

CURSO PROFESIONAL DE GIT Y GITHUB



12 de febrero de 2021

INDICE

1. ¿Por qué usar un sistema de control de versiones como Git?.....	4
1. Instalando Git y GitBash en Windows	4
2. Instalando Git en OSX	5
3. Instalando Git en Linux	5
4. Editores de código, archivos binarios y de texto plano	7
5. Introducción a la terminal y línea de comandos	8
6. ¿Qué es el staging y los repositorios? Ciclo básico de trabajo en Git	9
7. ¿Qué es un Branch (rama) y cómo funciona un Merge en Git?.....	12
7.1 Cambios en GitHub: de master a main	12
7.2 Master a Main	13
8. Crea un repositorio de Git y haz tu primer commit.....	14
8.1 Comandos de git aprendidos en esta clase	14
9. Analizar cambios en los archivos de tu proyecto con Git.....	17
10. Volver en el tiempo en nuestro repositorio utilizando reset y checkout	18
12. Git reset vs. Git rm	20
13. Flujo de trabajo básico con un repositorio remoto.....	22
14. Introducción a las ramas o branches de Git	23
15. Fusión de ramas con Git merge	23
16. Resolución de conflictos al hacer un merge.....	24
17. Cambios en GitHub: de master a main	24
18. Uso de GitHub	25
19. Cómo funcionan las llaves públicas y privadas	26
20. Configura tus llaves SSH en local	28
21. Conexión a GitHub con SSH	29
22. Tags y versiones en Git y GitHub	32
23. Manejo de ramas en GitHub.....	32
24. Configurar múltiples colaboradores en un repositorio de GitHub.....	34
25. Flujo de trabajo profesional: Haciendo merge de ramas de desarrollo a master.....	35
26. Flujo de trabajo profesional con Pull requests.....	36

27. Utilizando Pull Requests en GitHub	38
27.1 ¿Qué es DevOps?	38
28. Creando un Fork, contribuyendo a un repositorio	39
29. Haciendo deployment a un servidor (exclusivo para ver en vídeo)	40
30. Hazme un pull request.....	41
31. Ignorar archivos en el repositorio con .gitignore.....	41
32. Readme.md es una excelente práctica	41
33. Tu sitio web público con GitHub Pages.....	41
34. Git Rebase: reorganizando el trabajo realizado	42
35. Git Stash: Guardar cambios en memoria y recuperarlos después.....	42
36. Git Clean: limpiar tu proyecto de archivos no deseados	45
37. Git cherry-pick: traer commits viejos al head de un branch	45
38. Reconstruir commits en Git con amend	45
39. Git Reset y Reflog: úsese en caso de emergencia	45
40. Buscar en archivos y commits de Git con Grep y log.....	46
41. Comandos y recursos colaborativos en Git y GitHub.....	47

Introducción

El presente documento tiene como finalidad recopilar algunos puntos relevantes del Curso Profesional de Git y Github elaborados por Platzi.com.

El presente documento en ningún caso supe el contenido completo de la plataforma, pero si ayuda a tener presente los contenidos relevantes del curso.

Cada título principal corresponde a un vídeo, esto facilita al estudiante a encontrar el contenido preciso que está buscando para profundizar el contenido.

En el documento también están incorporados algunas imágenes y contenidos que suben otros estudiantes del curso y que facilitan el aprendizaje de los contenidos.

Por último, recalcar que es muy importante la práctica autónoma de cada estudiante.

1. ¿Por qué usar un sistema de control de versiones como Git?

Un sistema de control de versiones como Git nos ayuda a guardar el historial de cambios y crecimiento de los archivos de nuestro proyecto.

En realidad, los cambios y diferencias entre las versiones de nuestros proyectos pueden tener similitudes, algunas veces los cambios pueden ser solo una palabra o una parte específica de un archivo específico. Git está optimizado para guardar todos estos cambios de forma atómica e incremental, o sea, aplicando cambios sobre los últimos cambios, estos sobre los cambios anteriores y así hasta el inicio de nuestro proyecto.

- El comando para iniciar nuestro repositorio, o sea, indicarle a Git que queremos usar su sistema de control de versiones en nuestro proyecto, es `git init`.
- El comando para que nuestro repositorio sepa de la existencia de un archivo o sus últimos cambios es `git add`. Este comando no almacena las actualizaciones de forma definitiva, solo las guarda en algo que conocemos como “Staging Area” (no te preocupes, lo entenderemos más adelante).
- El comando para almacenar definitivamente todos los cambios que por ahora viven en el staging area es `git commit`. También podemos guardar un mensaje para recordar muy bien qué cambios hicimos en este commit con el argumento `-m "Mensaje del commit"`.
- Por último, si queremos mandar nuestros commits a un servidor remoto, un lugar donde todos podamos conectar nuestros proyectos, usamos el comando `git push`.

1. Instalando Git y GitBash en Windows

Windows y Linux tienen comandos diferentes, graban el enter de formas diferentes y tienen muchas otras diferencias. Cuando instales Git Bash en Windows debes elegir si prefieres trabajar con la forma de Windows o la forma de UNIX (Linux y Mac) .

Ten en cuenta que, normalmente, los entornos de desarrollo profesionales tienen personas que usan sistemas operativos diferentes. Esto significa que, si todos podemos usar los mismos comandos, el trabajo resultará más fácil para todos en el equipo.

Los comandos de UNIX son los más comunes entre los equipos de desarrollo. Así que, a menos que trabajes con tecnologías nativas de Microsoft (por ejemplo, .NET), la recomendación es que elijas la opción de la terminal tipo UNIX para obtener una mejor compatibilidad con todo tu equipo.

<https://git-scm.com/downloads>

[E-Book Pro Git](#)

2. Instalando Git en OSX

La instalación de GIT en Mac es un poco más sencilla. No debemos instalar GitBash porque Mac ya trae por defecto una consola de comandos (la puedes encontrar como “Terminal”). Tampoco debemos configurar OpenSSL porque viene listo por defecto.

OSX está basado en un Kernel de UNIX llamado BSD. Esto significa que hay algunas diferencias entre las consolas de Mac y Linux. Pero no vas a notar la diferencia a menos que trabajes con acceso profundo a las interfaces de red o los archivos del sistema operativo. Ambas consolas funcionan muy parecidas y comparten los comandos que vamos a ver durante el curso.

Puedes descargar Git aquí: <https://git-scm.com>

Puedes verificar que Git fue instalado correctamente con el comando `git --version`.

3. Instalando Git en Linux

Cada distribución de Linux tiene un comando especial para instalar herramientas y actualizar el sistema.

En las distribuciones derivadas de Debian (como Ubuntu) el comando especial es `apt-get`, en Red Hat es `yum` y en ArchLinux es `pacman`. Cada distribución tiene su comando especial y debes averiguar cómo funciona para poder instalar Git.

Antes de hacer la instalación, debemos hacer una actualización del sistema. En nuestro caso, los comandos para hacerlo son `sudo apt-get update` y `sudo apt-get upgrade`.

Con el sistema actualizado, ahora sí podemos instalar Git y, en este caso, el comando para hacerlo es `sudo apt-get install git`. También puedes verificar que Git fue instalado correctamente con el comando `git --version`.



INSTALANDO GIT EN LINUX



Importante

Antes de hacer la instalación, debemos hacer una actualización del sistema. En nuestro caso, los comandos para hacerlo son

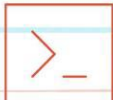
`sudo apt-get update`
y
`sudo apt-get upgrade`

El comando **`sudo`** (del inglés *super user do*) es una utilidad, que permite a los usuarios ejecutar programas con los privilegios administrador (root) de manera segura. Se debe anteponer al comando a ejecutar.

Para verificar que GIT esta instalado y actualizado, abrimos la **Terminal** y escribimos el comando

`git --version`

```
angelo@ANGELO:~$ git --version
git version 2.25.1
angelo@ANGELO:~$
```



Si quieres instalar Git en Linux a través de un instalador binario, en general puedes hacerlo mediante la herramienta básica de administración de paquetes que trae tu distribución.

• DEBIAN/UBUNTU

Para la última versión estable de Debian / Ubuntu: **`apt-get install git`**

Para actualizar GIT en UBUNTU, se puede ejecutar el siguiente comando:

```
$ add-apt-repository ppa:git-core/ppa
$ apt update; apt install git
```

• FEDORA

```
$ yum install git (Hasta la versión Fedora 21)
$ dnf install git (Fedora 22 y posterior)
```

• GENTOO

```
$ emerge --ask --verbose dev-vcs/git
```

• ARCH LINUX

```
$ pacman -S git
```

• OPENSUSE

```
$ zypper install git
```

• MAGEIA

```
$ urpmi git
```

• MAGEIA

```
$ nix-env -i git
```

• FREEBSD

```
$ pkg install git
```

• SOLARIS 9/10/11 (OPENCOSW)

```
$ pkgutil -i git
```

• SOLARIS 11 EXPRESS

```
$ pkg install developer/versioning/git
```

• OPENBSD

```
$ pkg_add git
```

• ALPINE

```
$ apk add git
```

• SLITAZ

```
$ tazpkg get-install git
```

Mayor detalle en 'Download for Linux and Unix'
(<https://git-scm.com/download/linux>)



Los usuarios de Linux pueden administrar Git principalmente desde la línea de comandos. Se recomienda usar Linux debido a su alto rendimiento, a que es Open Source, a su poderosa Terminal, para gestión de Servidores, desarrollo Web, Bash, etc.



4. Editores de código, archivos binarios y de texto plano

Un editor de código es una herramienta que nos brinda muchas ayudas para escribir código, algo así como un bloc de notas muy avanzado. Los editores más populares son VSCode, Sublime Text y Atom, pero no necesariamente debes usar alguno de estos para continuar con el curso.

Tipos de archivos y sus diferencias:

Archivos de Texto (.txt): Texto plano normal y sin nada especial. Lo vemos igual sin importar dónde lo abramos, ya sea con el bloc de notas o con editores de texto avanzados.

Archivos RTF (.rtf): Podemos guardar texto con diferentes tamaños, estilos y colores. Pero si lo abrimos desde un editor de código, vamos a ver que es mucho más complejo que solo el texto plano. Esto es porque debe guardar todos los estilos del texto y, para esto, usa un código especial un poco difícil de entender y muy diferente a los textos con estilos especiales al que estamos acostumbrados.

Archivos de Word (.docx): Podemos guardar imágenes y texto con diferentes tamaños, estilos o colores. Al abrirlo desde un editor de código podemos ver que es código binario, muy difícil de entender y muy diferente al texto al que estamos acostumbrados. Esto es porque Word está optimizado para entender este código especial y representarlo gráficamente.

Recuerda que debes habilitar la opción de ver la extensión de los archivos, de lo contrario, solo podrás ver su nombre. La forma de hacerlo en Windows es `Vista > Mostrar u ocultar > Extensiones de nombre de archivo`.

Editores de texto

- [Atom](#)
- [Sublime Text](#)
- [Visual Studio Code](#)

5. 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`).

6. ¿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).

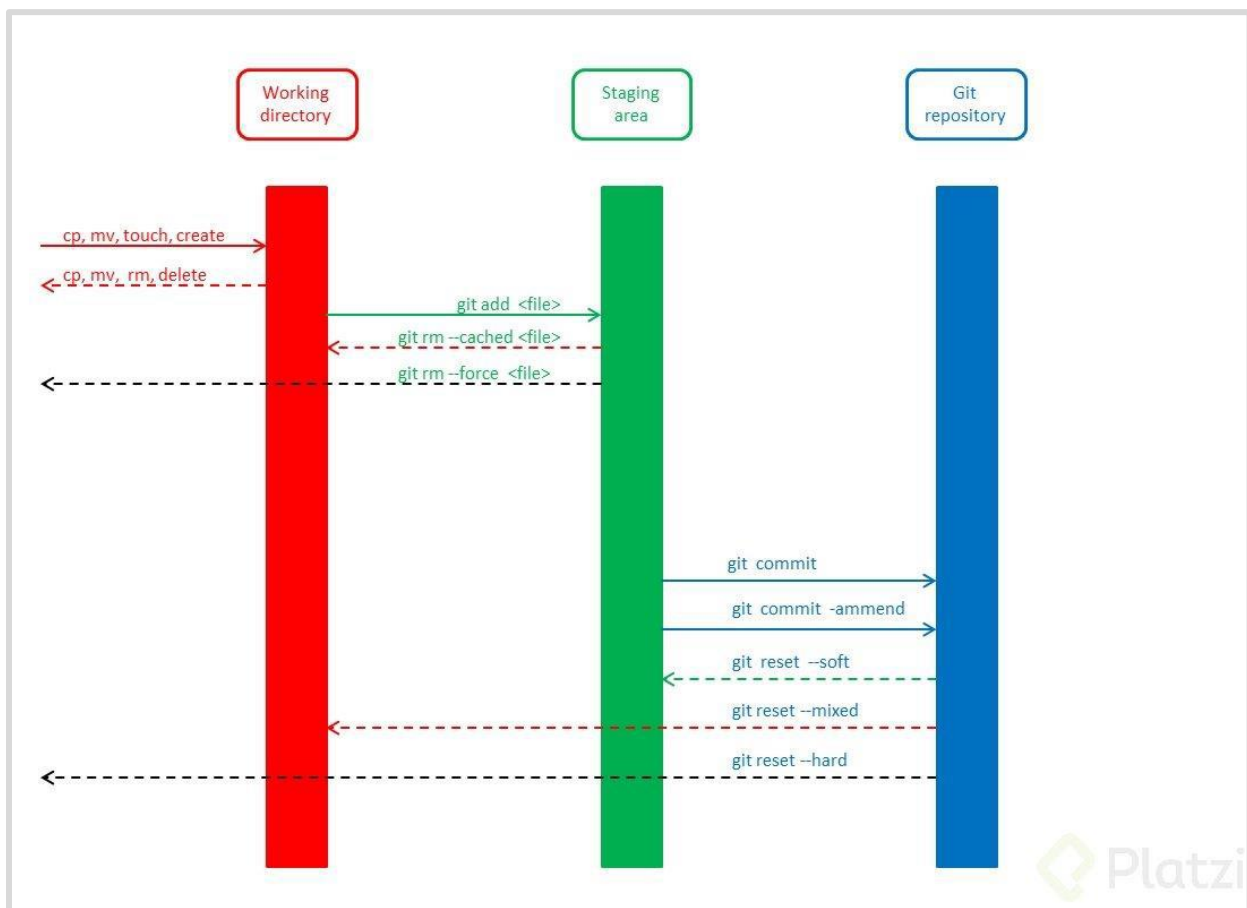
Ciclo de vida o estados de los archivos en Git:

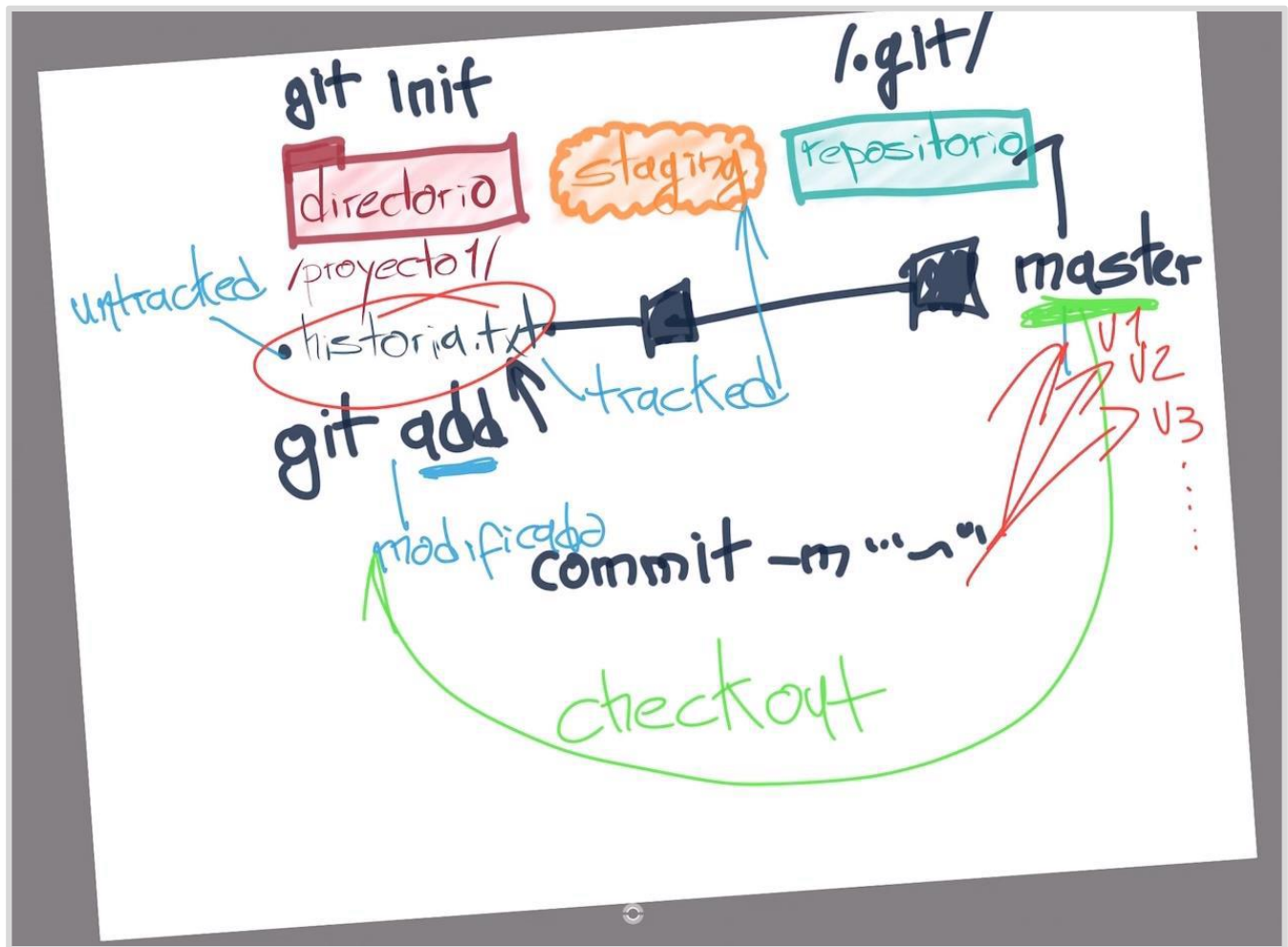
Cuando trabajamos con Git nuestros archivos pueden vivir y moverse entre 4 diferentes estados (cuando trabajamos con repositorios remotos pueden ser más estados, pero lo estudiaremos más adelante):

- **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 Tracked. Esta es una ocasión especial, los archivos han sido guardados 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).





7. ¿Qué es un Branch (rama) y cómo funciona un Merge en Git?

Git es una base de datos muy precisa con todos los cambios y crecimiento que ha tenido nuestro proyecto. Los commits son la única forma de tener un registro de los cambios. Pero las ramas amplifican mucho más el potencial de Git.

Todos los commits se aplican sobre una rama. Por defecto, siempre empezamos en la rama master (pero puedes cambiarle el nombre si no te gusta) y creamos nuevas ramas, a partir de esta, para crear flujos de trabajo independientes.

Crear una nueva rama se trata de copiar un commit (de cualquier rama), pasarlo a otro lado (a otra rama) y continuar el trabajo de una parte específica de nuestro proyecto sin afectar el flujo de trabajo principal (que continúa en la rama master o la rama principal).

Los equipos de desarrollo tienen un estándar: Todo lo que esté en la rama master va a producción, las nuevas features, características y experimentos van en una rama “development” (para unirse a master cuando estén definitivamente listas) y los issues o errores se solucionan en una rama “hotfix” para unirse a master tan pronto como sea posible.

Crear una nueva rama lo conocemos como **Checkout**. Unir dos ramas lo conocemos como **Merge**.

Podemos crear todas las ramas y commits que queramos. De hecho, podemos aprovechar el registro de cambios de Git para crear ramas, traer versiones viejas del código, arreglarlas y combinarlas de nuevo para mejorar el proyecto.

Solo ten en cuenta que combinar estas ramas (sí, hacer “merge”) puede generar conflictos. Algunos archivos pueden ser diferentes en ambas ramas. Git es muy inteligente y puede intentar unir estos cambios automáticamente, pero no siempre funciona. En algunos casos, somos nosotros los que debemos resolver estos conflictos “a mano”.

7.1 Cambios en GitHub: de master a main

El movimiento de #BlackLivesMatter ha ayudado a que GitHub sustituya algunas palabras usadas en su plataforma con relación al racismo.

Palabras como *master*, *whitelist*, *blacklist* y *slave* se encuentran en este proceso de cambio. Pero el más importante en ese momento y que ya ha empezado a tener efecto es que la rama *master* ahora se llamará *main*.

Como ya has aprendido en el [Curso de Git y GitHub](#) nuestra rama principal y a la que apunta nuestro proyecto es **Master**.

...or create a new repository on the command line

```
echo "# master" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:iKenshu/master.git
git push -u origin main
```



...or push an existing repository from the command line

```
git remote add origin git@github.com:iKenshu/master.git
git branch -M main
git push -u origin main
```



7.2 Master a Main

A partir del primero de octubre todos los repositorios **nuevos** que se creen empezarán a mostrar main como rama principal y tendrán que hacer sus comandos respectivos allí.

Los repositorios existentes seguirán usando **master** como rama principal, porque hacer el cambio de nombre en este momento puede generar muchos conflictos para esos proyectos. [Microsoft promete que para final de año](#), este cambio sea más fácil de aplicar para proyectos existentes.

También se agrega un paso adicional, al momento de crear un repositorio desde la línea de comandos. Como vemos en las imágenes ahora debemos hacer:

```
git branch -M main
```

Sabemos que `git branch` nos ayuda a crear una nueva rama dentro de nuestro repositorio y `-M` nos ayudará a mover todo el historial que tengamos (en caso de que los haya) en master a la nueva rama que estamos creando que se llama **main**

Es esperable que estos mismos cambios sean aplicados por otras compañías como Apple y LinkedIn. Creo que palabras como "blacklist" y "whitelist" son muy utilizados en la industria tecnológica, no es mal momento para empezar a cambiar dichos términos por otros.

8. 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).

8.1 Comandos de git aprendidos en esta clase

git init: inicia el repo y crea la carpeta `(.git)`. En la carpeta en la estamos ubicados.

git status: lo usamos para saber si tenemos un archivo añadido o borrado en nuestro proyecto, para saber en la rama en la que estamos y si tenemos commits.

git add: añade el archivo a la base de datos de git

git rm: lo usamos para borrar un archivo que hayamos añadido, para eliminarlo por completo de nuestra rama usamos **git rm --cached**.

git commit: se usa para añadir un archivo a nuestra rama, también podemos ponerle un `-m` al comando seguido de entre comillas para agregar un mensaje.

git config: muestra configuraciones de git también podemos usar `--list` para mostrar la configuración por defecto de nuestro git y si añadimos `--show-origin` nos muestra las configuraciones guardadas y su ubicación.

git config --global user.name: cambia de manera global el nombre del usuario, seguidamente ponemos entre comillas nuestro nombre.

git config --global user.email: cambia de manera global el email del usuario, seguidamente ponemos entre comillas nuestro nombre.

[Más comandos Git](#)

git log: se usa para ver la historia de nuestros archivos, los commits, el usuario que lo cambió, cuando se realizaron los cambios etc. seguidamente ponemos el nombre de nuestro archivo.

Aporte de Andrés Gutiérrez y Maxii Vallejos

**git**

Comandos basicos de Git 

Crea un repositorio de Git y haz tu primer commit

**Ideas**

Es posible combinar los comandos **git add** y **git commit** de la siguiente manera

\$ git commit -am "Mensaje"

este comando solo funcionara si a los archivos ya se les ha aplicado previamente un **git add**.

Quando tengas muchos archivos que añadir puedes usar

\$ git add .

de esta manera le estas diciendo a git que agregue todos los cambios en la carpeta, por que el "." se refiere a la carpeta donde estas posicionado.



**Notas Clase**

Sigue los siguientes pasos para crear un repositorio y hacer tu primer commit.

\$ git init



git init crea un repositorio llamado **.git** donde se guardara el registro de cambios atómicos del proyecto y crea un espacio en RAM llamado **staging**.

\$ git add "Tuarchivo"



git add envia tu archivo a **staging** donde se guarda temporalmente antes de ser enviado al repositorio.

\$ git commit -m "TuMensajedeCommit"



git commit envía el archivo al repositorio confirmando los cambios hechos y dejando un mensaje del usuario al agregar **-m**.

De esta manera haces tu primer commit en el repositorio creado.

**Resumen**

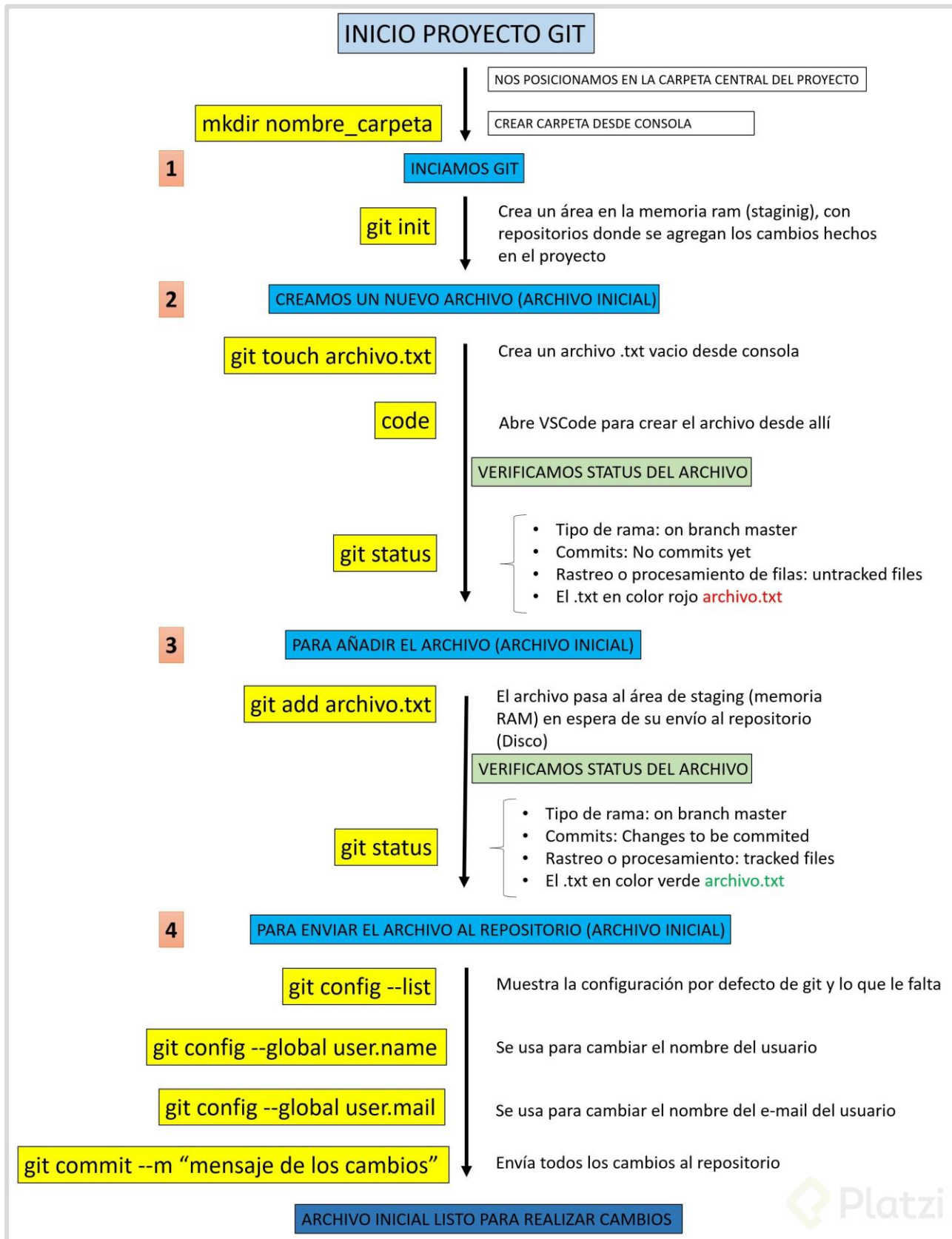
Antes de poder utilizar los comandos de **git** se tiene que ejecutar el comando **git init** en la carpeta donde se quiere usar, de esta manera se activa **git** en la carpeta.

Hay otros comandos que se utilizan en conjunto a los pasos mostrados, como **git status** y **git log**. Estos comandos son para observar como se están comportando los archivos y los **commits** en nuestro entorno de trabajo.

 @YisusJoe



Aporte de Jesús Joel Sarabia



Aporte de Jose Luis Mendoza Mogollón

9. Analizar cambios en los archivos de tu proyecto con Git

El comando `git show` nos muestra los cambios que han existido sobre un archivo y es muy útil para detectar cuándo se produjeron ciertos cambios, qué se rompió y cómo lo podemos solucionar. Pero podemos ser más detallados.

Si queremos ver la diferencia entre una versión y otra, no necesariamente todos los cambios desde la creación del archivo, podemos usar el comando `git diff commitA commitB`.

Recuerda que puedes obtener el ID de tus commits con el comando `git log`.


git

Comandos basicos de Git

Analizar cambios en los archivos de tu proyecto con Git

Ideas

Cuando utilices `git diff` es remomendame poner el **commit** más viejo al principio, de esta manera es mas intuitivo leer lo que nos responde `git`.



Ya que `git` toma el primer **commit** como el más antiguo de los dos.

Notas Clase

El comando `git show` nos muestra los cambios que han existido sobre los dos últimos **commits** de un archivo y es muy útil para detectar cuándo se produjeron cambios en un tiempo cercano y así ver cómo lo podemos solucionar.

```
$ git show
```

Si queremos ver la diferencia entre una versión y otra específicamente, podemos usar el comando.

```
$ git diff commitA commitB
```

Pára obtener los ID de tus commits usas el comando

```
$ git log
```

```
commit cb4f17145be2c4a239dc14b1b9489146348937e0 (HEAD -> master)
Author: TuUser <TuCorreo@gmail.com>
Date: Mon Jun 15 10:10:28 2020 -0600

Añadi carpeta de code

commit 118ba77ef9e30541e9685a901b51fa9411eb1eeb (origin/master)
Merge: 3c46a1ff 4bb5186
Author: TuUser <TuCorreo@gmail.com>
Date: Wed Jun 10 10:00:19 2020 -0600

Merge branch 'master' of github.com:YisusJoe/MonyBott
```

Después ejecutas el comando `git diff` con los **commits** que copiaste

```
$ git diff cb4f17145be2c4a239dc14b1b9489146348937e0 118ba77ef9e30541e9685a901b51fa9411eb1eeb
```

`git` responderá con los cambios realizados del archivo entre los dos **commits**.

Resumen

Los comandos `git show` y `git diff` son mus útiles al momento de buscar porque tu código dejo de funcionar, quien fue él que hizo los cambios y en que fecha se hicieron o simplemente para recordar los cambios en los archivos.

Utiliza estos comandos a tu favor, usa `git show` si quieres cambios entre los dos últimos **commits** pero si quieres ver cambios más viejos utiliza `git diff`.

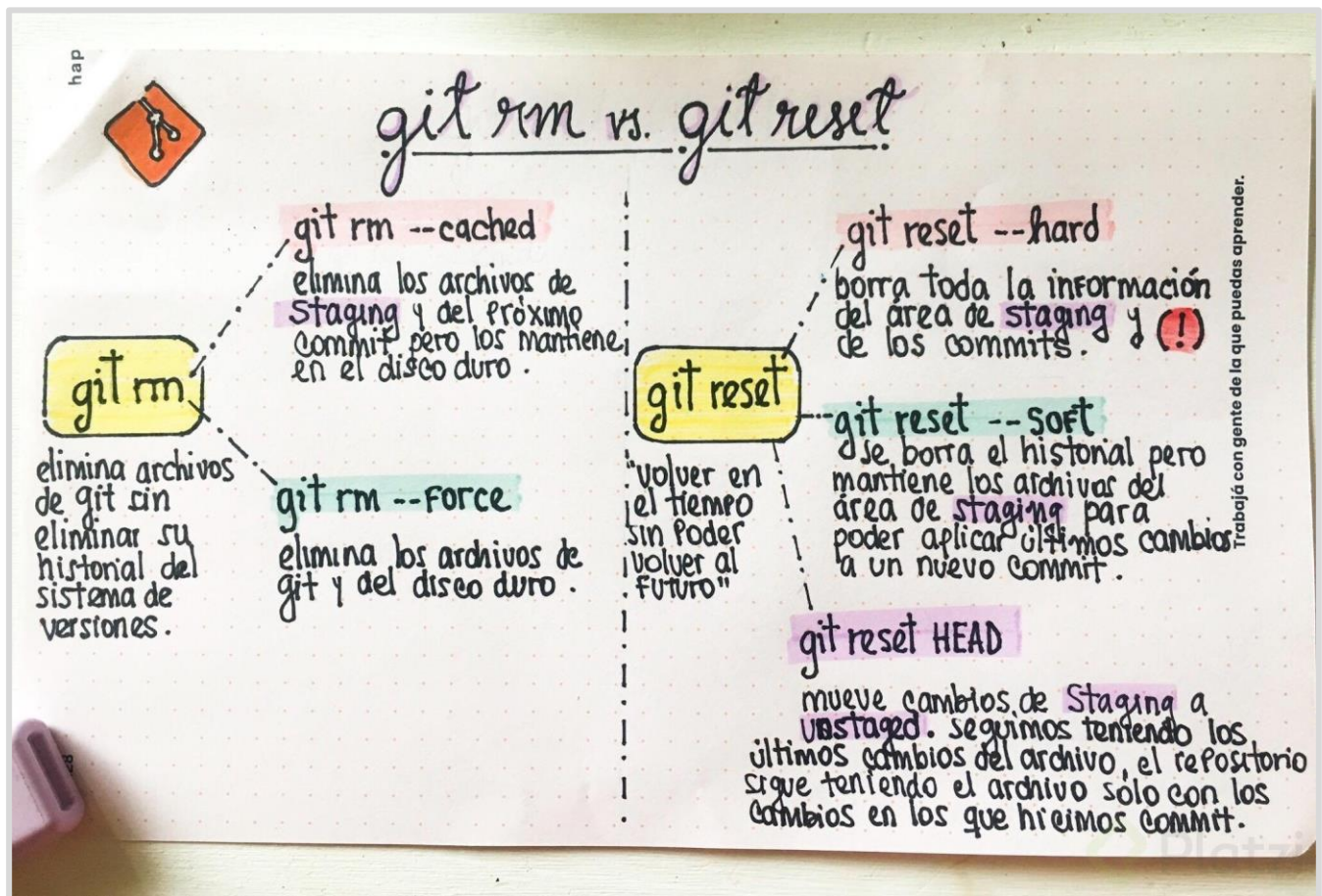
 @YisusJoe

10. Volver en el tiempo en nuestro repositorio utilizando reset y checkout

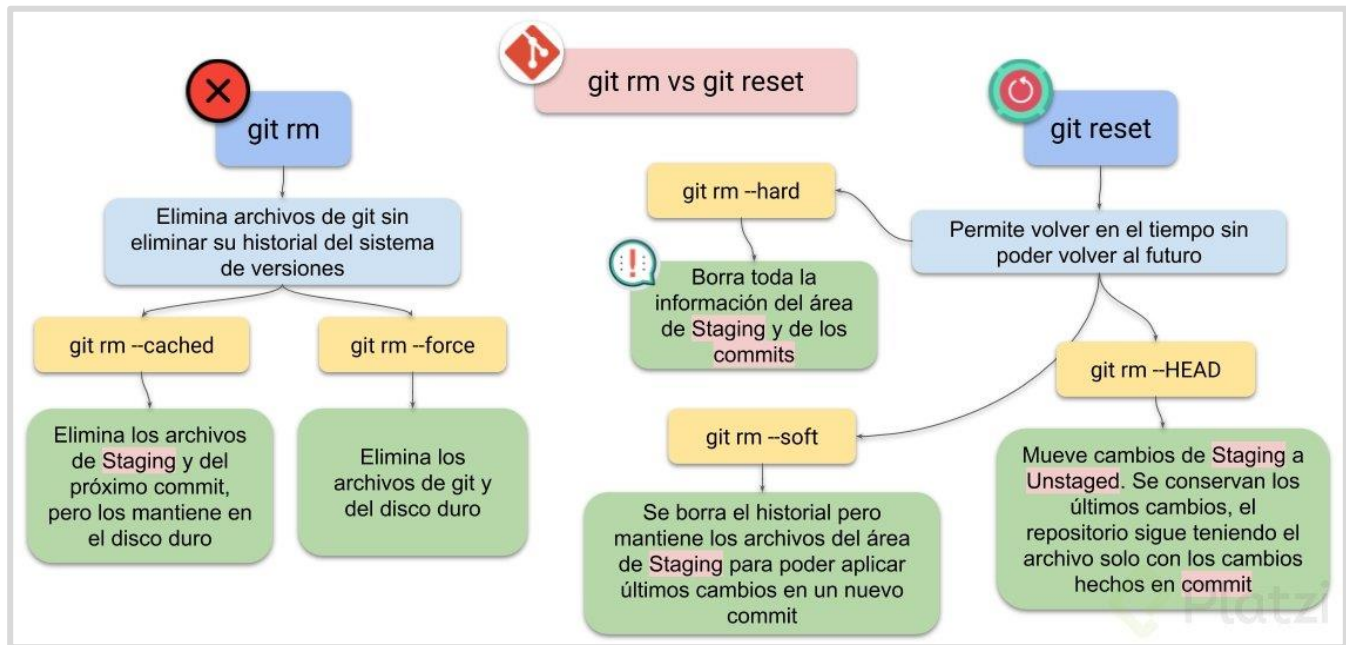
El comando `git checkout + ID` del commit nos permite viajar en el tiempo. Podemos volver a cualquier versión anterior de un archivo específico o incluso del proyecto entero. Esta también es la forma de crear ramas y movernos entre ellas.

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.



Aporte de María José Ledesma



Aporte de Kevin Mejia

12. 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. Borrarnos la historia y la debemos sobrescribir. 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`: Borrarnos 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 qué 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.

¿Cuál es la diferencia entre `git rm --cached` y `git reset HEAD`?

`git rm -cached <file>`: remueve el archivo del índice, esto quiere decir, que Git ya no le hará seguimiento. Aunque el archivo seguirá existiendo en tu directorio, tal y como está.

`git reset HEAD <file>`: devuelve el archivo a su último commit y este sigue en seguimiento por git, es decir podrás hacer add, commit, etc.

Si realizas `git status` podrás darte cuenta de que el archivo al que le aplicaste `git rm -cached <file>` ya no se encuentra en seguimiento por Git porque no aparece.

13. 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 merge` 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.

Algunos comandos que pueden ayudar cuando colaboren con proyectos muy grandes de github

1. `git log --oneline` - Te muestra el id commit y el título del commit.
2. `git log --decorate` - Te muestra donde se encuentra el head point en el log.
3. `git log --stat` - Explica el número de líneas que se cambiaron brevemente.
4. `git log -p` - Explica el número de líneas que se cambiaron y te muestra que se cambió en el contenido.
5. `git shortlog` - Indica que commits ha realizado un usuario, mostrando el usuario y el título de sus commits.
6. `git log --graph --oneline --decorate`
7. `git log --pretty=format:"%cn hizo un commit %h el dia %cd"` - Muestra mensajes personalizados de los commits.
8. `git log -3` - Limitamos el número de commits.
9. `git log --after="2018-1-2"`
10. `git log --after="today"`
11. `git log --after="2018-1-2" --before="today"` - Commits para localizar por fechas.
12. `git log --author="Name Author"` - Commits realizados por autor que cumplan exactamente con el nombre.

13. `git log --grep="INVIE"` - Busca los commits que cumplan tal cual está escrito entre las comillas.
14. `git log --grep="INVIE" -i` - Busca los commits que cumplan sin importar mayúsculas o minúsculas.
15. `git log --index.html` - Busca los commits en un archivo en específico.
16. `git log -S "Por contenido"` - Buscar los commits con el contenido dentro del archivo.
17. `git log > log.txt` - guardar los logs en un archivo txt

Aporte de Hellen Bar

[GitHub](#)

14. 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`).

15. Fusión de ramas con Git merge

El comando `git merge` nos permite crear un nuevo commit con la combinación de dos ramas (la rama donde nos encontramos cuando ejecutamos el comando y la rama que indiquemos después del comando).

```
# Crear un nuevo commit en la rama master combinando
# los cambios de la rama cabecera:

git checkout master
git merge cabecera

# Crear un nuevo commit en la rama cabecera combinando
# los cambios de cualquier otra rama:

git checkout cabecera
git merge cualquier-otra-rama
```

Asombroso, ¿verdad? Es como si Git tuviera super poderes para saber qué cambios queremos conservar de una rama y qué otros de la otra. El problema es que no siempre puede adivinar, sobretodo en algunos casos donde dos ramas tienen actualizaciones diferentes en ciertas líneas en los archivos. Esto lo conocemos como un **conflicto** y aprenderemos a solucionarlos en la siguiente clase.

Recuerda que al ejecutar el comando `git checkout` para cambiar de rama o commit puedes perder el trabajo que no hayas guardado. Guarda tus cambios antes de hacer `git checkout`.

Existe una extensión en Visual Studio Code que permite visualizar esto de forma gráfica, la extensión se llama Git Graph y se puede descargar en el siguiente [link](#) (Aporte de José Tuzinkievicz)

16. Resolución de conflictos al hacer un merge

Git nunca borra nada a menos que nosotros se lo indiquemos. Cuando usamos los comandos `git merge` o `git checkout` estamos cambiando de rama o creando un nuevo commit, no borrando ramas ni commits (recuerda que puedes borrar commits con `git reset` y ramas con `git branch -d`).

Git es muy inteligente y puede resolver algunos conflictos automáticamente: cambios, nuevas líneas, entre otros. Pero algunas veces no sabe cómo resolver estas diferencias, por ejemplo, cuando dos ramas diferentes hacen cambios distintos a una misma línea.

Esto lo conocemos como **conflicto** y lo podemos resolver manualmente, solo debemos hacer el merge, ir a nuestro editor de código y elegir si queremos quedarnos con alguna de estas dos versiones o algo diferente. Algunos editores de código como VSCode nos ayudan a resolver estos conflictos sin necesidad de borrar o escribir líneas de texto, basta con hundir un botón y guardar el archivo.

Recuerda que siempre debemos crear un nuevo commit para aplicar los cambios del merge. Si Git puede resolver el conflicto hará commit automáticamente. Pero, en caso de no pueda resolverlo, debemos solucionarlo y hacer el commit.

Los archivos con conflictos por el comando `git merge` entran en un nuevo estado que conocemos como **Unmerged**. Funcionan muy parecido a los archivos en estado Unstaged, algo así como un estado intermedio entre Untracked y Unstaged, solo debemos ejecutar `git add` para pasarlos al área de staging y `git commit` para aplicar los cambios en el repositorio.

17. Cambios en GitHub: de master a main

El escritor Argentino Julio Cortázar afirma que las palabras tienen color y peso. Por otro lado, los sinónimos existen por definición, pero no expresan lo mismo. Feo no es lo mismo que desagradable, ni aromático es lo mismo que oloroso.

Por lo anterior podemos afirmar que los sinónimos no expresan lo mismo, no tienen el mismo “color” ni el mismo “peso”.

Sí, esta lectura es parte del curso profesional de Git & GitHub. Quédate conmigo.

Desde el 1 de octubre de 2020 GitHub cambió el nombre de la rama principal: ya no es “master” -como aprenderás en el curso- sino main.

Este derivado de una profunda reflexión ocasionada por el movimiento #BlackLivesMatter.

La industria de la tecnología lleva muchos años usando términos como master, slave, blacklist o whitelist y esperamos pronto puedan ir desapareciendo.

Y sí, las palabras importan. Por lo que de aquí en adelante cada vez que escuches a Freddy mencionar “master” debes saber que hace referencia a “main”

Puedes leer un poco más aquí: [Cambios en GitHub: de master a main](#)

18. 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 descargará 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
```

19. 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 sin correr el riesgo de que sean interceptados por personas con malas intenciones.

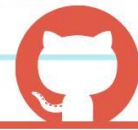
La forma de hacerlo es la siguiente:

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.

CÓMO FUNCIONAN LAS LLAVES PÚBLICAS Y PRIVADAS



También conocido como "Cifrado asimétrico de un solo camino"

Las llaves se crean con un proceso algorítmico y están vinculadas matemáticamente una con la otra, de esta manera están asociadas.

Incluso aunque tuvieras cientos de miles de computadoras y cada uno de ellos fuera capaz de intentar mil billones de claves cada segundo costaría billones de billones de años probar todas las posibilidades, solo para romper UN SOLO MENSAJE.

Este método es la base de todo el intercambio seguro de mensajes en la internet abierta, incluyendo los protocolos de seguridad conocidos como SSL y TLS que nos protegen cuando estamos navegando en la web.

<https://www.s>

Conforme los ordenadores sean más y más rápidos tendremos que desarrollar nuevas formas de hacer más difícil a los ordenadores romper el cifrado.

La criptografía asimétrica, también llamada criptografía de clave pública o criptografía de dos claves, es el método criptográfico que asegura que un mensaje enviado no pueda ser leído por ninguna otra persona que la persona destinataria del mensaje.

Llaves públicas y privadas

Una llave es pública y se puede entregar a cualquier persona, la otra llave es privada y el propietario debe guardarla de modo que **nadie tenga acceso a ella**.

THE KEYPAIR

PRIVATE KEY

PUBLIC KEY



¿CÓMO FUNCIONA EL CIFRADO ASIMÉTRICO?

Todos los usuarios generan dos archivos llamados "llaves": una pública y una privada.



Las llaves públicas son visibles para todo el mundo



Para enviar un mensaje a María, Pedro cifra el mensaje con la llave pública de María y luego lo FIRMA con su propia llave privada.



Si alguien captura el mensaje no podrá leerlo sin la llave privada de María.

María confirma que el mensaje es de Pedro usando la llave pública de Pedro y luego descifra el mensaje con su propia llave privada.



Las llaves públicas y privadas sirven para compartir información en Internet de una forma segura, incluso si el mensaje es interceptado la probabilidad de que se pueda descifrar es casi nula; si no se tiene la llave privada. Este método se usa incluso en el mundo financiero.

Importante: Nunca compartir la llave privada porque con esta, pueden acceder a tus proyectos, incluso los de tus clientes y perder TU información.



20. 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
```

Generar una nueva llave SSH: (Cualquier sistema operativo)

```
ssh-keygen -t rsa -b 4096 -C "youremail@example.com"
```

Comprobar proceso y agregarlo (Windows)

```
eval $(ssh-agent -s)
```

```
ssh-add ~/.ssh/id_rsa
```

Comprobar proceso y agregarlo (Mac)

```
eval "$(ssh-agent -s)"
```

¿Usas macOS Sierra 10.12.2 o superior?

Haz lo siguiente:

```
cd ~/.ssh
```

Crea un archivo config...

Con Vim `vim config`

Con VSCode `code config`

Pega la siguiente configuración en el archivo...

```
Host *
  AddKeysToAgent yes
  UseKeychain yes
  IdentityFile ~/.ssh/id_rsa
```

Agrega tu llave

```
ssh-add -K ~/.ssh/id_rsa
```

[Aporte Juan Luis Rojas León](#)

21. 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](#), 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
```

APUNTES CURSO GIT-GITHUB (REPOSITORIO REMOTO)

\$git remote add origin <https-url>	Establecer un origen remoto: sede del repositorio remoto para gestionar nuestro proyecto mediante conexión HTTPS
\$git remote -v	Verifica la existencia del origen remoto
\$git config -l	Permite ver los parámetros de configuración
\$git push origin master 	Fusiona una copia del master local con el remoto
\$git pull origin master 	Fusiona una copia del master remoto con el local
\$git config -l valores	Permite ver los parámetros de la configuración y sus valores
\$git config --global user.email "email"	Permite modificar el valor de la variable user.email en la configuración
\$git remote set-url origin <ssh-url> través de SSH (en lugar de HTTPS)	Configura git para conectar con el repositorio remoto a través de SSH

SSH: GENERACIÓN DE CLAVES PÚBLICA Y PRIVADA

\$ssh-keygen -t rsa -b 4096 -C "email"	Generación de claves de cifrado pública y privada
\$eval \$(ssh-agent -s)	Comprueba si el servicio de cifrado está activo
\$ssh-add <ruta-id_rsa> contiene la llave privada	Informa al sistema de la ubicación del archivo que contiene la llave privada

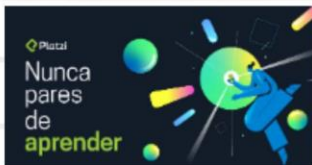
Posteriormente, hay que proporcionar a GitHub una copia de nuestra llave pública:
En la ruta: **Personal Settings/SSH and GPG keys/SSH keys/** hay que copiar el contenido del archivo **id_rsa.pub** que se generó con el comando ssh-keygen.

Aporte Ernesto Ventas Fernandez

CONEXIÓN A GITHUB CON SSH



Tutorial recomendado



Configurar llaves SSH en Git y GitHub

¿Para qué necesitamos la criptografía asimétrica?
Cuando enviamos datos por Internet, ya sea una
imagen, un archivo o sólo un simple mensa

ssh-keygen -p #comando
para cambiar la
passphrase existente sin
volver a generar el par de
claves

```
ssh -T git@github.com
```

#Comando para probar la conexión con Github, se ingresa con la passphrase adicional si se agregó.

```
git config --global
user.email "email"
```

#Comando para cambiar el email de nuestro usuario, debe ser el mismo que tenemos en GitHub para que este, nos pueda identificar.

Warning: Permanently added the RSA host key for IP address '140.82.114.3' to the list of known hosts.

Mensaje que asegura que la IP de Github es reconocida como de confianza, no es mensaje de error, solo de advertencia.

Cada usuario, cada computadora, cada persona debe tener una llave privada y pública única, no es buena practica compartir las llaves.

- **AGREGAR UNA CLAVE SSH NUEVA A TU CUENTA DE GITHUB**

- 1) Ubicar la carpeta `.ssh` oculta, abrir el archivo en tu editor de texto favorito, y copiarlo en tu portapapeles.

2) En la esquina superior derecha de tu cuenta en Github.com, da clic en tu foto de perfil y después da clic en Configuración.

3) En la barra lateral de configuración de usuario, da clic en Llaves SSH y GPG.



4) Haz clic en New SSH key (Nueva clave SSH) o Add SSH key (Agregar clave SSH).

5) En el campo "Title" (Título), agrega una etiqueta descriptiva para la clave nueva. Por ejemplo, si estás usando tu Mac personal, es posible que llames a esta "Personal MacBook Air". Copia tu clave en el campo "Key" (Clave).



7) Si se te solicita, confirma tu contraseña GitHub.



- CAMBIAR LA URL DE UN REMOTO

1) Abre la Terminal y cambiar el directorio de trabajo actual en tu proyecto local.

2) Enumerar tus remotos existentes a fin de obtener el nombre de los remotos que deseas cambiar.

```
Proyecto1 git/master
> git remote -v
origin https://github.com/ayenque/hyperblog.git (fetch)
origin https://github.com/ayenque/hyperblog.git (push)
```

5) Antes de cualquier cambio ejecutar el comando `git pull` y nos aparece un mensaje de advertencia, dandole "yes", luego nuevamente `git pull origin master`

```
#git pull --master
$ git pull
The authenticity of host 'github.com [140.82.111.3]' can't be established.
RSA fingerprint is SHA256:nThbg6vXORQJGwgKTcE678NGVxLWtqFkGzADZZ9RrYw==.
Are you sure you want to continue connecting (yes/no) (fingerprint is y)?
Please type yes, 'no' or the fingerprint!
y
Warning: Permanently added 'github.com' (RSA) to the list of known hosts.
Your configuration file was updated with the following details:
No key information for master was found.
For favor especifico a que rama quieres fumarstar.
ver git pull --> para detalles.

$ git pull --remote <rama>
$ git pull --remote=origin master

Si deseas configurar el rastreo de información para esta rama, puedes hacerlo con:
$ git branch --set-upstream-to=origin/<rama> master
```

3) Cambiar tu URL remota de HTTPS a SSH con el comando `git remote set-url`

```
Proyecto1 git/master
> git remote set-url origin git@github.com:ayenque/hyperblog.git
```

4) Verificar que la URL remota ha cambiado.

```
Proyecto1 git/master
> git remote -v
origin git@github.com:ayenque/hyperblog.git (fetch)
origin git@github.com:ayenque/hyperblog.git (push)
```

6) Hacemos los cambios en nuestro repositorio local y los confirmamos, hacemos git pull y luego para enviar los cambios git push origin master

```
#project gitmaster
$ git commit -m "Una versión del Hypervisor"
master [cd836]! Una versión del Hypervisor
# file changed, 3 insertion(+), 1 deletion(-)

# project gitmaster
$ git push origin master
Warning: Permanently added the RSA host key no: IP address [10.02.114.4] to the list of known hosts
Wrote ssh://hub.com:2022/projectgitmaster
# branch master      -> FETCH_HEAD
Ya está actualizado.

#project gitmaster
$ git push origin master
Enumerado objetos: 1, listo.
Comprimiendo objetos: 100.0%, listo.
Compresión de datos usando método RLE
Comprimiendo objetos: 100.0%, listo.
Enchufando objetos: 100.0%, 11 bytes / 215.00 KiB/s, listo.
Limpio (en total 7) transacciones.
remote: Receiving objects: 100% (2/2), completed with 2 local objects.
to ssh://hub.com:2022/projectgitmaster: master
777f635..c8d16dc master -> master

#project gitmaster
```



Luego de haber creado las llaves que están en nuestro Home, agregamos la publica a Github por perfil web, también debemos cambiar la URL con `git remote set-url` del enlace https previo. Como primer paso debemos traernos los cambios del servidor tomando en cuenta las advertencias de primer uso, no olvidar que debemos hacer esto (`git pull`) siempre antes de enviar un cambio (`git push`).

22. 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.

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`.

23. Manejo de ramas en GitHub

Puedes trabajar con ramas que nunca envías a GitHub, así como pueden haber ramas importantes en GitHub que nunca usas en el repositorio local. Lo importante es 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`.

MANEJO DE RAMAS EN GITHUB



Puedes trabajar con ramas que nunca envías 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.

En caso de error al momento de ejecutar gitk:

```
Proyecto1 git/master
> gitk
zsh: command not found: gitk
```

`sudo apt install gitk`
#Comando para instalar gitk, luego de instalar con éxito ejecutar `gitk` nuevamente

No olvidar siempre ejecutar primero **`git pull origin master`**

`git push origin header footer` # Se pueden enviar varias ramas a la vez al remoto, solo listando sus nombres después de origin.

`git push origin --delete nombre-rama` # Comando para eliminar un rama remota (en Github). Básicamente, lo que hace es eliminar el apuntador del servidor.

• `git branch`

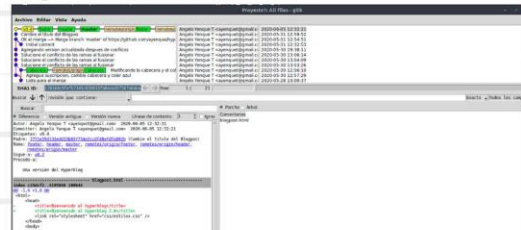
Comando para revisar las ramas creadas, pero si queremos ver más detalles podemos usar los siguientes comandos.

• `git show-branch`

• `git show-branch --all`

Comando para revisar las ramas creadas con su ubicación (remoto o local), pero además nos muestra la historia mas reciente de esas ramas y sus commits.

• `gitk`



```
Proyecto1 git/cabecera
> git show-branch
[cabecera] Modificando la cabecera y el color del texto
[cabecera] Una versión del Hyperblog
[cabecera] Cambie el título del Blog
[cabecera] Inicial commit
[cabecera] Agregando versión actualizada después de conflictos
[cabecera] Solucione el conflicto de las ramas al fusionar
[cabecera] Solucione el conflicto de las ramas al fusionar
[cabecera] Modificando la cabecera y el color del texto
Proyecto1 git/cabecera
> git show-branch --all
[cabecera] Modificando la cabecera y el color del texto
[cabecera] Una versión del Hyperblog
[origin/master] Una versión del Hyperblog
[cabecera] Una versión del Hyperblog
[cabecera] Cambie el título del Blog
[cabecera] Inicial commit
[cabecera] de el Merge -> Merge branch 'master' of https://github.com/ayenque/hyperblog
[cabecera] Inicial commit
[cabecera] Agregando versión actualizada después de conflictos
[cabecera] Solucione el conflicto de las ramas al fusionar
```

Con `gitk` podemos ver toda la historia de nuestro proyecto en un software para verlo de forma visual.

VAMOS A CREAR DOS RAMAS, FOOTER Y HEADER Y LUEGO LA ENVIAREMOS A GITHUB.

Nos ubicamos en la rama master con `git checkout master`

Creamos las dos ramas: `git branch header` y `git branch footer`

• `git push origin "nombre rama"`

Comando para enviar a Github (origin), una rama específica.

> `git push origin header`
> `git push origin footer`

Con esto enviamos las ramas a nuestro repositorio remoto (Github).

Revisamos en Github para corroborar que las ramas fueron enviadas correctamente.

```
Proyecto1 git/master
> git branch header
Proyecto1 git/master
> git branch footer
```

```
Proyecto1 git/master
> git push origin header
Total 0 (delta 0), reusado 0 (delta 0)
remote:
remote: Create a pull request for 'header' on GitHub by visiting:
remote: https://github.com/ayenque/hyperblog/pull/new/header
remote:
To github.com:ayenque/hyperblog.git
* [new branch] header -> header

Proyecto1 git/master
> git push origin footer
Total 0 (delta 0), reusado 0 (delta 0)
remote:
remote: Create a pull request for 'footer' on GitHub by visiting:
remote: https://github.com/ayenque/hyperblog/pull/new/footer
remote:
To github.com:ayenque/hyperblog.git
* [new branch] footer -> footer
```



Las ramas en Github son importantes porque representan un área de trabajo independiente de desarrollo dentro de nuestro proyecto. Al igual que en nuestro repositorio local, en Github podemos trabajar con ramas y nos permiten de la misma manera, traernos los cambios realizados en otras ramas y compararlos para unirlos con nuestros cambios, utilizando Git.

24. 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.

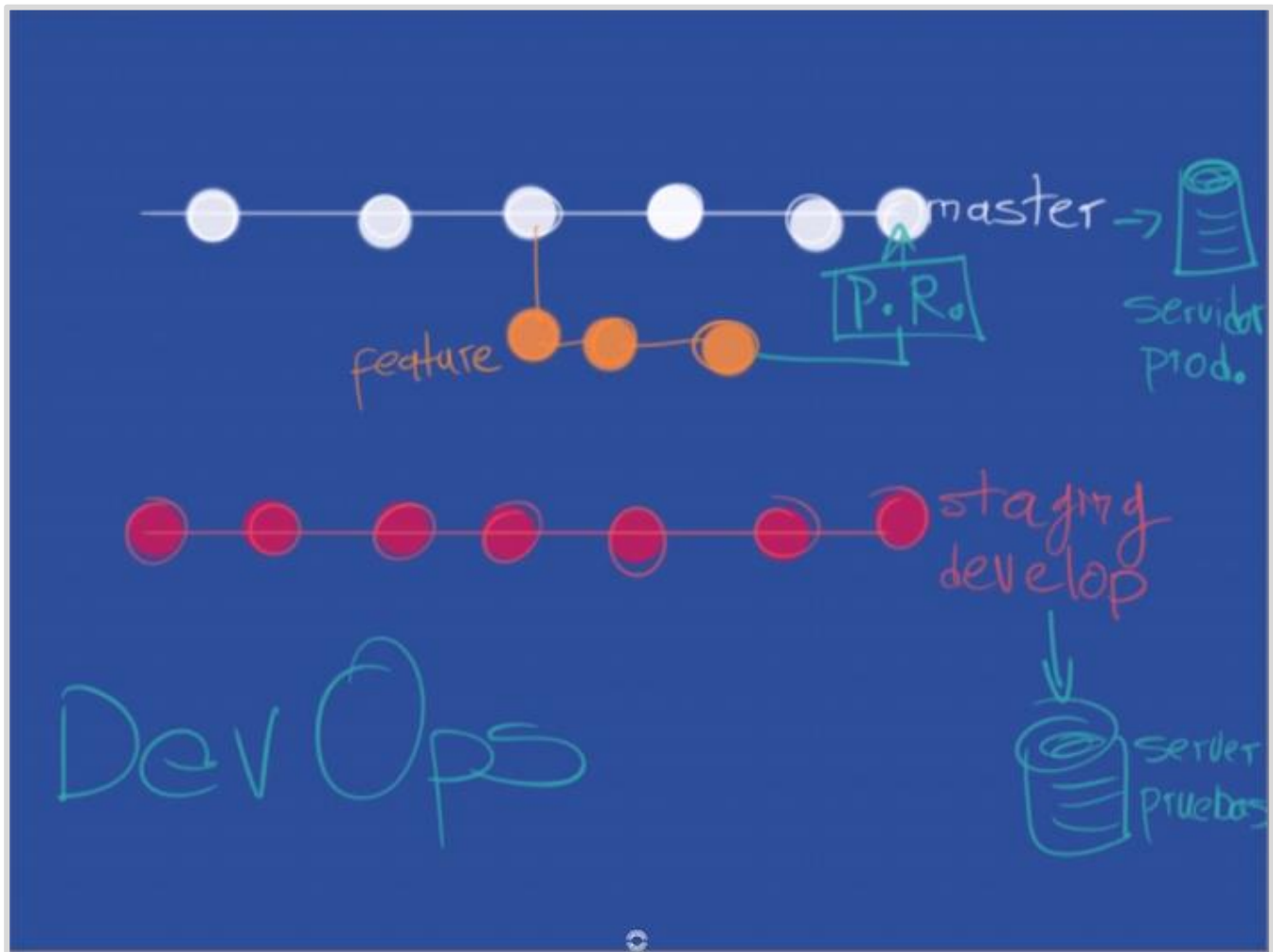
25. Flujo de trabajo profesional: Haciendo merge de ramas de desarrollo a master



26. 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*.



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 **pull request**.

Los **pull request** no es una característica de Git, si no de Github.

Los **pull request** también son importantes porque permiten a personas que no son colaboradores, trabajar y apoyar en nuestro proyecto.

Equivalencia en otras plataformas:

Pull Request	Pull Request	Merge Request

De acuerdo a diversos estudios, resulta más barato encontrar y corregir incidencias en etapas tempranas del desarrollo que encontrarlas y corregirlas en producción. De hecho, algunos estudios señalan que es 10 veces más caro corregir por cada fase del proceso que pasa.

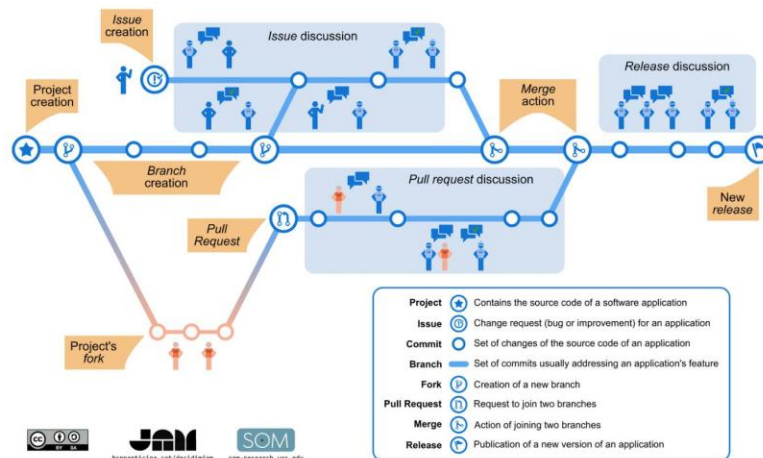
Para realizar pruebas enviamos el código a servidores que normalmente los llamamos **staging server**, luego de que se realizan las pruebas pertinentes tanto de código como de la aplicación estos pasan a el **servidor de producción**.



PULL REQUESTS

Es la acción de validar un código que se va a **mergear** de una rama a otra. En este proceso de validación pueden entrar los factores que queramos: Builds (validaciones automáticas), asignación de código a tareas, validaciones manuales por parte del equipo, despliegues, etc.

How GitHub projects are developed?
Where are the main discussion points?



La persona que hace todo esto, normalmente son los líderes de equipo o un perfil muy especial que se llama **DevOps** (permite que los roles que antes estaban aislados se coordinen y colaboren para producir productos mejores y más confiables).



Los pull request podrían compararse con un control de calidad interno donde el equipo tiene la oportunidad de detectar bugs o código que no sigue lineamientos, convenciones o buenas prácticas. Incluso puede presentar ahorros a la empresa. **GitHub** nos permite llevar un control e implementa un proceso para la atención y revisión de estas solicitudes.

27. Utilizando Pull Requests en GitHub

Es una funcionalidad de github (en gitlab llamada merge request y en bitbucket push request), en la que un colaborador pide que revisen sus cambios antes de hacer merge a una rama, normalmente master.

Al hacer un pull request se genera una conversación que pueden seguir los demás usuarios del repositorio, así como autorizar y rechazar los cambios.

El flujo del pull request es el siguiente

1. Se trabaja en una **rama paralela** los cambios que se desean (`git checkout -b <rama>`)
2. Se hace un **commit** a la rama (`git commit -am '<Comentario>'`)
3. Se **suben al remoto** los **cambios** (`git push origin <rama>`)
4. En GitHub se hace el `pull request` comparando la **rama master** con la rama del **fix**.
5. Uno, o varios colaboradores revisan que el **código sea correcto** y dan **feedback** (en el chat del pull request)
6. El colaborador hace los cambios que desea en la **rama** y lo **vuelve a subir** al remoto (automáticamente jala la historia de los cambios que se hagan en la rama, en remoto)
7. Se **aceptan los cambios** en GitHub
8. Se hace **merge** a `master` desde GitHub

Importante: Cuando se modifica una `rama`, también se modifica el `pull request`.

27.1 ¿Qué es DevOps?

Hay muchas posturas sobre si DevOps (Developer Operations) es una **disciplina**, **carrera** u **ocupación**. La gran mayoría coinciden en que es una cultura que promueve la comunicación e integración entre desarrolladores de software y los equipos encargados de la infraestructura de los servidores.

Su objetivo: **acelerar la entrega de nuevas versiones de software de manera ágil asegurando el mayor tiempo en línea y calidad del software**, todo esto, basándose en automatización, pruebas de calidad y la correcta administración de la infraestructura.

¿Qué necesito para implementar una cultura de DevOps?

Lo primero es **comunicar que la cultura DevOps se va a implementar**, mencionando los beneficios y haciendo que el equipo se sienta parte del cambio. Luego explicar las acciones que se van a tomar, no importa que no sean desarrolladores.

La **comunicación** y **colaboración** para tener una cultura DevOps son cruciales, es un trabajo entre los desarrolladores y los equipos encargados de la infraestructura de los servidores.

La tarea principal será la **automatización**, reducir los tiempos de despliegue del producto y mantener una calidad de desarrollo y estabilidad para el usuario. Debes tener claro que esto será un proceso sin fin, por lo que cada mejora te dará el tiempo que dedicarás para innovar el proceso y hacerlo cada vez mejor.

Conceptos básicos que debes saber para empezar

Infraestructura como código

Aquí es donde pasa la magia. Básicamente es colocar en código toda la infraestructura de tu servidor con el fin de que al ejecutarlo, tienes una instancia con todo lo necesario para que tu aplicación esté corriendo. Si un servidor se cae, puedes ejecutar tu código con la infraestructura y en cuestión de minutos tener tu aplicación corriendo de nuevo.

Continuous Integration

Es una serie de pasos que tú decides seguir o no, pero que entre más sigas, más te ayudarán a hacer integraciones de código lo más rápido posible. Es decir, al hacer un nuevo feature que va a un ambiente de Quality Assurance, puedes hacer pruebas automáticas y en caso de salir exitosas, se hace un deploy del nuevo feature.

Continuous Delivery

Esta es la evolución de CI (Continuous Integration). Es el flujo para hacer deploy automático, basado en testing e infraestructura necesaria para cada feature. Esto le da la capacidad a los desarrolladores de mover código a producción en cualquier momento.

Todo esto es con el fin de asegurar entregas de nuevas características de tu aplicación, lo más frecuente posible y de la manera más eficiente. Al final **es un conjunto de procesos y cambios en la cultura del equipo en pro de mejorar el flujo de desarrollo** y despliegue de nuestras aplicaciones y puedes aprender más en el [Curso Profesional de DevOps](#).

28. Creando un Fork, contribuyendo a un repositorio

Forks o Bifurcaciones

Es una característica única de GitHub en la que se crea una copia exacta del estado actual de un repositorio directamente en GitHub, éste repositorio podrá servir como otro origen y se podrá clonar (como cualquier otro repositorio), en pocas palabras, lo podremos utilizar como un git cualquiera.

Un fork es como una bifurcación del repositorio completo, tiene una historia en común, pero de repente se bifurca y pueden variar los cambios, ya que ambos proyectos podrán ser modificados en paralelo y para estar al día un colaborador tendrá que estar actualizando su fork con la información del original.

Al hacer un fork de un proyecto en GitHub, te conviertes en dueño@ del repositorio fork, puedes trabajar en éste con todos los permisos, pero es un repositorio completamente diferente que el original, teniendo alguna historia en común.

Los forks son importantes porque es la manera en la que funciona el open source, ya que, una persona puede no ser colaborador de un proyecto, pero puede contribuir al mismo, haciendo mejor software que pueda ser utilizado por cualquiera.

Al hacer un fork, GitHub sabe que se hizo el fork del proyecto, por lo que se le permite al colaborador hacer pull request desde su repositorio propio.

Trabajando con más de 1 repositorio remoto

Cuando trabajas en un proyecto que existe en **diferentes repositorios remotos** (normalmente a causa de un fork) es muy probable que desees poder **trabajar con ambos repositorios**, para ésto puedes crear un **remoto adicional** desde consola.

```
git remote add <nombre_del_remoto> <url_del_remoto>
git remote upstream https://github.com/freddier/hyperblog
```

Al crear un **remoto adicional** podremos, hacer **pull** desde el nuevo **origen** (en caso de tener permisos podremos hacer fetch y push)

```
git pull <remoto> <rama>
git pull upstream master
```

Éste **pull** nos traerá los **cambios del remoto**, por lo que se estará al día en el proyecto, el flujo de trabajo cambia, en adelante se estará trabajando haciendo **pull desde el upstream** y **push al origen** para pasar a hacer **pull request**.

```
git pull upstream master
git push origin master
```

Aporte de David Behar

29. Haciendo deployment a un servidor (exclusivo para ver en vídeo)

30. Hazme un pull request

Queremos que uses las habilidades ya aprendidas para aplicarlas en esta clase. Haz un *fork* del repositorio de GitHub y realiza las tareas que te indicaremos en esta clase. **Ojo**, debes seguir las reglas e instrucciones que se dieron en el video.

Regla a seguir:

1. Dentro del ID “post” luego de “suscríbete y dale like” agrega otra línea o párrafo con tu nombre o tu nombre de usuario en Platzi.

31. 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**.

<https://git-scm.com/docs/gitignore>

32. Readme.md es una excelente práctica

`README.md` es una excelente práctica en tus proyectos, *md* significa **Markdown**, que es una 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 de Markdown o md <https://pandao.github.io/editor.md/en.html>

33. 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!

<https://pages.github.com/>

34. Git Rebase: reorganizando el trabajo realizado

El comando rebase es ***una mala práctica, nunca se debe usar***, pero para efectos del 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
```

`rebase` reescribe la historia del repositorio, cambia la historia de donde comenzó la rama y solo debe ser usado de manera local.

35. 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 haciendo 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.

Stashed

El `stash` nos sirve para guardar cambios para después, Es una lista de estados que nos guarda algunos cambios que hicimos en Staging para poder cambiar de rama sin perder el trabajo que todavía no guardamos en un commit

Esto es especialmente útil porque hay veces que no se permite cambiar de rama, esto porque tenemos cambios sin guardar, no siempre es un cambio lo suficientemente bueno como para hacer un commit, pero no queremos perder ese código en el que estuvimos trabajando.

El `stash` nos permite cambiar de ramas, hacer cambios, trabajar en otras cosas y, más adelante, retomar el trabajo con los archivos que teníamos en Staging pero que podemos recuperar ya que los guardamos en el Stash.

`git stash`

El comando `git stash` guarda el trabajo actual del Staging en una lista diseñada para ser temporal llamada Stash, para que pueda ser recuperado en el futuro.

Para agregar los cambios al stash se utiliza el comando:

```
git stash
```

Podemos poner un mensaje en el stash, para así diferenciarlos en `git stash list` por si tenemos varios elementos en el stash. Esto con:

```
git stash save "mensaje identificador del elemento del stashed"
```

Obtener elementos del stash

El stashed se comporta como una [Stack](#) de datos comportándose de manera tipo [LIFO](#) (del inglés *Last In, First Out*, «último en entrar, primero en salir»), así podemos acceder al método `pop`.

El método **pop** recuperará y sacará de la lista el **último estado del stashed** y lo insertará en el **staging area**, por lo que es importante saber en qué branch te encuentras para poder recuperarlo, ya que el stash será **agnóstico a la rama o estado en el que te encuentres**, siempre recuperará los cambios que hiciste en el lugar que lo llamas.

Para recuperar los últimos cambios desde el stash a tu staging area utiliza el comando:

```
git stash pop
```

Para aplicar los cambios de un stash específico y eliminarlo del stash:

```
git stash pop stash@{<num_stash>}
```

Para retomar los cambios de una posición específica del Stash puedes utilizar el comando:

```
git stash apply stash@{<num_stash>}
```

Donde el `<num_stash>` lo obtienes desde el `git stash list`

Listado de elementos en el stash

Para ver la lista de cambios guardados en Stash y así poder recuperarlos o hacer algo con ellos podemos utilizar el comando:

```
git stash list
```

Retomar los cambios de una posición específica del Stash || Aplica los cambios de un stash específico

Crear una rama con el stash

Para crear una rama y aplicar el stash mas reciente podemos utilizar el comando

```
git stash branch <nombre_de_la_rama>
```

Si deseas crear una rama y aplicar un stash específico (obtenido desde `git stash list`) puedes utilizar el comando:

```
git stash branch nombre_de_rama stash@{<num_stash>}
```

Al utilizar estos comandos **crearás una rama** con el nombre `<nombre_de_la_rama>`, te pasarás a ella y tendrás el **stash especificado** en tu **staging area**.

Eliminar elementos del stash

Para eliminar los cambios más recientes dentro del stash (el elemento 0), podemos utilizar el comando:

```
git stash drop
```

Pero si en cambio conoces el `índice` del stash que quieres borrar (mediante `git stash list`) puedes utilizar el comando:

```
git stash drop stash@{<num_stash>}
```

Donde el `<num_stash>` es el `índice` del cambio guardado.

Si en cambio deseas eliminar todos los elementos del stash, puedes utilizar:

```
git stash clear
```

Consideraciones

- El cambio más reciente (al crear un stash) **SIEMPRE** recibe el valor 0 y los que estaban antes aumentan su valor.
- Al crear un stash tomará los archivos que han sido modificados y eliminados. Para que tome un archivo creado es necesario agregarlo al Staging Area con `git add [nombre_archivo]` con la intención de que git tenga un seguimiento de ese archivo, o también utilizando el comando `git stash -u` (que guardará en el stash los archivos que no estén en el staging).
- Al aplicar un stash este no se elimina, es buena práctica eliminarlo.

Aporte de David Behar

36. 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 deberían 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`

El parametro `-d` ayuda con el borrado de carpetas untracked.

37. Git cherry-pick: traer commits viejos al head de un branch

Existe un mundo 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**.

38. 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 los agregará al commit anterior.

39. 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.

Git nunca olvida, `git reflog`

Git guarda todos los cambios aunque decidas borrarlos, al borrar un cambio lo que estás haciendo sólo es actualizar la punta del branch, para gestionar éstas puntas existe un mecanismo llamado registros de referencia o *reflogs*.

.

La gestión de estos cambios es mediante los hash'es de referencia (o *ref*) que son apuntadores a los commits.

.

Los `recoges` registran cuándo se actualizaron las referencias de Git en el repositorio local (sólo en el local), por lo que si deseas ver cómo has modificado la historia puedes utilizar el comando:

```
git reflog
```

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

- `git checkout` Puedes moverte sin realizar ningún cambio al commit exacto de la `ref`

```
git checkout eff544f
```

- `git reset`: Hará que el último commit sea el pasado por la `ref`, usar este comando sólo si sabes exactamente qué estás haciendo

```
git reset --hard eff544f # Perderá todo lo que se encuentra en staging y en el Working directory y se moverá el head al commit eff544f
```

```
git reset --soft eff544f # Te recuperará todos los cambios que tengas diferentes al commit eff544f, los agregará al staging area y moverá el head al commit eff544f
```

- `git merge`: Puedes hacer merge de un commit en específico, funciona igual que con una branch, pero te hace el merge del estado específico del commit mandado

```
git checkout master
```

```
git merge eff544f # Fusionará en un nuevo commit la historia de master con el momento específico en el que vive eff544f
```

Aporte de David Behar

40. 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>".`

Ejemplos

- `git grep color -->` use la palabra color
- `git grep la -->` donde use la palabra la
- `git grep -n color-->` en que lineas use la palabra color
- `git grep -n platzi -->` en que lineas use la palabra platzi
- `git grep -c la -->` cuantas veces use la palabra la
- `git grep -c paltzi -->` cuantas veces use la palabra platzi
- `git grep -c "<p>"-->` cuantas veces use la etiqueta <p>
- `git log-S "cabecera" -->` cuantas veces use la palabra cabecera en todos los commits.
- `grep-->` para los archivos
- `log -->` para los commits

Aporte de ehuacachi

41. Comandos y recursos colaborativos en Git y GitHub

- **git shortlog -sn** = muestra cuantos commit han hecho cada miembro del equipo.
- **git shortlog -sn --all** = muestra cuantos commit han hecho cada miembro del equipo hasta los que han sido eliminado
- **git shortlog -sn --all --no-merge** = muestra cuantos commit han hecho cada miembro quitando los eliminados sin los merges
- **git blame ARCHIVO** = muestra quien hizo cada cosa linea por linea
- **git COMANDO --help** = muestra como funciona el comando.
- **git blame ARCHIVO -Llinea_inicial,linea_final=** muestra quien hizo cada cosa linea por linea indicándole desde que linea ver ejemplo -L35,50
- ****git branch -r ****= se muestran todas las ramas remotas
- **git branch -a** = se muestran todas las ramas tanto locales como remotas
- **git shortlog -sn --all --no-merges=** muestra quién ha efectuado cambios en que archivos sin considerar los merges