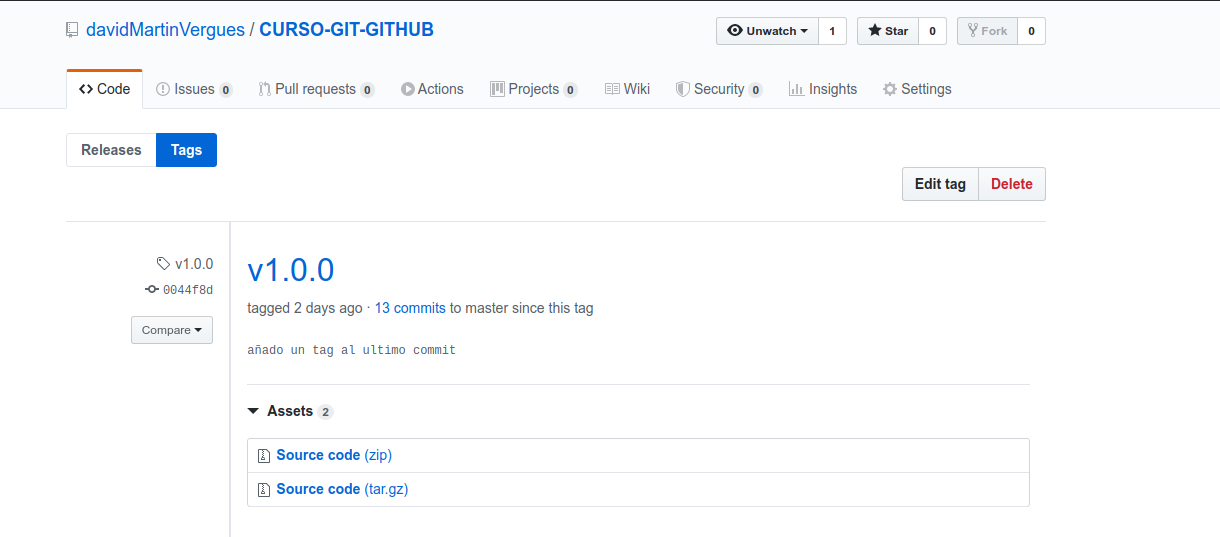
1.- añadimos un repositorio remoto con nombre origin y en la dirección especificada.

2.-fijamos que al hacer push lo haga de la rama master por defecto git push -u origin master

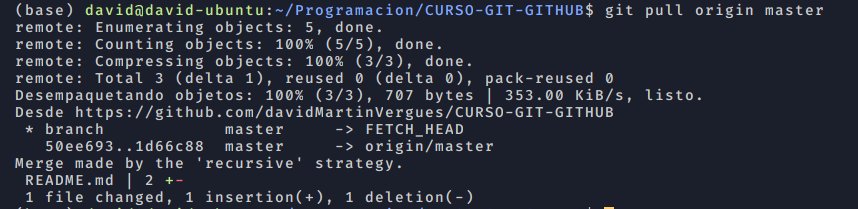
1. Comprobamos que establecimos correctamente el repositorio remoto con git remote -v esto nos va a dar dos informaciones como se llama el repo remoto**1**, de donde coge los datos(o hace el pull)**2**, donde envía los datos**3**. Por lo general ambas direcciones, fetch y push, serán iguales.



1. Cuando hacemos push por defecto no sube los tags, para que lo haga debemos hacer un push específico para los tags git push --tags. Los tags me sirven pra marcar algo importante durante el desarrollo y desde gitHub tb podré bajar el repo tal cual estaba durante ese tag. Los tags en github se encuentran en la pestaña release.



1. Imaginemos que un compañero hace modificaciones en alguno de los archivos, si queremos descargarnos esos cambios necesitamos hacer un git pull.



# Apendices

## Evitar que GIT pida credenciales en cada push a GitHub

Un repositorio git puede ser accedido utilizando distintos protocolos, de entre los cuales los más comunes son SSH y HTTPS.

HTTPS tiene algunas ventajas sobre SSH: Es más fácil de utilizar, porque no necesita la generación de un par de claves pública/privada, y normalmente no tiene problemas en atravesar firewalls y proxies. El inconveniente es que es necesario introducir el usuario y contraseña cada vez que se accede al repositorio remoto con una operación pull o push.

Pero también es posible configurar git para que almacene estas credenciales y las utilice cuando sea necesario, sin solicitarlas cada vez.

La manera más sencilla de evitar que git solicite usuario y contraseña cada vez, es establecer un asistente de credenciales en la configuración de git:

$ git config --global credential.helper store

Con esto, en el siguiente acceso que se realice al repositorio remoto, el usuario y contraseña introducidos quedan almacenados en un fichero ~/.git-credentials.

Si nos preocupa que la contraseña quede guardada en el disco duro, podemos utilizar un asistente de credenciales «cache». En este caso, la contraseña queda guardada en memoria, durante un tiempo limitado que se puede configurar (por defecto, 15 minutos). Por ejemplo, para utilizar este asistente, con una duración de una hora:

$ git config --global credential.helper 'cache --timeout=86400'

