

22 de abril de 2020

David Martin Vergues

Curso Git y GitHub

UDEMY course

[Tema 1 2](#_Toc38488305)

[¿Cómo funciona GIT? 2](#_Toc38488306)

[Primeros pasos 2](#_Toc38488307)

[Tipos de add 7](#_Toc38488308)

[Como revisar el log 8](#_Toc38488309)

[Creando alias 9](#_Toc38488310)

[Tema 2 10](#_Toc38488311)

[Más allá de los fundamentos 10](#_Toc38488312)

[Uso de Diff 10](#_Toc38488313)

[Reset, sacando del stage 11](#_Toc38488314)

[Checkout – para deshacer cambios 11](#_Toc38488315)

[Modificar mensaje del commit (--amend) 11](#_Toc38488316)

[Reset –soft modificando un commit 12](#_Toc38488317)

# 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

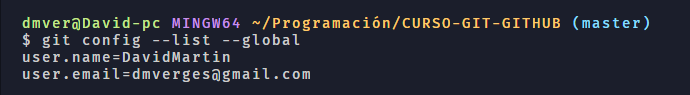
Lo primero que debemos hacer es guardar un nombre de usuario y un mail en nuestra máquina para presentarnos quienes somos en GIT, estas credenciales serán las usadas en todos los repositorios que tengamos en local. Podemos optar por tener un nombre y un mail específicos por repositorio. Todo depende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





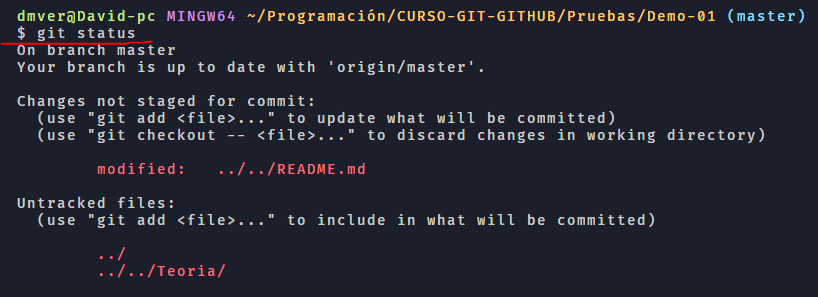
Usamos: init status add commit .gitignore

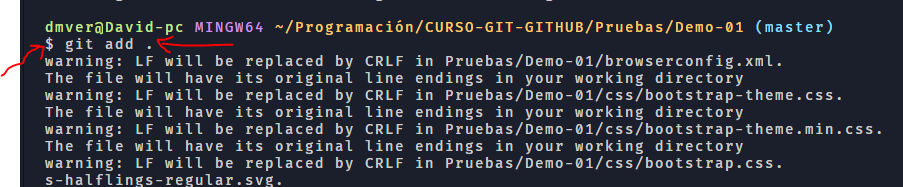
Cuando hacemos un init en nuestro repositorio local se crea la carpeta git donde estará todo lo necesario para guardar nuestros cambios.



Luego creamos nuestros archivos

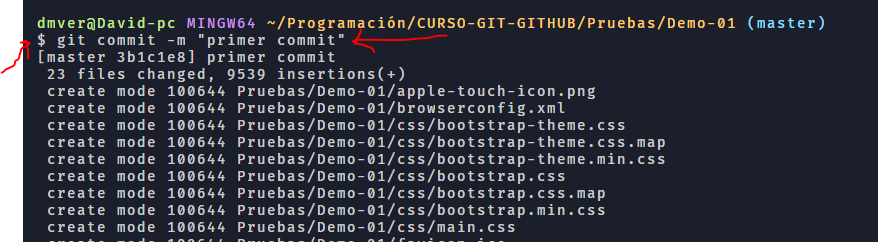
Cuando queremos ver el estado de git usamos git status y si hay cambios que hay que incluir usamos git add.





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.

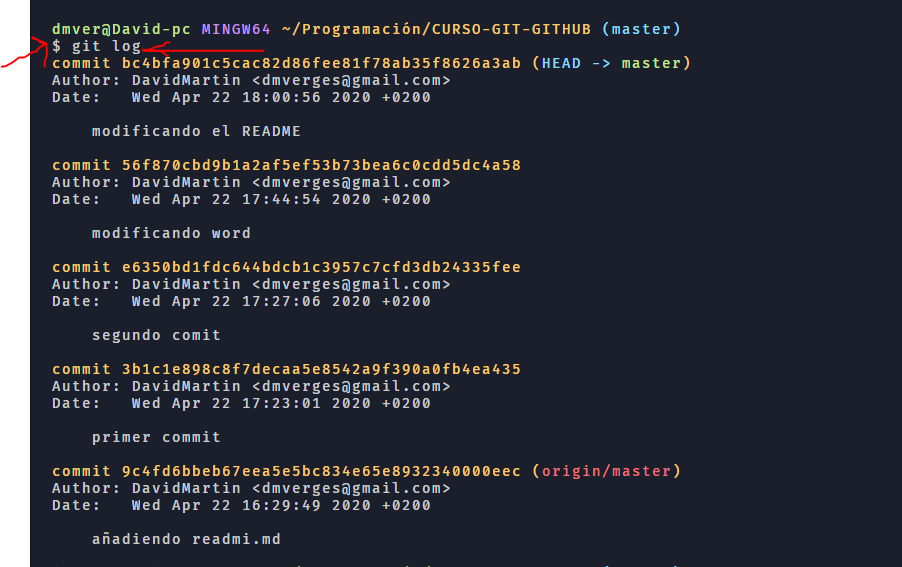


Usamos: checkout log

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

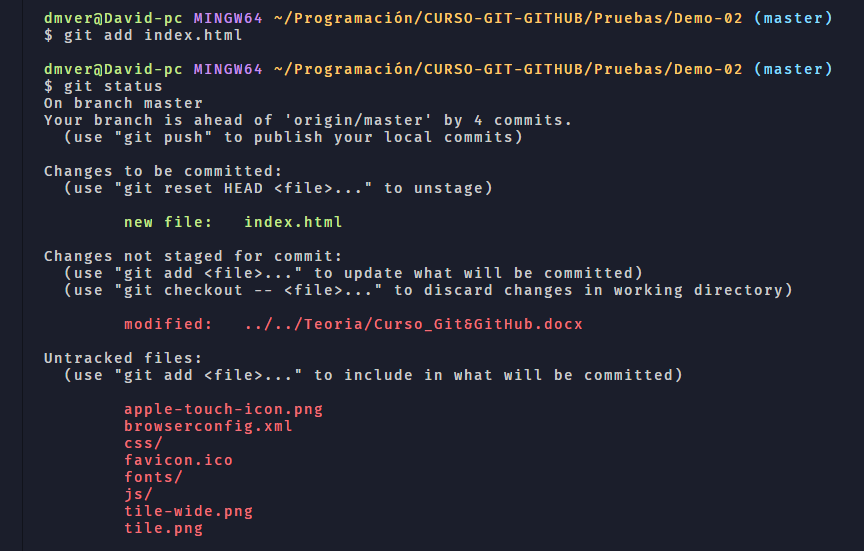


A medida que vamos haciendo commits se va creando un registro con toda la información. Para acceder a este registro usamos log



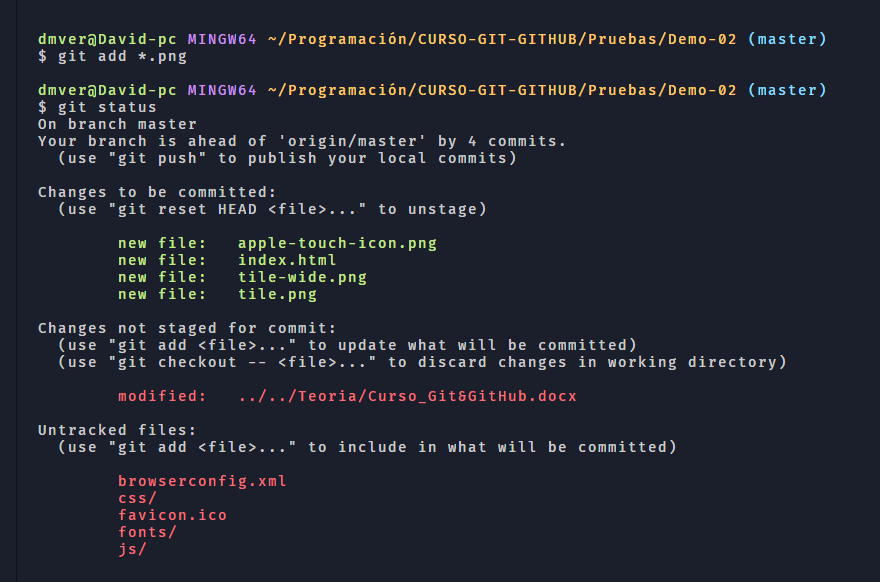
Lo correcto sería hacer commits por archivos, es decir si tenemos varios archivos modificados incluir cada uno de los archivos por separado y hacer el commit. Por jemplo hago el commit solo del index.

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

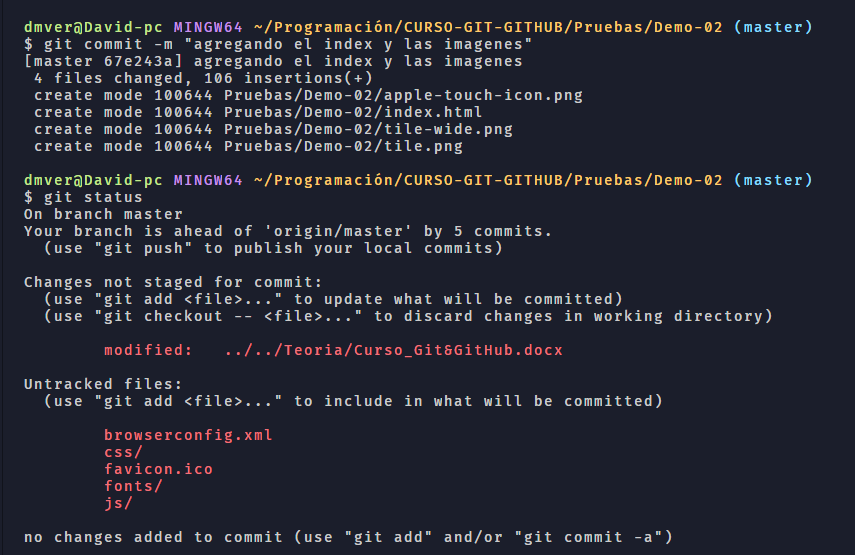


En la imagen vemos como index está en el stage y el resto de archivos en rojo todavía no.

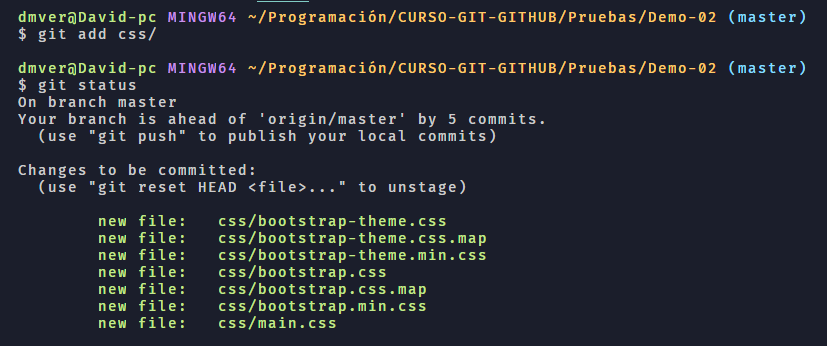
Para añadir todos los archivos de un mismo tipo git add \*.png



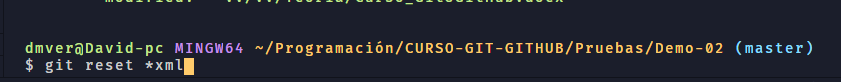
Hacemos el commit indicando que trackeamos el index y las imágenes.

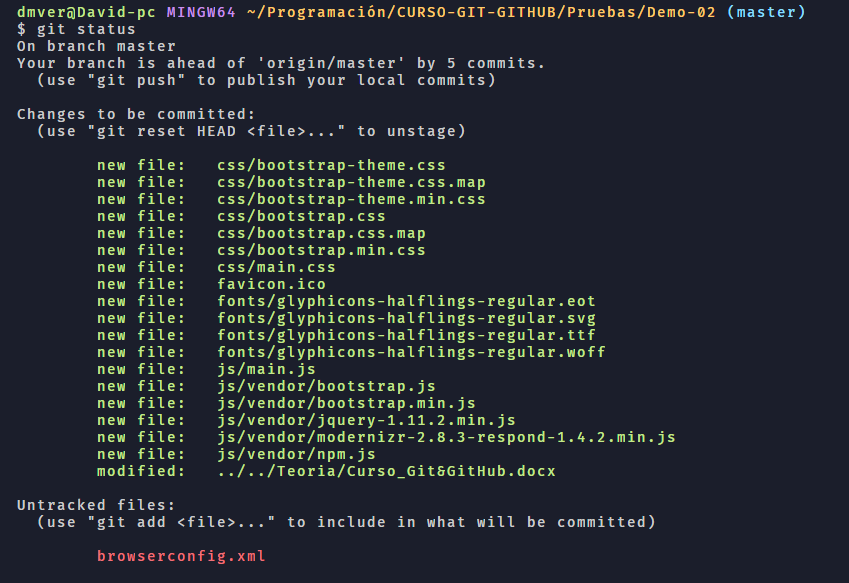


Agregar toda una carpeta el stage



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

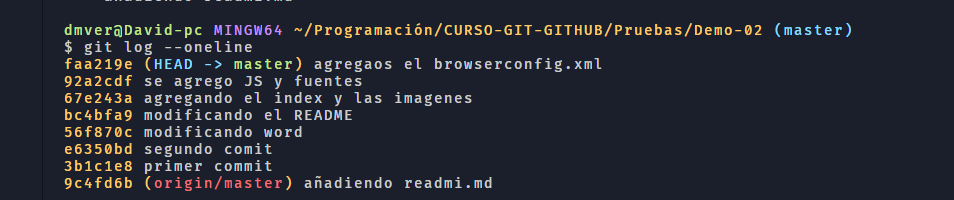
### Como revisar el log

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

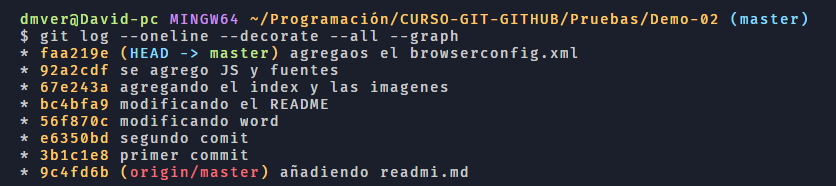
Cuando hacemos un log vemos el HEAD que nos indica el último commit de la rama en la que estamos, en este caso master



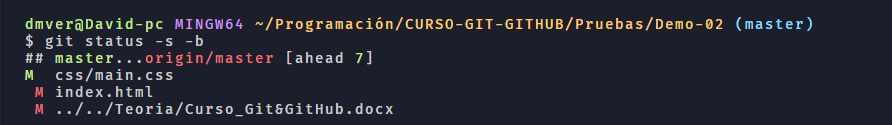
Oneline



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

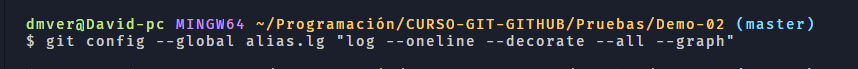


En M rojo modificados pero fuera del stage y M verde modificado pero dentro del stage

### 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



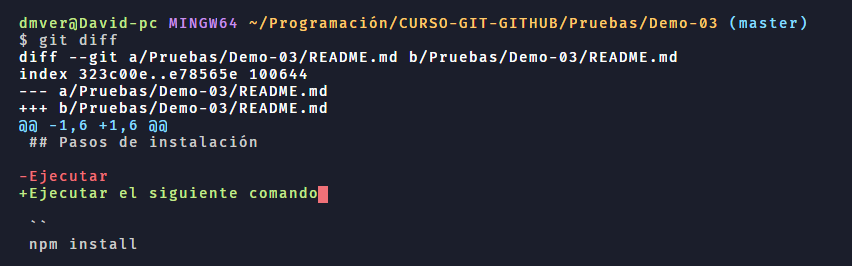
# Tema 2

## Más allá de los fundamentos

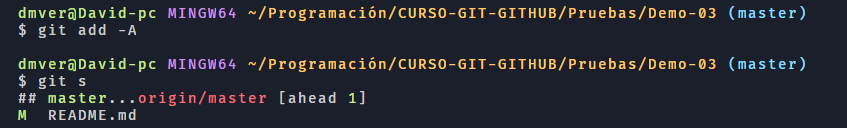
### Uso de Diff

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

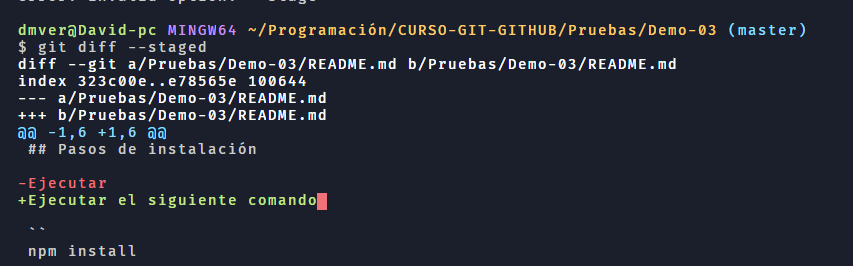
Para saber que modificaciones hicimos en un archivo hacemos un diff, nos da las modificaciones entre el último commit y el momento actual.



Incorporamos el archivo al stage

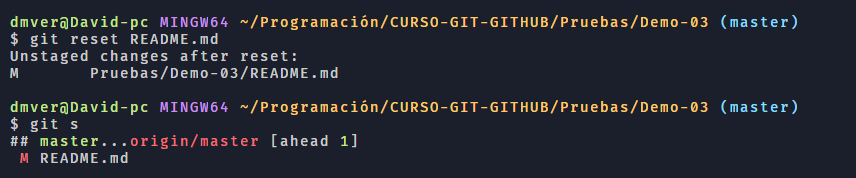


Ahora al ejecutar diff no nos muestra nada para saber las modificaciones de una archivo que ya está en el stage le añadimos –staged



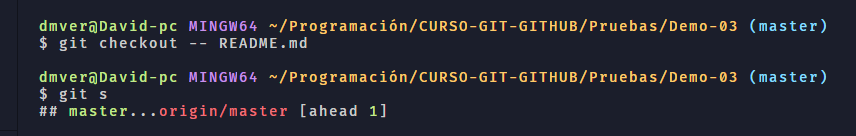
### Reset, sacando del stage

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



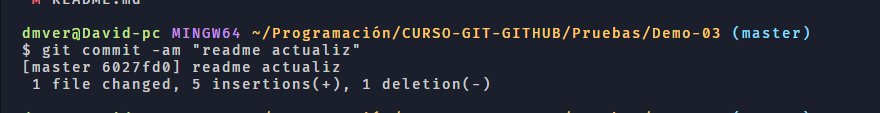
### Checkout – para deshacer cambios

Y lo volvemos a la versión anterior con checkout – nombreArchvo



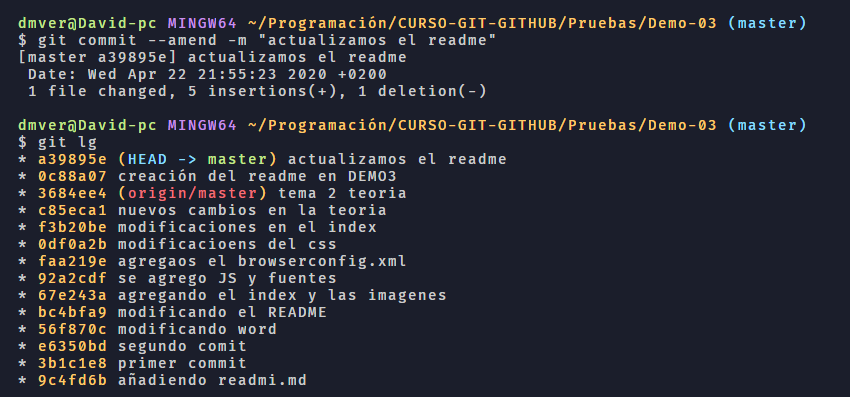
Vemos como al hacer status ya no hay ningún archivo modificado pendiente de poner en el stage.

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.



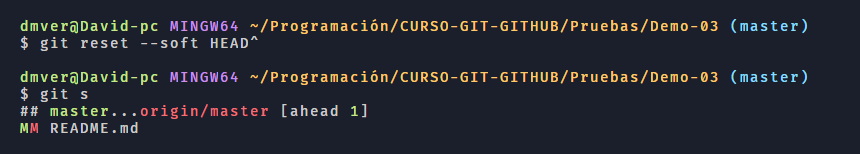
### Modificar mensaje del commit (--amend)

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.

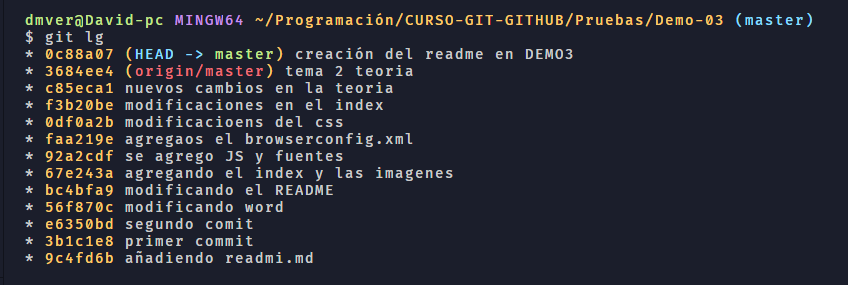


### Reset - -soft modificando un commit

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

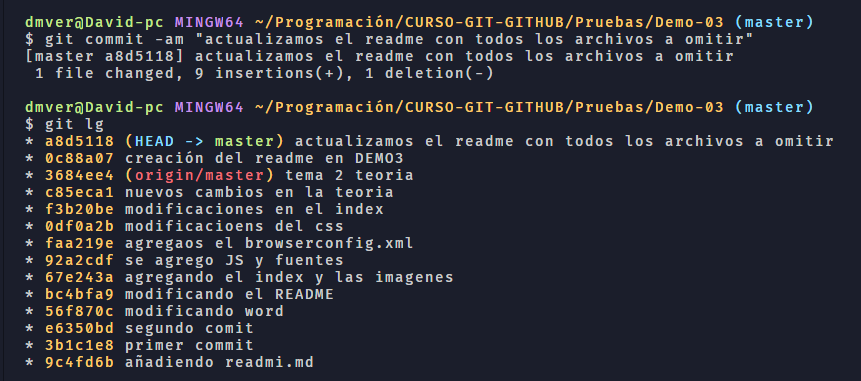
(commit con las modificaciones correctas)

A39895e (commit eliminado)

0c88a07

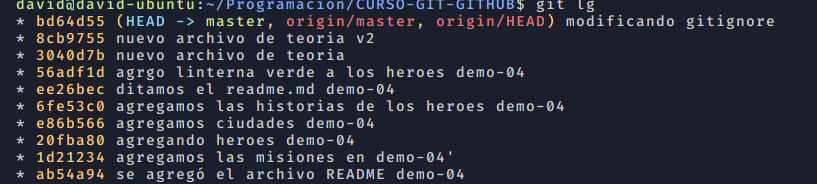
(commit anterior)

Así si hacemos un log queda así



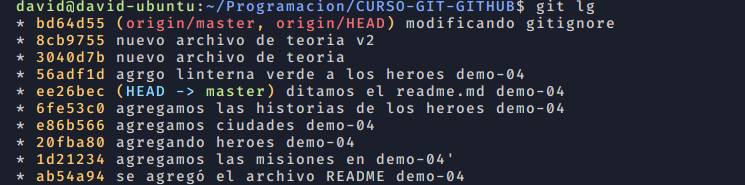
En lugar de usar HEAD^ podemos añadir el id del commit al cual queremos volver

Yo tengo este log:

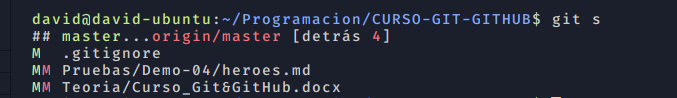
y quiero modificar el commit donde agrego linterna verde entonces voy al commir anterior el ee26bec



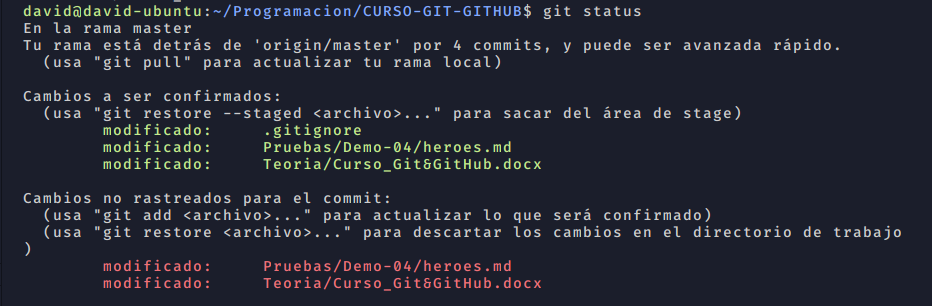
Ahora vemos como el HEAD apunta a este commit



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



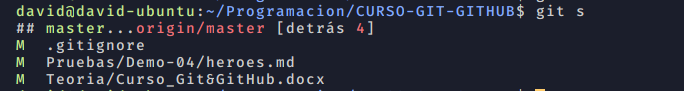
Si miramos la forma extendida se entiende mejor



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

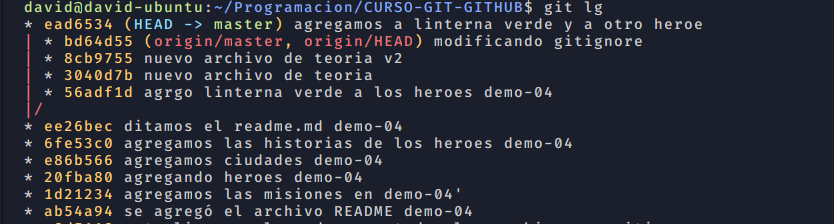




Y hacemos el commit



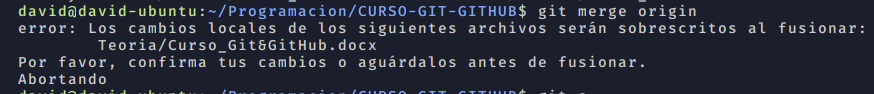
Ahora si hacemos un log vemos el camino



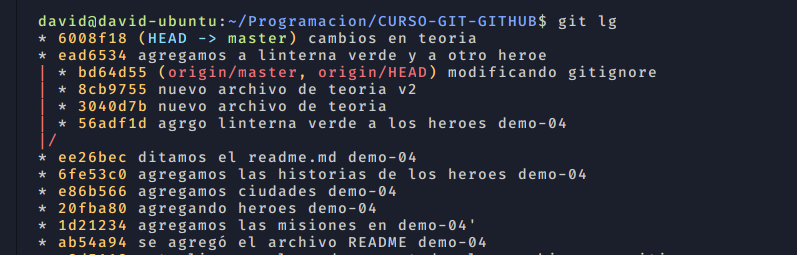
Usamos: merge

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.

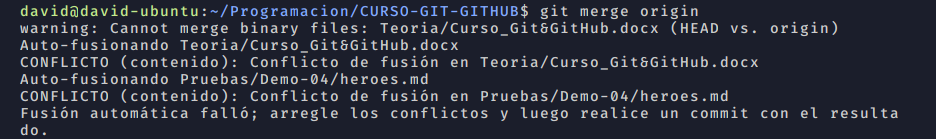
Antes de nada guardo todos los archivos abiertos como por ejemplo el word con la teoría por eso me obliga a hacer añadirlo al stage



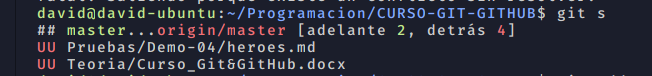
Lo que hago es mirar el status veo que efectivamente teoria esta fuera del stage lo agrego y hago un commit



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



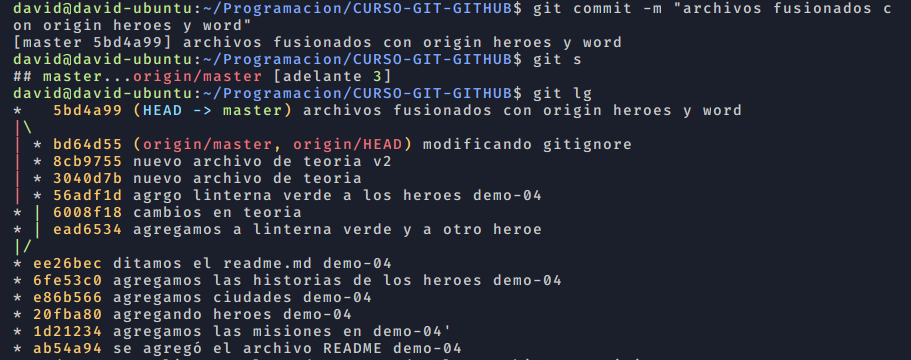
Obviamente cuando intento el merge me dice que hay conflictos por que los archivos modificados en local no concuerdan con los de origin(los de gitHub) estos archivos son el de heroes.md y el word de teoria, si hago un status veo como estan marcados con el flag **UU** que significa updated/unmerged.



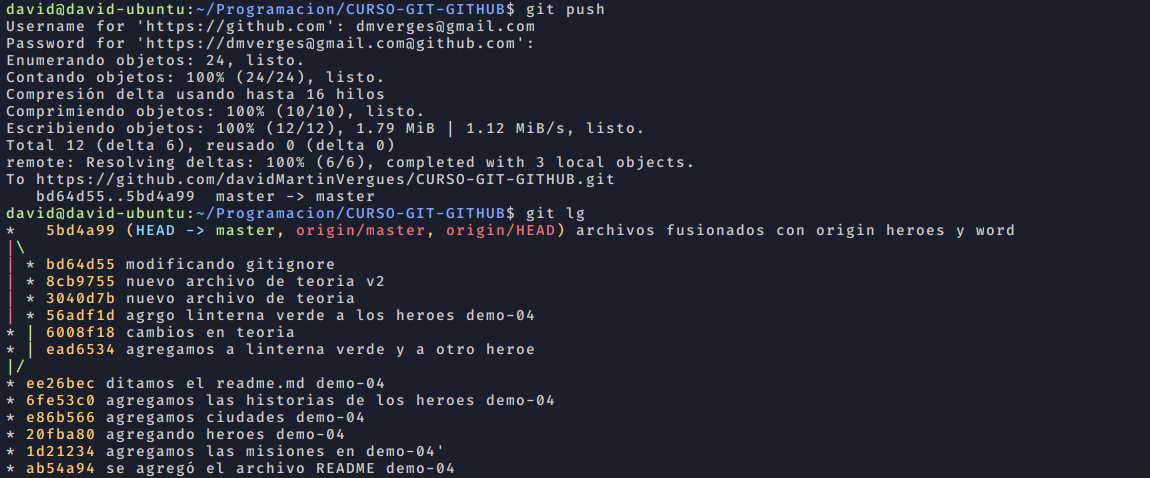
Pero que pasa que el .md el vsCode me muestra las opciones que tengo de manera grafica y me compara ambos archivos así yo puedo elegir con que cambios me quedo así que resulelvo ese comflicto y lo mergea. Así que decido no hacer nada con el word y añado ambos archivos directamente al stage



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

# 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'

