





NOTAS IMPORTANTES.

**Introducción a la terminal y línea de comandos**

**Diferencias** entre la estructura de archivos de Windows, Mac o Linux.

* La ruta principal en Windows es C:\, en UNIX es solo /.
* Windows no hace diferencia entre mayúsculas y minúsculas, pero UNIX sí.

Recuerda que GitBash usa la ruta /c para dirigirse a C:\ (o /d para dirigirse a D:\) en Windows. Por lo tanto, la ruta del usuario con el que estás trabajando es /c/Users/Nombre de tu usuario

**Comandos básicos en la terminal:**

* **pwd**: Nos muestra la ruta de carpetas en la que te encuentras ahora mismo.
* **mkdir**: Nos permite crear carpetas (por ejemplo, mkdir Carpeta-Importante).
* **touch**: Nos permite crear archivos (por ejemplo, touch archivo.txt).
* **rm**: Nos permite borrar un archivo o carpeta (por ejemplo, rm archivo.txt). Mucho cuidado con este comando, puedes borrar todo tu disco duro.
* **cat**: Ver el contenido de un archivo (por ejemplo, cat nombre-archivo.txt).
* **ls**: Nos permite cambiar ver los archivos de la carpeta donde estamos ahora mismo. Podemos usar uno o más argumentos para ver más información sobre estos archivos (los argumentos pueden ser -- + el nombre del argumento o - + una sola letra o shortcut por cada argumento).  
  - ls -a: Mostrar todos los archivos, incluso los ocultos.  
  - ls -l: Ver todos los archivos como una lista.
* **cd**: Nos permite navegar entre carpetas.  
  - cd /: Ir a la ruta principal:  
  - cd o cd ~: Ir a la ruta de tu usuario  
  - cd carpeta/subcarpeta: Navegar a una ruta dentro de la carpeta donde estamos ahora mismo.  
  - cd .. (cd + dos puntos): Regresar una carpeta hacia atrás.  
  - Si quieres referirte al directorio en el que te encuentras ahora mismo puedes usar cd . (cd + un punto).
* **history**: Ver los últimos comandos que ejecutamos y un número especial con el que podemos repetir su ejecución.
* **! + número**: Ejecutar algún comando con el número que nos muestra el comando history (por ejemplo, !72).
* **clear**: Para limpiar la terminal. También podemos usar los atajos de teclado Ctrl + L o Command + L.

Todos estos comandos tienen una función de autocompletado, o sea, puedes escribir la primera parte y presionar la tecla Tab para que la terminal nos muestre todas las posibles carpetas o comandos que podemos ejecutar. Si presionas la tecla Arriba puedes ver el último comando que ejecutamos.

Recuerda que podemos descubrir todos los argumentos de un comando con el argumento --help (por ejemplo, cat --help).

**¿Qué es el staging y los repositorios? Ciclo básico de trabajo en Git**

Para iniciar un repositorio, o sea, activar el sistema de control de versiones de Git en tu proyecto, solo debes ejecutar el comando git init.

Este comando se encargará de dos cosas: primero, crear una carpeta .git, donde se guardará toda la base de datos con cambios atómicos de nuestro proyecto; y segundo, crear un área que conocemos como Staging, que guardará temporalmente nuestros archivos (cuando ejecutemos un comando especial para eso) y nos permitirá, más adelante, guardar estos cambios en el repositorio (también con un comando especial).  
**Ciclo de vida o estados de los archivos en Git**:  
Cuando trabajamos con Git nuestros archivos pueden vivir y moverse entre 4 diferentes estados (cuando trabajamos con repositorios remotos pueden ser más estados, pero lo estudiaremos más adelante):

* **Archivos Tracked**: son los archivos que viven dentro de Git, no tienen cambios pendientes y sus últimas actualizaciones han sido guardadas en el repositorio gracias a los comandos git add y git commit.
* **Archivos Staged**: son archivos en Staging. Viven dentro de Git y hay registro de ellos porque han sido afectados por el comando git add, aunque no sus últimos cambios. Git ya sabe de la existencia de estos últimos cambios, pero todavía no han sido guardados definitivamente en el repositorio porque falta ejecutar el comando git commit.
* **Archivos Unstaged**: entiéndelos como archivos *“Tracked pero Unstaged”*. Son archivos que viven dentro de Git pero no han sido afectados por el comando git add ni mucho menos por git commit. Git tiene un registro de estos archivos, pero está desactualizado, sus últimas versiones solo están guardadas en el disco duro.
* **Archivos Untracked**: son archivos que NO viven dentro de Git, solo en el disco duro. Nunca han sido afectados por git add, así que Git no tiene registros de su existencia.  
  Recuerda que hay un caso muy raro donde los archivos tienen dos estados al mismo tiempo: staged y untracked. Esto pasa cuando guardas los cambios de un archivo en el área de Staging (con el comando git add), pero antes de hacer commit para guardar los cambios en el repositorio haces nuevos cambios que todavía no han sido guardados en el área de Staging (en realidad, todo sigue funcionando igual pero es un poco divertido).  
  **Comandos para mover archivos entre los estados de Git**:
* **git status**: nos permite ver el estado de todos nuestros archivos y carpetas.
* **git add**: nos ayuda a mover archivos del Untracked o Unstaged al estado Staged. Podemos usar git nombre-del-archivo-o-carpeta para añadir archivos y carpetas individuales o git add -A para mover todos los archivos de nuestro proyecto (tanto Untrackeds como unstageds).
* **git reset HEAD**: nos ayuda a sacar archivos del estado Staged para devolverlos a su estado anterior. Si los archivos venían de Unstaged, vuelven allí. Y lo mismo se venían de Untracked.
* **git commit**: nos ayuda a mover archivos de Unstaged a Staged. Esta es una ocasión especial, los archivos han sido guardado o actualizados en el repositorio. Git nos pedirá que dejemos un mensaje para recordar los cambios que hicimos y podemos usar el argumento -m para escribirlo (git commit -m "mensaje").
* **git rm**: este comando necesita alguno de los siguientes argumentos para poder ejecutarse correctamente:  
  - git rm --cached: Mueve los archivos que le indiquemos al estado Untracked.  
  - git rm --force: Elimina los archivos de Git y del disco duro. Git guarda el registro de la existencia de los archivos, por lo que podremos recuperarlos si es necesario (pero debemos usar comandos más avanzados).

# Crea un repositorio de Git y haz tu primer commit

Si quieres ver los archivos ocultos de una carpeta puedes habilitar la opción de *Vista > Mostrar u ocultar > Elementos ocultos* (en Windows) o ejecutar el comando *ls -a*.

Le indicaremos a Git que queremos crear un nuevo repositorio para utilizar su sistema de control de versiones. Solo debemos posicionarnos en la carpeta raíz de nuestro proyecto y ejecutar el comando **git init**.

Recuerda que al ejecutar este comando (y de aquí en adelante) vamos a tener una nueva carpeta oculta llamada .git con toda la base de datos con cambios atómicos en nuestro proyecto.

Recuerda que Git está optimizado para trabajar en equipo, por lo tanto, debemos darle un poco de información sobre nosotros. No debemos hacerlo todas las veces que ejecutamos un comando, basta con ejecutar solo una sola vez los siguientes comandos con tu información:

git config --global user.email "tu@email.com"

git config --global user.name "Tu Nombre"

Existen muchas otras configuraciones de Git que puedes encontrar ejecutando el comando git config --list (o solo git config para ver una explicación más detallada).

# Volver en el tiempo en nuestro repositorio utilizando branches y checkout

El comando **git checkout** + ID del commit nos permite viajar en el tiempo. Podemos volver a cualquier versión anterior de un archivo específico o incluso del proyecto entero. Esta también es la forma de crear ramas y movernos entre ellas.

También hay una forma de hacerlo un poco más “ruda”: usando el comando git reset. En este caso, no solo “volvemos en el tiempo”, sino que borramos los cambios que hicimos después de este commit.

Hay dos formas de usar git reset: con el argumento --hard, borrando toda la información que tengamos en el área de staging (y perdiendo todo para siempre). O, un poco más seguro, con el argumento --soft, que mantiene allí los archivos del área de staging para que podamos aplicar nuestros últimos cambios pero desde un commit anterior.

# Git reset vs. Git rm

Git reset y git rm son comandos con utilidades muy diferentes, pero aún así se confunden muy fácilmente.

## git rm

Este comando nos ayuda a eliminar archivos de Git sin eliminar su historial del sistema de versiones. Esto quiere decir que si necesitamos recuperar el archivo solo debemos “viajar en el tiempo” y recuperar el último commit antes de borrar el archivo en cuestión.

Recuerda que git rm no puede usarse así nomás. Debemos usar uno de los flags para indicarle a Git cómo eliminar los archivos que ya no necesitamos en la última versión del proyecto:

* git rm --cached: Elimina los archivos del área de Staging y del próximo commit pero los mantiene en nuestro disco duro.
* git rm --force: Elimina los archivos de Git y del disco duro. Git siempre guarda todo, por lo que podemos acceder al registro de la existencia de los archivos, de modo que podremos recuperarlos si es necesario (pero debemos usar comandos más avanzados).

## git reset

Este comando nos ayuda a volver en el tiempo. Pero no como git checkout que nos deja ir, mirar, pasear y volver. Con git reset volvemos al pasado sin la posibilidad de volver al futuro. Borramos la historia y la debemos sobreescribir. No hay vuelta atrás.

Este comando es **muy peligroso** y debemos usarlo solo en caso de emergencia. Recuerda que debemos usar alguna de estas dos opciones:

Hay dos formas de usar git reset: con el argumento --hard, borrando toda la información que tengamos en el área de staging (y perdiendo todo para siempre). O, un poco más seguro, con el argumento --soft, que mantiene allí los archivos del área de staging para que podamos aplicar nuestros últimos cambios pero desde un commit anterior.

* git reset --soft: Borramos todo el historial y los registros de Git pero guardamos los cambios que tengamos en Staging, así podemos aplicar las últimas actualizaciones a un nuevo commit.
* git reset --hard: Borra todo. Todo todito, absolutamente todo. Toda la información de los commits y del área de staging se borra del historial.

**¡Pero todavía falta algo!**

* git reset HEAD: Este es el comando para sacar archivos del área de Staging. No para borrarlos ni nada de eso, solo para que los últimos cambios de estos archivos no se envíen al último commit, a menos que cambiemos de opinión y los incluyamos de nuevo en staging con git add, por supuesto.

# ¿Por que esto es importante?

Imagina el siguiente caso:

Hacemos cambios en los archivos de un proyecto para una nueva actualización. Todos los archivos con cambios se mueven al área de staging con el comando git add. Pero te das cuenta de que uno de esos archivos no está listo todavía. Actualizaste el archivo pero ese cambio no debe ir en el próximo commit por ahora.

¿Qué podemos hacer?

Bueno, todos los cambios están en el área de Staging, incluido el archivo con los cambios que no están listos. Esto significa que debemos sacar ese archivo de Staging para poder hacer commit de todos los demás.

¡Al usar git rm lo que haremos será eliminar este archivo completamente de git! Todavía tendremos el historial de cambios de este archivo, con la eliminación del archivo como su última actualización. Recuerda que en este caso no buscábamos eliminar un archivo, solo dejarlo como estaba y actualizarlo después, no en este commit.

En cambio, si usamos git reset HEAD, lo único que haremos será mover estos cambios de Staging a Unstaged. Seguiremos teniendo los últimos cambios del archivo, el repositorio mantendrá el archivo (no con sus últimos cambios pero sí con los últimos en los que hicimos commit) y no habremos perdido nada.

**Conclusión**: Lo mejor que puedes hacer para salvar tu puesto y evitar un incendio en tu trabajo es conocer muy bien la diferencia y los riesgos de todos los comandos de Git.

**Flujo de trabajo básico con un repositorio remoto**

**No veas esta clase** a menos que hayas practicado todos los comandos de las clases anteriores.

Por ahora, nuestro proyecto vive únicamente en nuestra computadora. Esto significa que no hay forma de que otros miembros del equipo trabajen en él.

Para solucionar esto están los **servidores remotos**: un nuevo estado que deben seguir nuestros archivos para conectarse y trabajar con equipos de cualquier parte del mundo.

Estos servidores remotos pueden estar alojados en GitHub, GitLab, BitBucket, entre otros. Lo que van a hacer es guardar el mismo repositorio que tienes en tu computadora y darnos una URL con la que todos podremos acceder a los archivos del proyecto para descargarlos, hacer cambios y volverlos a enviar al servidor remoto para que otras personas vean los cambios, comparen sus versiones y creen nuevas propuestas para el proyecto.

Esto significa que debes aprender algunos nuevos comandos:

* **git clone url\_del\_servidor\_remoto**: Nos permite descargar los archivos de la última versión de la rama principal y todo el historial de cambios en la carpeta .git.
* **git push**: Luego de hacer git add y git commit debemos ejecutar este comando para mandar los cambios al servidor remoto.
* **git fetch**: Lo usamos para traer actualizaciones del servidor remoto y guardarlas en nuestro repositorio local (en caso de que hayan, por supuesto).
* **git merge**: También usamos el comando git fetch con servidores remotos. Lo necesitamos para combinar los últimos cambios del servidor remoto y nuestro directorio de trabajo.
* **git pull**: Básicamente, git fetch y git merge al mismo tiempo.

# Introducción a las ramas o branches de Git

Las ramas son la forma de hacer cambios en nuestro proyecto sin afectar el flujo de trabajo de la rama principal. Esto porque queremos trabajar una parte muy específica de la aplicación o simplemente experimentar.

La cabecera o HEAD representan la rama y el commit de esa rama donde estamos trabajando. Por defecto, esta cabecera aparecerá en el último commit de nuestra rama principal. Pero podemos cambiarlo al crear una rama (git branch rama, git checkout -b rama) o movernos en el tiempo a cualquier otro commit de cualquier otra rama con los comandos (git reset id-commit, git checkout rama-o-id-commit).

# Uso de GitHub

GitHub es una plataforma que nos permite guardar repositorios de Git que podemos usar como servidores remotos y ejecutar algunos comandos de forma visual e interactiva (sin necesidad de la consola de comandos).

Luego de crear nuestra cuenta, podemos crear o importar repositorios, crear organizaciones y proyectos de trabajo, descubrir repositorios de otras personas, contribuir a esos proyectos, dar estrellas y muchas otras cosas.

El README.md es el archivo que veremos por defecto al entrar a un repositorio. Es una muy buena práctica configurarlo para describir el proyecto, los requerimientos y las instrucciones que debemos seguir para contribuir correctamente.

Para clonar un repositorio desde GitHub (o cualquier otro servidor remoto) debemos copiar la URL (por ahora, usando HTTPS) y ejecutar el comando git clone + la URL que acabamos de copiar. Esto descargara la versión de nuestro proyecto que se encuentra en GitHub.

Sin embargo, esto solo funciona para las personas que quieren empezar a contribuir en el proyecto. Si queremos conectar el repositorio de GitHub con nuestro repositorio local, el que creamos con git init, debemos ejecutar las siguientes instrucciones:

# Primero: Guardar la URL del repositorio de GitHub

# con el nombre de origin

git remote add origin URL

# Segundo: Verificar que la URL se haya guardado

# correctamente:

git remote

git remote -v

# Tercero: Traer la versión del repositorio remoto y

# hacer merge para crear un commit con los archivos

# de ambas partes. Podemos usar git fetch y git merge

# o solo el git pull con el flag --allow-unrelated-histories:

git pull origin master --allow-unrelated-histories

# Por último, ahora sí podemos hacer git push para guardar

# los cambios de nuestro repositorio local en GitHub:

git push origin master

**Cómo funcionan las llaves públicas y privadas**

Las llaves públicas y privadas nos ayudan a cifrar y descifrar nuestros archivos de forma que los podamos compartir archivos sin correr el riesgo de que sean interceptados por personas con malas intenciones.

La forma de hacerlo es la siguiente:

1. Ambas personas deben crear su llave pública y privada.
2. Ambas personas pueden compartir su llave pública a las otras partes (recuerda que esta llave es pública, no hay problema si la “interceptan”).
3. La persona que quiere compartir un mensaje puede usar la llave pública de la otra persona para cifrar los archivos y asegurarse que solo puedan ser descifrados con la llave privada de la persona con la que queremos compartir el mensaje.
4. El mensaje está cifrado y puede ser enviado a la otra persona sin problemas en caso de que los archivos sean interceptados.
5. La persona a la que enviamos el mensaje cifrado puede usar su llave privada para descifrar el mensaje y ver los archivos.

Puedes compartir tu llave pública pero nunca tu llave privada.

En la siguiente clase vamos a crear nuestras llaves para compartir archivos con GitHub sin correr el riesgo de que sean interceptados.

# Configura tus llaves SSH en local

**Primer paso: Generar tus llaves SSH**. Recuerda que es muy buena idea proteger tu llave privada con una contraseña.

ssh-keygen -t rsa -b 4096 -C "tu@email.com"

**Segundo paso**: Terminar de configurar nuestro sistema.

**En Windows y Linux**:

# Encender el "servidor" de llaves SSH de tu computadora:

eval $(ssh-agent -s)

# Añadir tu llave SSH a este "servidor":

ssh-add ruta-donde-guardaste-tu-llave-privada

**En Mac**:

# Encender el "servidor" de llaves SSH de tu computadora:

eval "$(ssh-agent -s)"

# Si usas una versión de OSX superior a Mac Sierra (v10.12)

# debes crear o modificar un archivo "config" en la carpeta

# de tu usuario con el siguiente contenido (ten cuidado con

# las mayúsculas):

Host \*

AddKeysToAgent yes

UseKeychain yes

IdentityFile ruta-donde-guardaste-tu-llave-privada

# Añadir tu llave SSH al "servidor" de llaves SSH de tu

# computadora (en caso de error puedes ejecutar este

# mismo comando pero sin el argumento -K):

ssh-add -K ruta-donde-guardaste-tu-llave-privada

# Conexión a GitHub con SSH

Luego de crear nuestras llaves SSH podemos entregarle la llave pública a GitHub para comunicarnos de forma segura y sin necesidad de escribir nuestro usuario y contraseña todo el tiempo.

Para esto debes entrar a la [Configuración de Llaves SSH en GitHub](https://github.com/settings/keys), crear una nueva llave con el nombre que le quieras dar y el contenido de la llave pública de tu computadora.

Ahora podemos actualizar la URL que guardamos en nuestro repositorio remoto, solo que, en vez de guardar la URL con HTTPS, vamos a usar la URL con SSH:

git remote set-url origin url-ssh-del-repositorio-en-github

**Tags y versiones en Git y GitHub**

Los tags o etiquetas nos permiten asignar versiones a los commits con cambios más importantes o significativos de nuestro proyecto.(En fin de cuentas lo que hacen los tags es agrupar una cantidad de comits en específicos agrupándolas como si fueran versiones)

Comandos para trabajar con etiquetas:

* Crear un nuevo tag y asignarlo a un commit: **git tag -a nombre-del-tag id-del-commit**.
* Borrar un tag en el repositorio local: **git tag -d nombre-del-tag**.
* Listar los tags de nuestro repositorio local: **git tag** o **git show-ref --tags**.
* Publicar un tag en el repositorio remoto: **git push origin --tags**.
* Borrar un tag del repositorio remoto: git tag -d nombre-del-tag y **git push origin :refs/tags/nombre-del-tag**.

**Manejo de ramas en GitHub**

Puedes trabajar con ramas que nunca envias a GitHub, así como pueden haber ramas importantes en GitHub que nunca usas en el repositorio local. Lo importantes que aprendas a manejarlas para trabajar profesionalmente.

* Crear una rama en el repositorio local: **git branch nombre-de-la-rama** o **git checkout -b nombre-de-la-rama**.
* Publicar una rama local al repositorio remoto: **git push origin nombre-de-la-rama**.

Recuerda que podemos ver gráficamente nuestro entorno y flujo de trabajo local con Git usando el comando gitk.

# Configurar múltiples colaboradores en un repositorio de GitHub

Por defecto, cualquier persona puede clonar o descargar tu proyecto desde GitHub, pero no pueden crear commits, ni ramas, ni nada.

Existen varias formas de solucionar esto para poder aceptar contribuciones. Una de ellas es añadir a cada persona de nuestro equipo como colaborador de nuestro repositorio.

Solo debemos entrar a la configuración de colaboradores de nuestro proyecto (Repositorio > Settings > Collaborators) y añadir el email o username de los nuevos colaboradores.

# Flujo de trabajo profesional con Pull requests

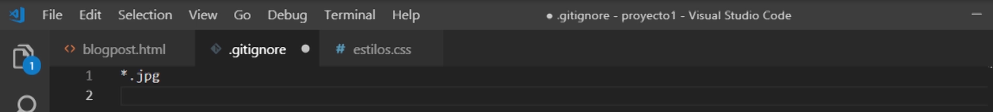
En un entorno profesional normalmente se bloquea la rama **master**, y para enviar código a dicha rama pasa por un code review y luego de su aprobación se unen códigos con los llamados merge request.

Para realizar pruebas enviamos el código a servidores que normalmente los llamamos staging develop (servidores de pruebas) luego de que se realizan las pruebas pertinentes tanto de código como de la aplicación estos pasan a el servidor de producción con el ya antes mencionado merge request.

# Ignorar archivos en el repositorio con .gitignore

No todos los archivos que agregas a un proyecto deberían ir a un repositorio, por ejemplo cuando tienes un archivo donde están tus contraseñas que comúnmente tienen la extensión .env o cuando te estás conectando a una base de datos; **son archivos que nadie debe ver**.

En la raíz del proyecto se debe crear un archivo con el nombre “.gitignore” y dentro se colocan los archivos que queremos ignorar ara que no se suban al repositorio



# Readme.md es una excelente práctica

README.md es una excelente práctica en tus proyectos, md significa **Markdown** es un especie de código que te permite cambiar la manera en que se ve un archivo de texto.

Lo interesante de Markdown es que funciona en muchas páginas, por ejemplo la edición en Wikipedia; es un lenguaje intermedio que no es HTML, no es texto plano, es una manera de crear excelentes texto formateados.

Editor online para creación de Readme.md 🡪 <https://pandao.github.io/editor.md/en.html>

# Tu sitio web público con GitHub Pages

GitHub tiene un servicio de hosting gratis llamado GitHub Pages, tu puedes tener un repositorio donde el contenido del repositorio se vaya a GitHub y se vea online.

Publica tu página en GitHub Pages y compártelo con la comunidad en el área de discusiones de la clase, ¡te esperamos!

Link 🡪 <https://pages.github.com/>

# Git Rebase: Reorganizando el trabajo realizado

El comando rebase es **una mala práctica, nunca se debe usar**, pero para efectos de curso te lo vamos a enseñar para que hagas tus propios experimentos. Con rebase puedes recoger todos los cambios confirmados en una rama y ponerlos sobre otra.

# Cambiamos a la rama que queremos traer los cambios

git checkout experiment

# Aplicamos rebase para traer los cambios de la rama que queremos

git rebase master

# Git Stash: Guardar cambios en memoria y recuperarlos después

Cuando necesitamos regresar en el tiempo porque borramos alguna línea de código pero no queremos pasarnos a otra rama porque nos daría un error ya que debemos pasar ese “mal cambio” que hicimos a stage, podemos usar git stash para regresar el cambio anterior que hicimos.

git stash es típico cuando estamos cambios que no merecen una rama o no merecen un rebase si no simplemente estamos probando algo y luego quieres volver rápidamente a tu versión anterior la cual es la correcta.

**Git Clean: Limpiar tu proyecto de archivos no deseados**

A veces creamos archivos cuando estamos realizando nuestro proyecto que realmente no forman parte de nuestro directorio de trabajo, que no se debería agregar y lo sabemos.

* Para saber qué archivos vamos a borrar tecleamos git clean --dry-run
* Para borrar todos los archivos listados (que no son carpetas) tecleamos git clean -f

# Git cherry-pick: Traer commits viejos al head de un branch

* Existe un mundotag alternativo en el cual vamos avanzando en una rama pero necesitamos en master uno de esos avances de la rama, para eso utilizamos el comando git cherry-pick IDCommit.
* cherry-pick es una mala práctica porque significa que estamos reconstruyendo la historia, **usa cherry-pick con sabiduría**. Si no sabes lo que estás haciendo **ten mucho cuidado**.

# Reconstruir commits en Git con amend

A veces hacemos un commit, pero resulta que no queríamos mandarlo porque faltaba algo más. Utilizamos git commit --amend, amend en inglés es remendar y lo que hará es que los cambios que hicimos nos lo agregará al commit anterior.

**Git Reset y Reflog: Úsese en caso de emergencia**

¿Qué pasa cuando todo se rompe y no sabemos qué está pasando? Con git reset HashDelHEAD nos devolveremos al estado en que el proyecto funcionaba.

* git reset --soft HashDelHEAD te mantiene lo que tengas en staging ahí.
* git reset --hard HashDelHEAD resetea absolutamente todo incluyendo lo que tengas en staging.

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

Con - -> $ git reflog Se puede ver toda la historia incluso si se ha hecho algún reset.

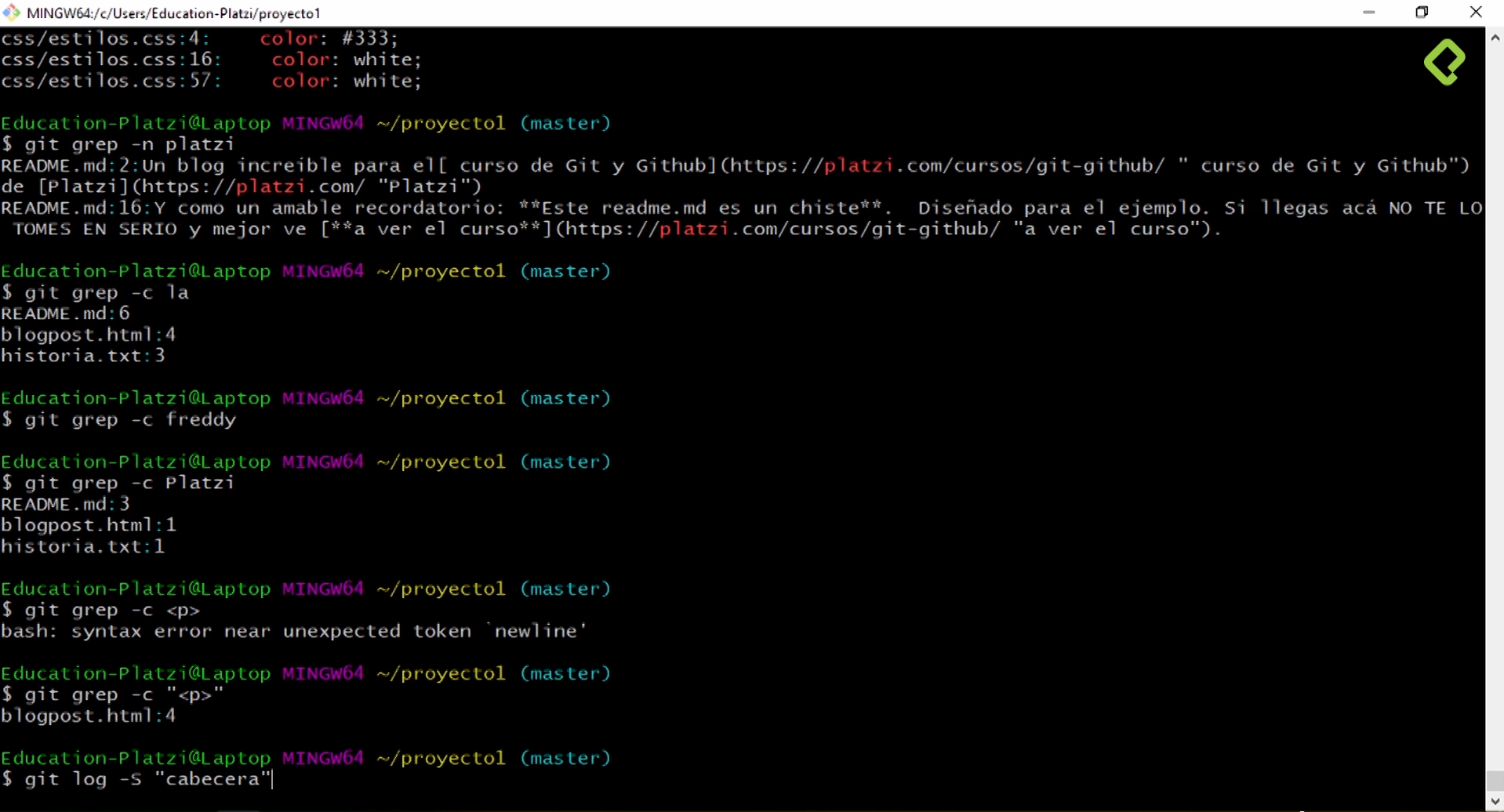
**Buscar en archivos y commits de Git con Grep y log**

A medida que nuestro proyecto se hace grande vamos a querer buscar ciertas cosas.

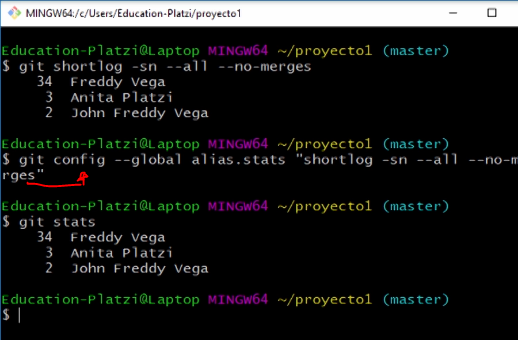
Por ejemplo: ¿cuántas veces en nuestro proyecto utilizamos la palabra *color*?

Para buscar utilizamos el comando git grep color y nos buscará en todo el proyecto los archivos en donde está la palabra *color*.

* Con git grep -n color nos saldrá un output el cual nos dirá en qué línea está lo que estamos buscando.
* Con git grep -c color nos saldrá un output el cual nos dirá cuántas veces se repite esa palabra y en qué archivo.
* Si queremos buscar cuántas veces utilizamos un atributo de HTML lo hacemos con git grep -c "<p>".



CREAR PALABRAS CLAVE QUE EJECUTEN COMANDOS.



En este caso la palabra “stats” va ser lo mismo que ejecutar “git shorlog -sn –all –no-merges”

Solo seria ejecutar el comando 🡪 git stats