CURSO PROFESIONAL DE GIT Y GITHUB

INTRODUCCIÓN A GIT

¿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, únicamente las guarda en algo que conocemos como “Staging Area” (área de montaje o ensayo).
- 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.

Comandos básicos de git

- git init: inicializa un repositorio de GIT en la carpeta donde se ejecute el comando.
- git add: añade los archivos especificados al área de preparación (staging).
- git commit -m "commit description": confirma los archivos que se encuentran en el área de preparación y los agrega al repositorio.
- git commit -am "commit description": añade al staging area y hace un commit mediante un solo comando. (No funciona con archivos nuevos)
- git status: ofrece una descripción del estado de los archivos (untracked, ready to commit, nothing to commit).
- git rm (. -r, filename) (–cached): remueve los archivos del index.
- git config --global user.email tu@email.com: configura un email.
- git config --global user.name : configura un nombre.
- git config --list: lista las configuraciones.

Analizar cambios en los archivos de un proyecto Git

- git log: lista de manera descendente los commits realizados.
- git log --stat: además de listar los commits, muestra la cantidad de bytes añadidos y eliminados en cada uno de los archivos modificados.
- git log --all --graph --decorate --oneline: muestra de manera comprimida toda la historia del repositorio de manera gráfica y embellecida.
- git show filename: permite ver la historia de los cambios en un archivo.
- git diff : compara diferencias entre en cambios confirmados.

Volver en el tiempo con branches y checkout

- git reset --soft/hard: regresa al commit especificado, eliminando todos los cambios que se hicieron después de ese commit.
- git checkout : permite regresar al estado en el cual se realizó un commit o branch especificado, pero no elimina lo que está en el staging area.
- git checkout – : deshacer cambios en un archivo en estado modified (que ni fue agregado a staging)

Git rm y git reset

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.

git rm no puede usarse por sí solo, así nomás. Se debe utilizar uno de los flags para indicar a Git cómo eliminar los archivos que ya no se necesitan 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 aplicar comandos más avanzados).

git reset

Con git reset volvemos al pasado sin la posibilidad de volver al futuro. Borramos la historia y la debemos sobreescribir.

- git reset --soft: Vuelve el branch al estado del commit especificado, manteniendo los archivos en el directorio de trabajo y lo que haya en staging considerando todo como nuevos cambios. Así podemos aplicar las últimas actualizaciones a un nuevo commit.
- git reset --hard: Borra absolutamente todo. Toda la información de los commits y del área de staging se borra del historial.

- git reset HEAD: No borra los archivos ni sus modificaciones, solo los saca del área de staging, de forma que los últimos cambios de estos archivos no se envíen al último commit. Si se cambia de opinión se los puede incluir nuevamente con git add.

Ramas o branches en git

Al crear una nueva rama se copia el último commit en esta nueva rama. Todos los cambios hechos en esta rama no se reflejarán en la rama master hasta que hagamos un merge.

- git branch : crea una nueva rama.
- git checkout : se mueve a la rama especificada.
- git merge : fusiona la rama actual con la rama especificada y produce un nuevo commit de esta fusión.
- git branch: lista las ramas generadas.

¿QUÉ ES GIT?

Git es un sistema de control de versiones distribuido que te permite registrar los cambios que haces en tus archivos y volver a versiones anteriores si algo sale mal. Fue diseñado por Linus Torvalds para garantizar la eficiencia y confiabilidad del mantenimiento de versiones de aplicaciones que tienen un gran número de archivos de código fuente.

- Git está optimizado para guardar cambios de forma incremental.
- Permite contar con un historial, regresar a una versión anterior y agregar funcionalidades.
- Lleva un registro de los cambios que otras personas realicen en los archivos.

Git fue diseñado para operar en un entorno Linux. Actualmente, es multiplataforma, es decir, es compatible con Linux, MacOS y Windows. En la máquina local se encuentra Git, se utiliza bajo la terminal o línea de comandos y tiene comandos como *merge, pull, add, commit* y *rebase*, entre otros.

Para qué proyectos sirve Git

Con Git se obtiene una mayor eficiencia usando archivos de texto plano, ya que con archivos binarios no puede guardar solo los cambios, sino que debe volver a grabar el archivo completo ante cada modificación, por mínima que sea, lo que hace que incremente demasiado el tamaño del repositorio.

"Guardar archivos binarios en el repositorio de Git no es una buena práctica, únicamente deberían guardarse archivos pequeños (como logos) que no sufran casi modificaciones durante la vida del proyecto. Los binarios deben guardarse en un CDN".

Cómo usar git stash

Características de Git

Git te ayuda a trabajar de manera más organizada y colaborativa en proyectos de desarrollo de software. Estas son algunas de sus principales características:

Control de versiones

Git almacena la información como un conjunto de archivos. Te permite llevar un registro de los cambios que haces en tus archivos, lo que significa que siempre puedes volver a versiones anteriores si algo sale mal.

Ramificación

Puedes crear ramas en tu proyecto, lo que te permite trabajar en diferentes características o aspectos del mismo sin afectar el trabajo de los demás.

Colaboración

En Git, varias personas pueden trabajar en diferentes aspectos del proyecto al mismo tiempo.

Seguridad

No existen cambios, corrupción en archivos o cualquier alteración sin que Git lo sepa. Git cuenta con 3 estados en los que es posible localizar archivos: Staged, Modified y Committed.

Flexibilidad

Casi todo en Git es local. Es difícil que se necesiten recursos o información externos, basta con los recursos locales con los que cuenta.

Comandos

Git tiene una sintaxis de comandos bastante sencilla y fácil de aprender, lo que lo hace accesible incluso para principiantes en programación.

¿Qué es un sistema de control de versiones?

El SCV o VCS (por sus siglas en inglés) es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas llevar el historial del ciclo de vida de un proyecto, comparar cambios a lo largo del tiempo, ver quién los realizó o revertir el proyecto entero a un estado anterior. Cualquier tipo de archivo que se encuentre en un ordenador puede ponerse bajo control de versiones.

¿En qué se diferencia de Github?

Github es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. Se emplea principalmente para la creación de código fuente de programas de computadora.

Puede considerarse a Github como la red social de código para los programadores y en muchos casos es visto como un *curriculum vitae*, pues aquí se guarda el portafolio de proyectos de programación.

Características de Github

- GitHub permite alojar proyectos en repositorios de forma gratuita y pública, pero tiene una forma de pago para privados.
- Puedes compartir fácilmente tus proyectos.
- Permite colaborar para mejorar los proyectos de otros y a otros mejorar o aportar a los tuyos.
- Ayuda a reducir significativamente los errores humanos, a tener un mejor mantenimiento de distintos entornos y a detectar fallos de una forma más rápida y eficiente.
- Es la opción perfecta para poder trabajar en equipo en un mismo proyecto.

- Ofrece todas las ventajas del sistema de control de versiones Git, pero también tiene otras herramientas que ayudan a tener un mejor control de los proyectos.

INSTALANDO GITBASH EN WINDOWS

Al implementar **Git Bash**, podrás tener una terminal de Git en el sistema operativo Windows. Esto te ayudará a acceder a tus repositorios al conectarte desde GitHub. Descubramos más acerca de esta integración.

¿Qué es Git bash?

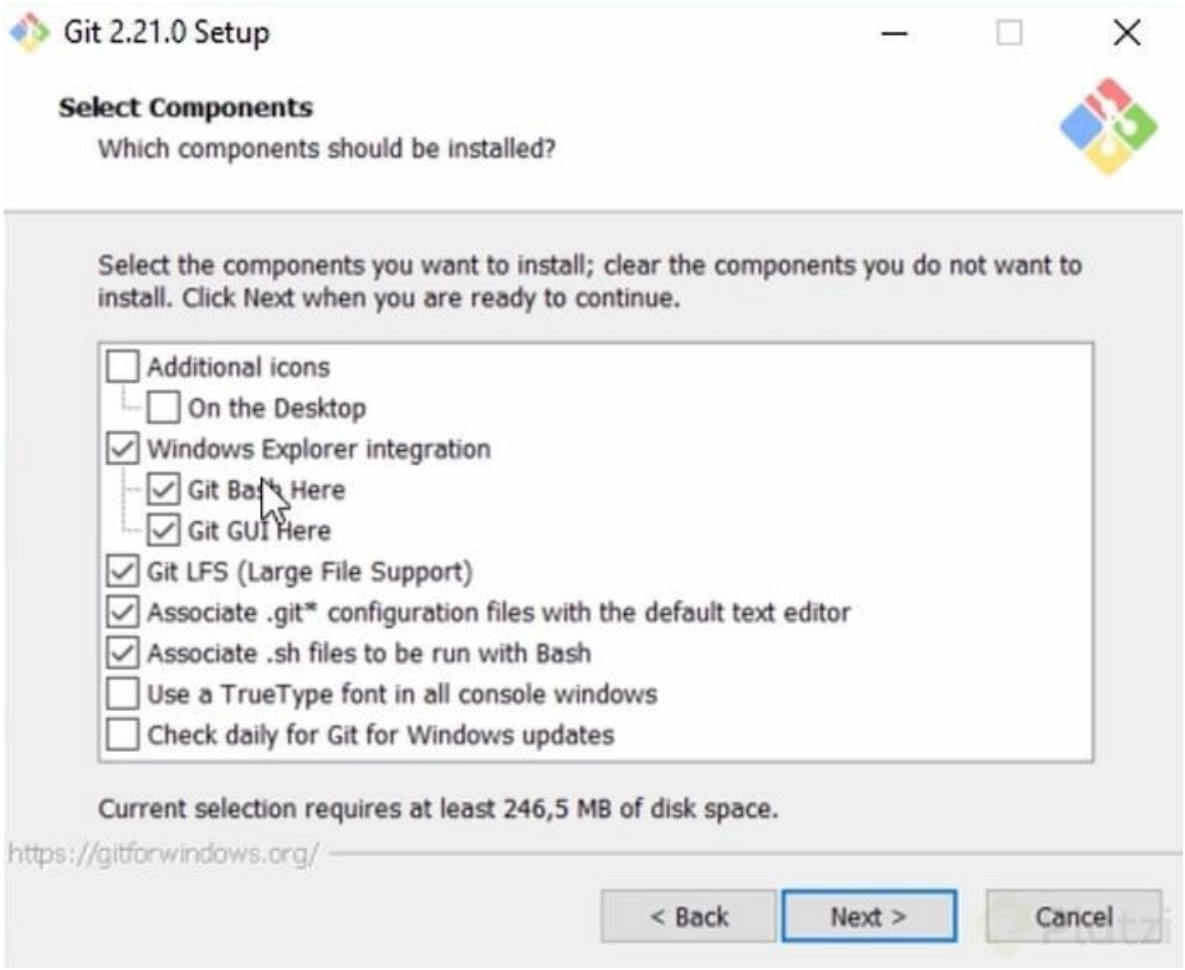
Git Bash es una aplicación en línea que une las utilidades de Git y Bash en un sistema operativo de Windows. Mejora la experiencia de la línea de comandos y se comporta como una especie de intérprete para un sistema operativo Unix, pero adaptado para funcionar en Windows.

Además, el programa permite a los usuarios acceder a las herramientas de Git,y realizar tareas de control de versiones a través de comandos de texto en lugar de una interfaz gráfica de usuario. Incluye otros comandos de Unix y herramientas como ssh , scp y rsync

Instalación de Git bash paso a paso

Para instalar este sistema de control de versiones en Windows, simplemente dirígete al repositorio de descarga.

1. Una vez descargado, ejecútalo como cualquier otra aplicación de Windows.
2. Al iniciar el instalador, asegúrate de marcar la opción de instalar Git Bash en el computador. Esto nos permitirá correr comandos de Linux en la consola sin problemas para trabajar con Git.



¿Cómo utilizar Git Bash?

Git bash funciona al igual que el Bash estándar en UNIX. Una vez instalada solo tienes que abrirla en tu ordenador y luego podrás inicializar un nuevo repositorio GIT.

1. Navega hasta el directorio que deseas usar como repositorio y ejecuta el comando “git init”. Esto creará un nuevo repositorio Git en el directorio actual.
2. Abre Git Bash en tu ordenador. Esto se puede hacer de varias maneras, como hacer clic con el botón derecho del ratón en una carpeta y seleccionar “Git Bash Here” en el menú contextual.
3. Inicializa un nuevo repositorio Git.

¿Cómo navegar por las carpetas en Bash?

El comando de Bash pwd se utiliza para imprimir el directorio de trabajo actual, que es la carpeta o ruta en la que reside la sesión actual de Bash. Es equivalente a ejecutar cd en un terminal DOS de Windows.

Por otro lado, el comando de Bash ls se emplea para enumerar el contenido del directorio de trabajo actual. Este comando es equivalente a ejecutar el comando DIR en un terminal de host de consola de Windows.

Es importante destacar que tanto el host de la consola de Bash como el de Windows tienen un comando cd, que se emplea para cambiar de directorio. Si se invoca cd con el nombre de un directorio adjunto, se cambiará el directorio de trabajo actual de las sesiones del terminal al argumento del directorio especificado.

Comandos complementarios de Git Bash

Windows y Linux difieren en comandos, en la forma en que registran el "enter" y en muchas otras cosas. Cuando instales Git Bash en Windows, debes elegir entre trabajar con la forma de Windows o la forma de UNIX (Linux y Mac). Algunos comandos que pueden ser familiares son:

- git checkout
- git commit
- git clone
- git pull
- git push

En entornos de desarrollo profesionales, es común que las personas utilicen sistemas operativos diferentes. Si todos podemos usar los mismos comandos, el trabajo resultará más fácil.

Los comandos de UNIX son los más frecuentes en equipos de desarrollo, por lo que, a menos que trabajes con tecnologías nativas de Microsoft (por ejemplo, .NET), es recomendable elegir la opción de la terminal tipo UNIX para una mejor compatibilidad con todo el equipo.

Repasa: ¿Qué es Git?

EDITORES DE CÓDIGO, ARCHIVOS BINARIOS Y DE TEXTO PLANO

Un editor de código o **IDE** 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 es obligatorio usar alguno de estos para programar. Conoce más a fondo sobre qué es un IDE.

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.

Conceptos importantes de Git

- Bug: Error en el código
- Repository: Donde se almacena todo el proyecto, el cual puede vivir tanto en local como en remoto. El repositorio guarda un historial de versiones y, más importante, de la relación de cada versión con la anterior para que pueda hacerse el árbol de versiones con las diferentes ramas.
- Fork: Si en algún momento queremos contribuir al proyecto de otra persona, o si queremos utilizar el proyecto de otro como el punto de partida del nuestro. Esto se conoce como "fork".
- Clone: Una vez se decide hacer un fork , hasta ese momento sólo existe en GitHub. Para poder trabajar en el proyecto, toca clonar el repositorio elegido al computador personal.
- Branch: Es una bifurcación del proyecto que se está realizando para anexar una nueva funcionalidad o corregir un bug.
- Master: Rama donde se almacena la última versión estable del proyecto que se está realizando. La rama master es la que está en producción en cada momento (o casi) y debería estar libre de bugs. Así, si esta rama está en producción, sirve como referente para hacer nuevas funcionalidades y/o arreglar bugs de última hora.
- Commit: consiste en subir cosas a la versión local del repositorio. De esta manera se puede trabajar en la rama de forma local sin tener que modificar ninguna versión en remoto ni tener que tener la última versión remota, cosa muy útil en grandes desarrollos trabajados por varias personas.
- Push: Consiste en enviar todo lo que se ha confirmado con un commit al repositorio remoto. Aquí es donde se une nuestro trabajo con el de los demás.
- Checkout: Acción de descargarse una rama del repositorio GIT local (sí, GIT tiene su propio repositorio en local para poder ir haciendo commits) o remoto.
- Fetch: Actualiza el repositorio local bajando datos del repositorio remoto al repositorio local sin actualizarlo, es decir, se guarda una copia del repositorio remoto en el local.

- Merge: La acción de merge es la continuación natural del fetch. El merge permite unir la copia del repositorio remoto con tu repositorio local, mezclando los diferentes códigos.
- Pull: Consiste en la unión del fetch y del merge, esto es, recoge la información del repositorio remoto y luego mezcla el trabajo en local con esta.
- Diff: Se utiliza para mostrar los cambios entre dos versiones del mismo archivo.

INTRODUCCIÓN A LA TERMINAL Y LÍNEA DE COMANDOS

La línea de comandos nos permite interactuar con nuestro computador sin necesidad de utilizar una interfaz gráfica. Sin embargo, los computadores emplean distintos sistemas de archivos y manejan diferentes comandos, dependiendo del sistema operativo que utilicen.

Aprende más sobre: qué es un comando.

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 tiene 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).

Comandos Principales

- pwd nos muestra el path o ruta de la carpeta en donde nos encontramos ubicados
- cd me permite acceder (entrar) a una carpeta en un nivel o varios niveles
- cd .. me permite salir de una carpeta en un nivel o varios niveles **OJO los dos puntos deben ser separados por un espacio del comando cd**
- ls me muestra los archivos que contiene una carpeta, puede ser la ubicación actual o una ruta especifica, no muestra los archivos ocultos
- ls -a me muestra los archivos que contiene una carpeta, puede ser la ubicación actual o una ruta especifica, incluyendo los archivos ocultos
- ls -l me lista los archivos que contiene una carpeta con sus atributos, puede ser la ubicación actual o una ruta especifica, no muestra los archivos ocultos
- ls -la me lista los archivos que contiene una carpeta con sus atributos, puede ser la ubicación actual o una ruta especifica, incluyendo los archivos ocultos
- clear limpiar la consola o terminal, o un shorcut crtl + L
- mkdir <nombre carpeta> nos permite crear una carpeta
- touch <nombre del archivo> nos permite crear un archivo
- cat <nombre del archivo> me permite visualizar el contenido del un archivo y lo muestra en el terminal
- history nos muestra un historial de los comandos que hemos utilizado
- rm <nombre del archivo> me permite borrar un archivo

OJO en Windows el terminal no es case sensitive (Sensible las mayusculas), con Linux,y UNIX si son case sensitive

Trabajando con archivos

ls – listar contenido de un directorio
ls -al – listado con atributos y archivos ocultos
cd newdir/ - moverse al directorio *newdir*
cd – moverse al directorio home
pwd – mostrar la ruta actual
rm *file* – borrar el archivo *file*
rm -r *dir* – borrar el directorio *dir*
rm -f *file* – borrar *file* sin emitir mensajes de error
rm -rf *dir* – igual que el anterior pero con el directorio dir [**]
cp *file1 file2* – copiar *file1* en *file2*
cp -r *dir1 dir2* – copiar el dir1 en dir2 (si no existe se crea)
mv *file1 file2* – renombra *file1* como *file2*. Si *file2* es un directorio lo mueve dentro del mismo.
ln -s *file link* – crea un enlace simbólico de *link* hacia *file*.
touch *file* – crea o actualiza *file*
cat > *file* – redirecciona la entrada estándar a *file*
more *file* – muestra el contenido de *file*
head *file* – muestra las 10 primeras filas de *file*
tail *file* – muestra las 10 últimas filas de *file*
tail -f *file* – muestra las 10 últimas filas de *file* a medida que va creciendo.

Gestión de procesos

ps – muestra los procesos activos del usuario
top – muestra todos los procesos activos
kill *pid* – mata el proceso con id *pid*
killall *proc* – mata todos los procesos *proc* [**]
bg – lista los procesos parados o en segund plano
fg – lleva proceso más reciente a primer plano
fg *n* – lleva proceso *n* a primer plano

File Permissions

chmod *octal file* – establece en file los permisos especificados en octal (usuario, grupo y otros)

- 4 – lectura (r)
- 2 – escritura (w)
- 1 – ejecución (x)

Ejemplos:
chmod 777 – lectura/escritura/ejecución para todos.
chmod 755 – rwx para el propietario, rx para su grupo y otros

SSH

ssh *user@host* – conectar a *host* como *user*
ssh -p *port user@host* – conectar a *host* por el puerto *port* como *user*
ssh-copy-id *user@host* – añadir clave de *user* a *host* para autenticarte

Búsqueda

grep *pattern files* – buscar patrón *pattern* en *files*
grep -r *pattern dir* – buscar recursivamente patrón *pattern* en *dir*
command | grep *pattern* – buscar patrón *pattern* en la salida de *command*
locate file – Busca instancias de *file*

Información del sistema

date – Consulta la fecha y hora actual
cal – Muestra el calendario del mes actual
uptime – tiempo que lleva encendida la máquina
w – muestra usuarios conectados a la máquina
whoami – nombre de mi usuario
finger *user* – muestra información sobre *user*
uname -a – información sobre el núcleo
cat /proc/cpuinfo – información sobre la cpu
cat /proc/meminfo – información sobre la memoria
man *command* – páginas de manual sobre *command*
df – espacio libre en los discos
du – espacio usado por los directorios
free – uso de memorio y swap
whereis app – localiza el binario, fuente y página de manual de *app*
which app – localiza el comando *app*

Compresión

tar cf *file.tar files* – empaqueta *files* en un fichero *files.tar*
tar xf *file.tar* – extrae el contenido de *file.tar*
tar czf *file.tar.gz files* – empaqueta y comprime (gzip) files en files.tar.gz
tar xzf *file.tar.gz* – extract y descomprime usando Gzip
tar cjf *file.tar.bz2* – empaqueta y comprime (bzip2) files en files.tar.bz2
tar xjf *file.tar.bz2* – extract y descomprime usando Bzip2
gzip *file* – comprime *file* y lo renombra como *file.gz*
gzip -d *file.gz* – descomprime *file.gz* a *file*

Redes

ping *host* – hace ping a *host* y muestra los datos
whois *domain* – información del dominio *domain*
dig *domain* – configuración DNS de *domain*
dig -x *host* – DNS inverso de *host*
wget *file* – descarga *file*
wget -c *file* – continua una descarga parada

Instalación

Instalar desde los fuentes:
./configure
make
make install
dpkg -i *pkg.deb* – instalar paquete DEB
rpm -Uvh *pkg.rpm* – isntalar paquete RPM

Combinaciones de teclas

Ctrl+C – Interrumpe el comando activo
Ctrl+Z – Suspende el comando activo, con **fg** se reanuda y con **bg** se lleva a segundo plano
Ctrl+D – abandona sesión actual, similar a **exit**
Ctrl+W – borra una palabra en la línea actual
Ctrl+U – borra toda la línea
!! - repite el último comando
exit – abandona la sesión actual

[**] usar con mucho cuidado

COMANDOS BÁSICOS EN GIT

CREA UN REPOSITORIO DE GIT Y HAZ TU PRIMER COMMIT

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).

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.

Comandos para iniciar tu repositorio con Git

- git init: para inicializar el repositorio git y el staged
- git add nombre_del_archivo.txt: enviar el archivo al staged
- git status: ver el estado, si se requiere agregar al starget o si se requiere commit
- git conf: para ver las posibles configuraciones
- git conf --list: para ver la lista de configuraciones hechas

- git conf --list --show-origin: para mostrar las configuraciones y sus rutas
- git rm --cached nombre_del_archivo.txt: para eliminar el archivo del staged(ram)
- git rm nombre_del_archivo.txt: para eliminar del repositorio

Si por algún motivo te equivocaste en el nombre o email que configuraste al principio, lo puedes modificar de la siguiente manera:
git config --global --replace-all user.name “Aquí va tu nombre modificado”
O si lo deseas eliminar y añadir uno nuevo
git config --global --unset-all user.name :Elimina el nombre del usuario
git config --global --add user.name “Aquí va tu nombre”

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.

Cuando 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"
```

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

```
$ git commit -m "TuMensajedeCommit"
```

gid 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

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**.

Comandos para analizar cambios en GIT

- **git init**: inicializar el repositorio
- **git add nombre_de_archivo.extensión**: agregar el archivo al repositorio
- **git commit -m "Mensaje"**: Agregamos los cambios para el repositorio
- **git add**: Agregar los cambios de la carpeta en la que nos encontramos agregar todo
- **git status**: visualizar cambios
- **git log nombre_de_archivos.extensión**: histórico de cambios con detalles
- **git push**: envía a otro repositorio remoto lo que estamos haciendo
- **git pull**: traer repositorio remoto
- **ls**: listado de carpetas en donde me encuentro. Es decir, como emplear dir en windows.
- **pwd**: ubicación actual
- **mkdir**: make directory nueva carpeta
- **touch archivo.extensión**: crear archivo vacío

- **cat archivo.extensión**: muestra el contenido del archivo
- **history**: historial de comandos utilizados durante esa sesión
- **rm archivo.extensión**: Eliminación de archivo
- **comando --help**: ayuda sobre el comando
- **git checkout**: traer cambios realizados
- **git rm --cached archivo.extensión**: se utiliza para devolver el archivo que se tiene en ram. Cuando escribimos git add, lo devuelve a estado natural mientras está en *staging*.
- **git config --list**: muestra la lista de configuración de git
- **git config --list --show-origin**: rutas de acceso a la configuración de git
- **git log archivo.extensión**: muestra la historia del archivo

¿QUÉ ES EL STAGING?

El staging es el lugar donde se guardan temporalmente los cambios, para luego ser llevados definitivamente al repositorio. El repositorio es el lugar donde se guardan todos los registros de los cambios realizados a los archivos.

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.

¿Qué es el área de staging?

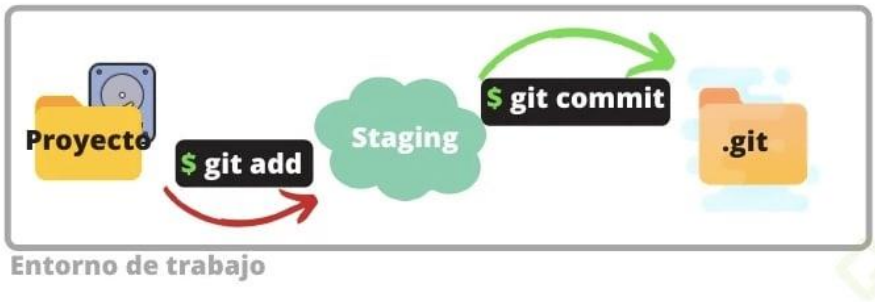
El área de staging se puede ver como un limbo donde nuestros archivos están por ser enviados al repositorio o ser regresados a la carpeta del proyecto.

¿Qué es git init?

git inites el comando que activa git en nuestro proyecto creando un espacio en memoria RAM llamado staging y una carpeta .git.

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; 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).



Cómo funciona el staging y el repositorio: ciclo básico de trabajo en git:

El flujo de trabajo básico en git es algo así:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiéndolos a tu área de preparación (staging).
3. Confirmas los cambios (commit), lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de git.

Veamos a detalle las 3 secciones principales que tiene un proyecto en git.

Working directory

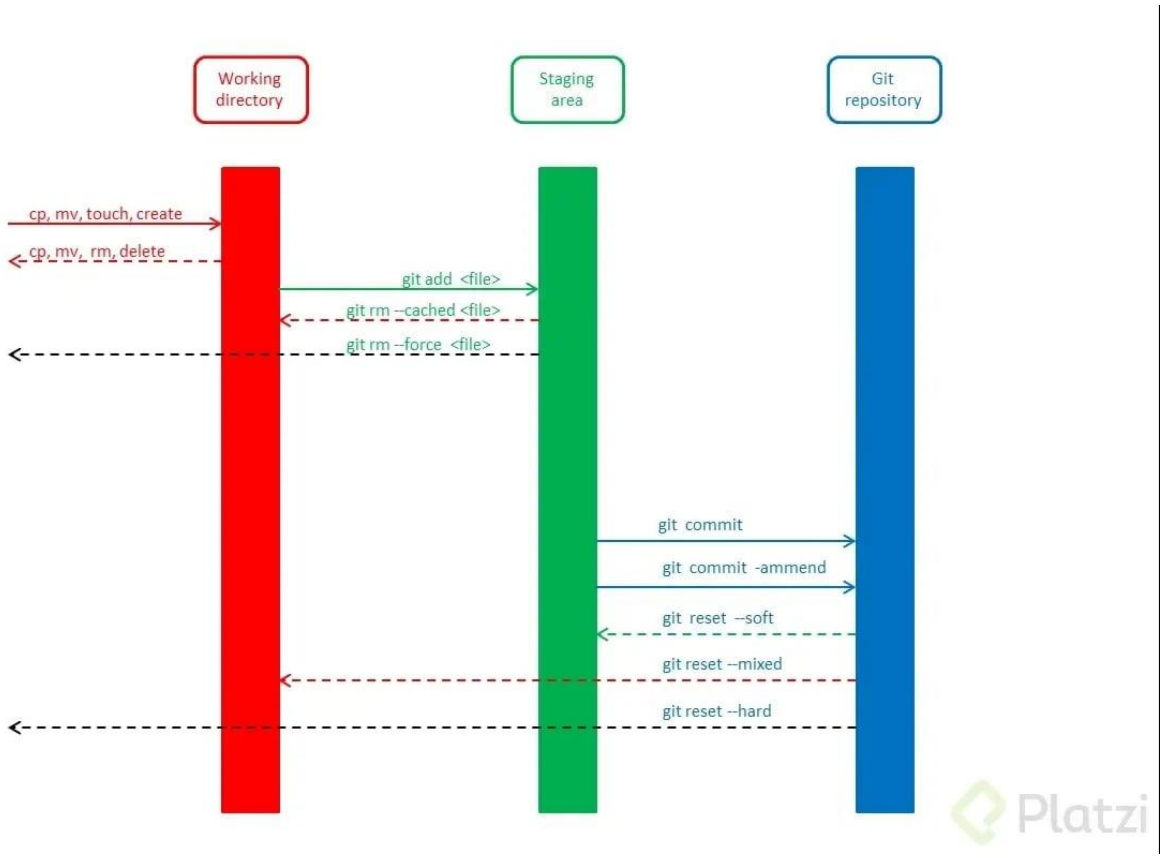
El *working directory* es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de git y se colocan en el disco para que los puedas usar o modificar.

Staging area

Es un área que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice (*index*).

.git directory (repository)

En el *repository* se almacenan los metadatos y la base de datos de los objetos para tu proyecto. Es la parte más importante de git (carpeta .git) y es lo que se copia cuando clonas un repositorio desde otra computadora.



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.

Git stash: guarda cambios temporalmente

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.

Comandos para mover archivos entre los estados de Git

Estos son los comandos más importantes que debes conocer:

Git status

git status nos permite ver el estado de todos nuestros archivos y carpetas.

Git add

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

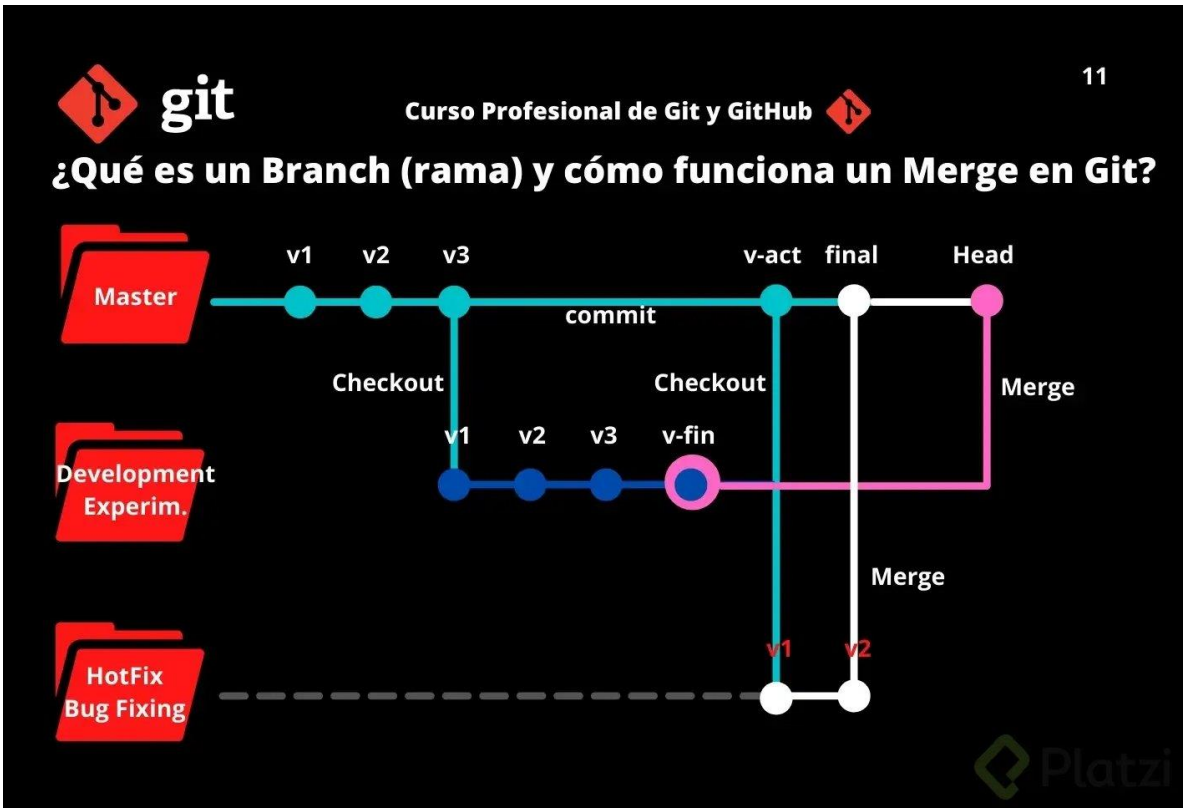
Este comando necesita alguno de los siguientes argumentos para poder ejecutarse correctamente:

- 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).

¿QUÉ ES BRANCH (RAMA) Y CÓMO FUNCIONA UN MERGE EN GIT?

Una **rama o branch** es una versión del código del proyecto sobre el que estás trabajando. Estas ramas ayudan a mantener el orden en el control de versiones y manipular el código de forma segura.

En otras palabras, un branch o rama en Git es una rama que proviene de otra. Imagina un árbol, que tiene una rama gruesa, y otra más fina, en la rama más gruesa tenemos los commits principales y en la rama fina tenemos otros commits que pueden ser de hotfix, devlopment entre otros.



Clases de branches o ramas en Git

Estas son las ramas base de un proyecto en Git:

1. Rama main (Master)

Por defecto, el proyecto se crea en una rama llamada Main (anteriormente conocida como Master). Cada vez que añades código y guardas los cambios, estás haciendo un commit, que es añadir el nuevo código a una rama. Esto genera nuevas versiones de esta rama o branch, hasta llegar a la versión actual de la rama Main.

2. Rama development

Cuando decides hacer experimentos, puedes generar ramas experimentales (usualmente llamadas development), que están basadas en alguna rama main, pero sobre las cuales puedes hacer cambios a tu gusto sin necesidad de afectar directamente al código principal.

3. Rama hotfix

En otros casos, si encuentras un bug o error de código en la rama Main (que afecta al proyecto en producción), tendrás que crear una nueva rama (que usualmente se llaman bug fixing o hot fix) para hacer los arreglos necesarios. Cuando los cambios estén listos, los tendrás que

fusionar con la rama Main para que los cambios sean aplicados. Para esto, se usa un comando llamado *Merge*, que mezcla los cambios de la rama que originaste a la rama Main.

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

Cómo crear un branch o rama en Git

El comando git branch permite crear una rama nueva. Si quieres empezar a trabajar en una nueva función, puedes crear una rama nueva a partir de la rama master con git branch new_branch. Una vez creada, puedes usar git checkout new_branch para cambiar a esa rama.

Recuerda que todas tus versiones salen de la rama principal o Master y de allí puedes tomar una versión específica para crear otra rama de versiones.

Cómo hacer merge

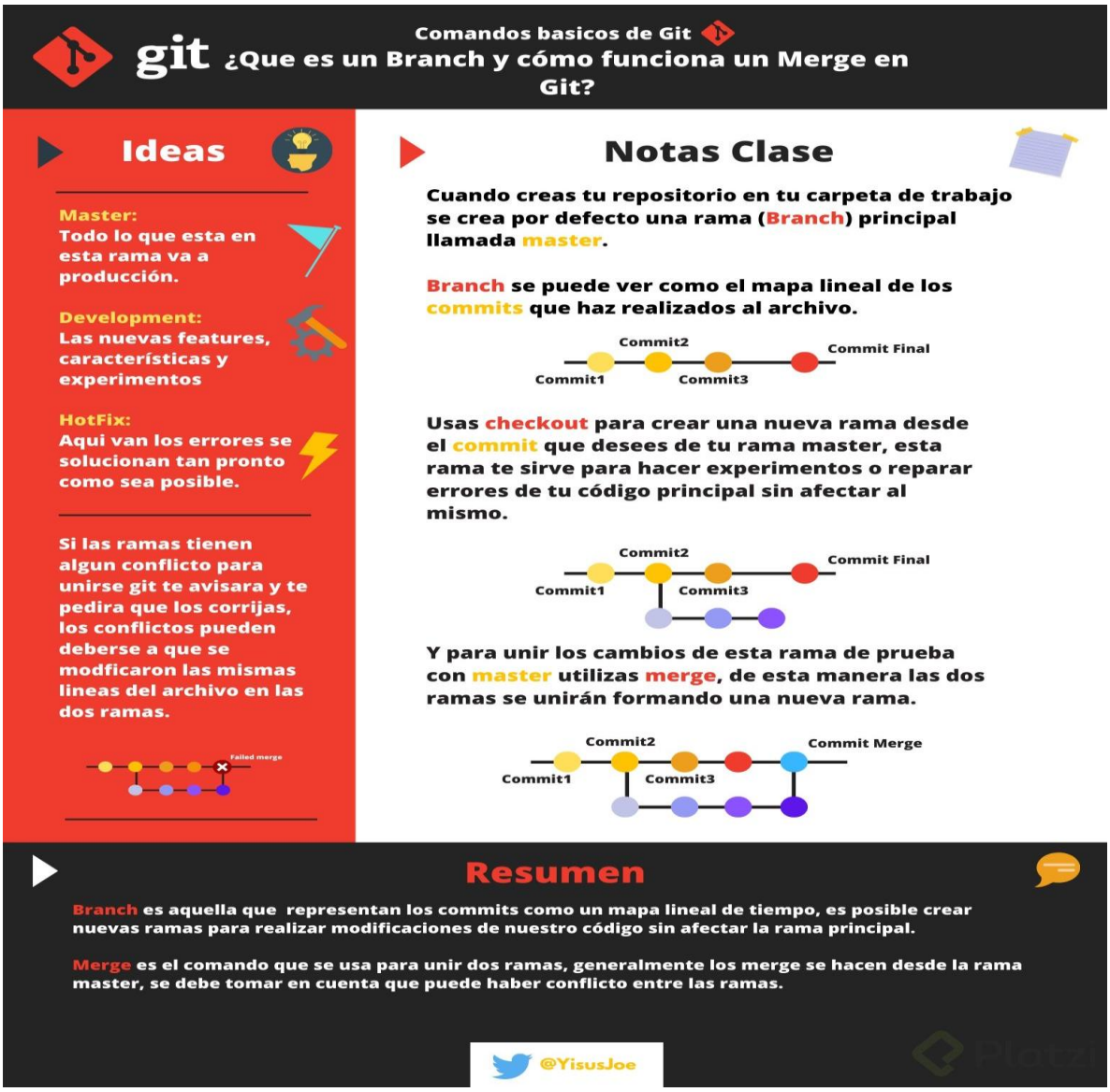
Producir una nueva rama se conoce como **Checkout**. Unir dos ramas lo conocemos como **Merge**.

Cuando haces merge de estas ramas con el código principal, su código se fusiona originando una nueva versión de la rama master (o main) que ya tiene todos los cambios que aplicaste en tus experimentos o arreglos de errores.

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

Descubre qué son los git tags

Solo ten en cuenta que combinar estas ramas (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*.



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.

Repasa qué es branch

Cómo usar Git Reset

Para volver a commits previos, borrando los cambios realizados desde ese commit, podemos utilizar:

- git reset --soft [SHA 1]: elimina los cambios hasta el staging area
- git reset --mixed [SHA 1]: elimina los cambios hasta el working area
- git reset --hard [SHA 1]: regresa hasta el commit del [SHA-1] Donde el SHA-1 es el identificador del commit

Git y GitHub

* git init

Estas en la carpeta en la que están los archivos de tu proyecto.
Abres la terminal y ejecutas git init para indicar que esta es la carpeta central de mis archivos.
Cuando ejecutamos el comando, ademas, creamos:

a. Staging, es el area en la memoria RAM que es donde al principio iremos agregando los cambios.
b. Repository (repositorio/master): Es el directorio que conocemos como /.git/ que es el lugar donde al final del proyecto quedarán todos los cambios.

* git config –global user.name "MiNombre"

-Configuramos el git a nuestro nombre.

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

-Configuramos git con nuestro correo.

* git config --list

-Nos muestra la configuración por defecto de nuestro git.

* git add archivo.txt

- Agregamos el archivo al Staging, en este punto el archivo espera a que lo agreguemos al repositorio.
Nota: antes de usar este comando, el archivo lo llamamos como un archivo untracked", luego de usar el comando el archivo lo podemos llamar un archivo Tracked, cuando está ya en el Staging

- git add .
- Agregamos TODOS los archivos dentro de nuestra carpeta al Staging.

* git rm –cached archivo.txt

-Eliminamos el archivo del Staging area y lo volvemos Untracked.

* git commit -m "mensaje"

Luego de escribir un mensaje descriptivo, mandamos el archivo al repositorio/master.
Nota: Cada commit es una nueva versión con cambios de tu archivo.

* git status

-Nos muestra cual es el status de nuestro proyecto.

* git log
-Nos muestra el historial de commits hechos desde el mas reciente hasta el mas viejo.

* git log --stat
-Nos muestra el hislos cambios especificos de cada commit desde el mas reciente hasta el mas viejo.

* git show archivo.txt
-Muestra los cambios hechos al archivo en el último commit.

* git diff
-Nos muestra las diferencias entre los cambios que estan en Staging y los que aun no se an añadido.

* git diff #commit1 #commit2
-Nos muestra las diferencias entre las 2 versiones de los commits.

---

* git reset #commit --soft
-Volvemos nuestro archivo a la version del commit indicada, pero sin borrar los commits usados anteriormente.

* git reset #commit --hard
-Volvemos nuestro archivo a la version del commit indicada, BORRANDO TODOS los commits usados anteriores a la version del commit indicada.

* git checkout #commit archivo.txt
-Volvemos nuestro archivo a la version del commit indicada, sin cambiar nada en el Staging.
    - git checkout master archivo.txt
    -Volvemos nuestro archivo a la version que teniamos antes de hacer el primer checkout, sin cambiar nada en el Staging.

Comandos basicos de Git

Volver en el tiempo en nuestro repositorio utilizando reset y checkout

Ideas

Aunque git reset --hard se considera el más peligroso por borrar absolutamente todo, realmente es el más utilizado por los programadores.

Al utilizar git checkout te da la posibilidad de crear una rama nueva para guardar los cambios que hagas utilizando.

```
$ git switch -c <new-branch-name>
```

De esta manera puedes experimentar con versiones anteriores de tu programa actual sin afectarlo.

Al usar checkout para ir a un commit entras en un estado llamado detached HEAD.

Notas Clase

Si quieres "volver en el tiempo" y regresar a una versión anterior de tu archivo puedes usar git reset.

Hay dos formas de usar git reset con el argumento --hard que lo que hace es borrar toda la información de registro que tengamos incluso lo que este en el área de staging.

Y el argumento --soft igual borra los cambios que has hecho a tu archivo pero lo que tenias en staging se mantiene ahí dándote la posibilidad de de aplicar los cambios.

Pero si lo que buscas es volver a cualquier versión anterior sin BORRAR el historial del archivo, utiliza el comando.

```
$ git checkout + ID_Commit
```

De esta manera vuelves al commit que desees.

En este estado puedes ver el archivo, hacer cambios experimentales e incluso confirmarlos y al volver al commit presente los cambios no tendrán efecto.

Resumen

Si quieres "volver en el tiempo" y ver los cambios que has realizado en tus archivos utiliza git checkout, este comando te deja ver el archivo, modificarlo y si quieres guardar los cambios. Estando en el commit seleccionado te da la posibilidad de crear una rama nueva para no interferir con la rama principal, utilizando el comando git switch -c <new-branch-name>.

También puedes usar git reset con los atributos --hard o --soft para "regresar en el tiempo" pero borrando los commits posteriores. Con --hard borras todo incluso lo que esta en el área de staging y con --soft también borra todo pero no borra los cambios que dejaste en staging.

@YisusJoe

GIT RESET VS. GIT RM

Git reset vs. Git rm

13/43

Ir a la nueva versión

LECTURA

Los comandos git reset y git rm tienen utilidades muy diferentes, pero pueden confundirse fácilmente.

Git reset

El comando git reset es una herramienta poderosa que te permite deshacer o revertir cambios en tu repositorio de Git. Lo puedes ejecutar de tres maneras diferentes, con las líneas de comando --soft, --mixed y --hard.

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.

Tres árboles en Git

Para entender lo anterior, recordemos que los "tres árboles" de Git son estructuras de datos basadas en nodos y punteros que Git utiliza para hacer seguimiento a un cronograma de ediciones, aunque no sean estructuras en forma de árbol en el sentido tradicional.

La mejor forma de entender estos mecanismos es creando un conjunto de cambios en un repositorio y siguiéndolos a través de los tres árboles. Averigüémoslo.

$ mkdir git_reset_test

$ cd git_reset_test/

$ git init .

Initialized empty Git repository in /git_reset_test/.git/

$ touch reset_lifecycle_file

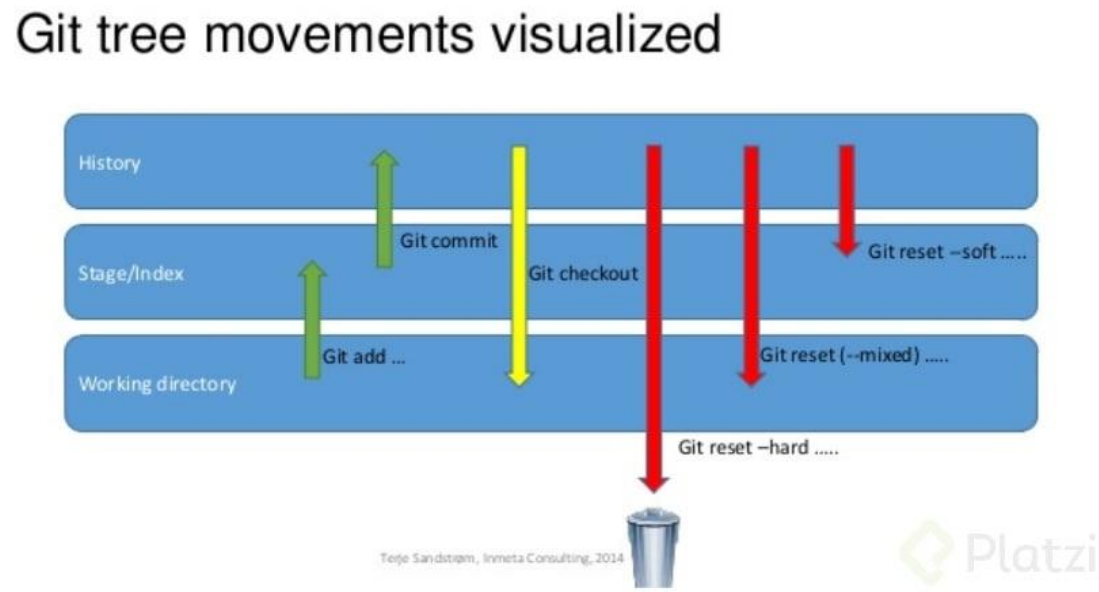
$ git add reset_lifecycle_file

$ git **commit** -m"initial commit"

[**main** (root-**commit**) d386d86] **initial commit**

1 **file changed**, 0 insertions(+), 0 deletions(-)

create mode 100644 reset_lifecycle_file



¿Cómo funciona Git Reset en tu flujo de trabajo?

git reset permite moverte entre diferentes commits para deshacer o rehacer cambios. Git guarda todos lo nuevo del repositorio como commits, que son instantáneas del estado del código en un momento dado y existen variaciones de este comando.

Variaciones de Git Reset

- git reset --soft: Borra el historial y los registros de Git de commits anteriores, pero guarda los cambios en Staging para aplicar las últimas actualizaciones a un nuevo commit.
- git reset --hard: Deshace todo, absolutamente todo. Toda la información de los commits y del área de staging se elimina del historial.
- git reset --mixed: Borra todo, exactamente todo. Toda la información de los commits y del área de staging se elimina del historial.
- git reset HEAD: El comando git reset saca archivos del área de staging sin borrarlos ni realizar otras acciones. Esto impide que los últimos cambios en estos archivos se envíen al último commit.

Podemos incluirlos de nuevo en staging con git add si cambiamos de opinión.

Ten en cuenta que, si deshaces commits en un repositorio compartido en GitHub, estarás cambiando su historia y esto puede causar problemas de sincronización con otros colaboradores.

¿Qué es git reset HEAD?

git reset HEAD es un comando que te permite revertir los cambios que ya habías preparado para subir, y moverlos de vuelta a tu proyecto. Con este comando puedes cancelar los cambios que ya habías agregado, para que puedas revisarlos, modificarlos o deshacerlos antes de confirmarlos con un commit.

Git rm

Por otro lado, git rm es un comando que nos ayuda a eliminar archivos de Git sin eliminar su historial del sistema de versiones. Para recuperar el archivo eliminado, necesitamos retroceder en la historia del proyecto, recuperar el último commit y obtener la última confirmación antes de la eliminación del archivo.

Es importante tener en cuenta que git rm no puede usarse sin evaluarlo antes. Debemos usar uno de los flags siguientes para indicarle a Git cómo eliminar los archivos que ya no necesitamos en la última versión del proyecto.

Variaciones de Git rm

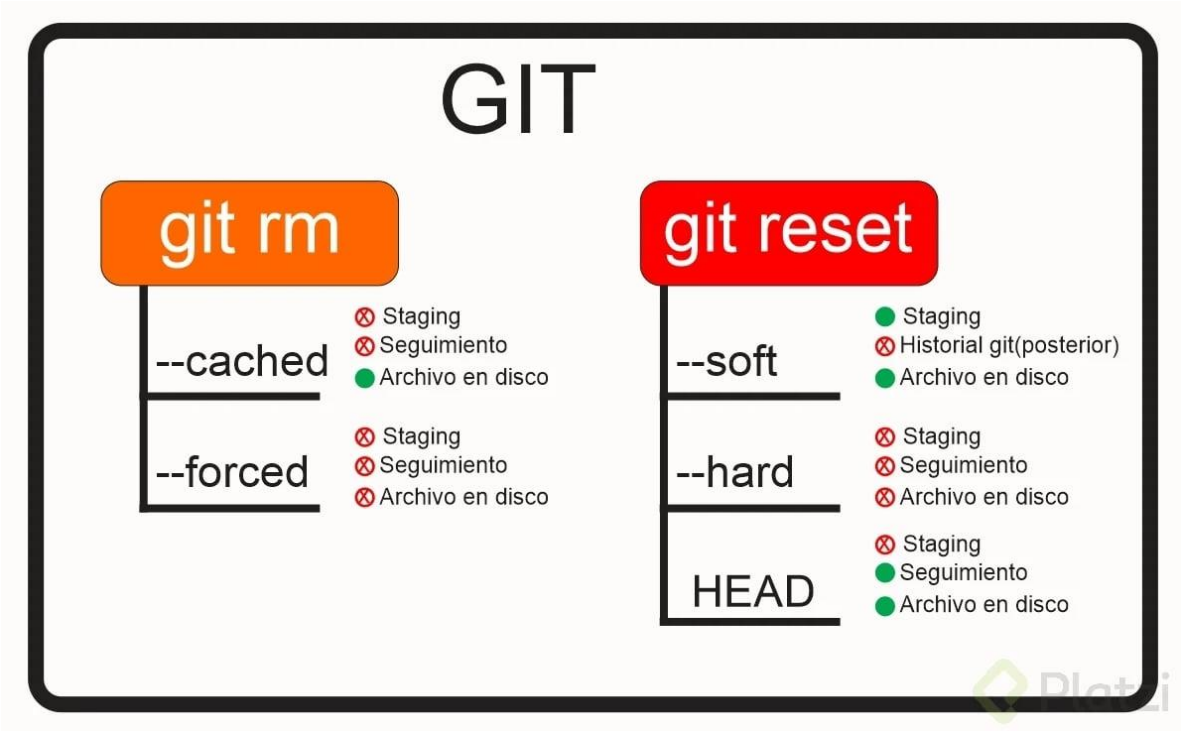
- git rm --cached: Elimina archivos del repositorio local y del área de staging, pero los mantiene en el disco duro. Deja de trackear el historial de cambios de estos archivos, por lo que quedan en estado untracked.
- git rm --force: Elimina los archivos de Git y del disco duro. Git guarda todo, por lo que podemos recuperar archivos eliminados si es necesario (empleando comandos avanzados).

¡Al usar git rm lo que haremos será eliminar este archivo completamente de git!

¿Cuál es la diferencia entre git rm y git reset Head?

La diferencia principal entre git rm y git reset HEAD radica en que git rm elimina archivos del repositorio y de la historia del proyecto,

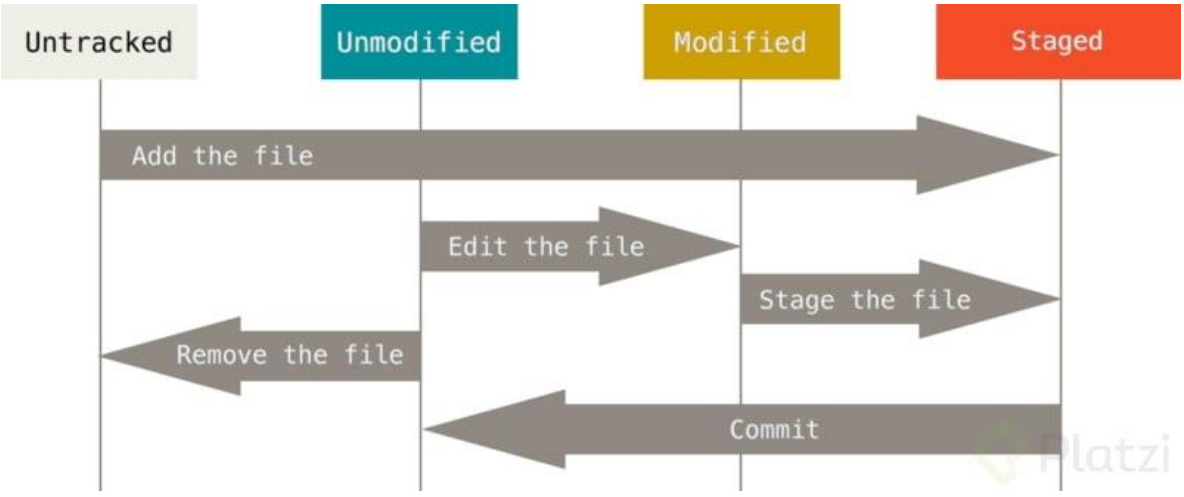
mientras que git reset saca los cambios del área de preparación y los mueve del espacio de trabajo, sin afectar la historia del repositorio.



Es importante tener en cuenta el efecto que cada comando tiene en el proyecto y usarlos según tus necesidades y objetivos específicos.

¿Cuándo utilizar git reset en lugar de git revert?

Para reescribir la historia del repositorio y eliminar confirmaciones anteriores, se utiliza git reset. Para deshacer cambios de confirmaciones anteriores de forma segura sin modificar la historia del repositorio, se emplea git revert.



Resumen

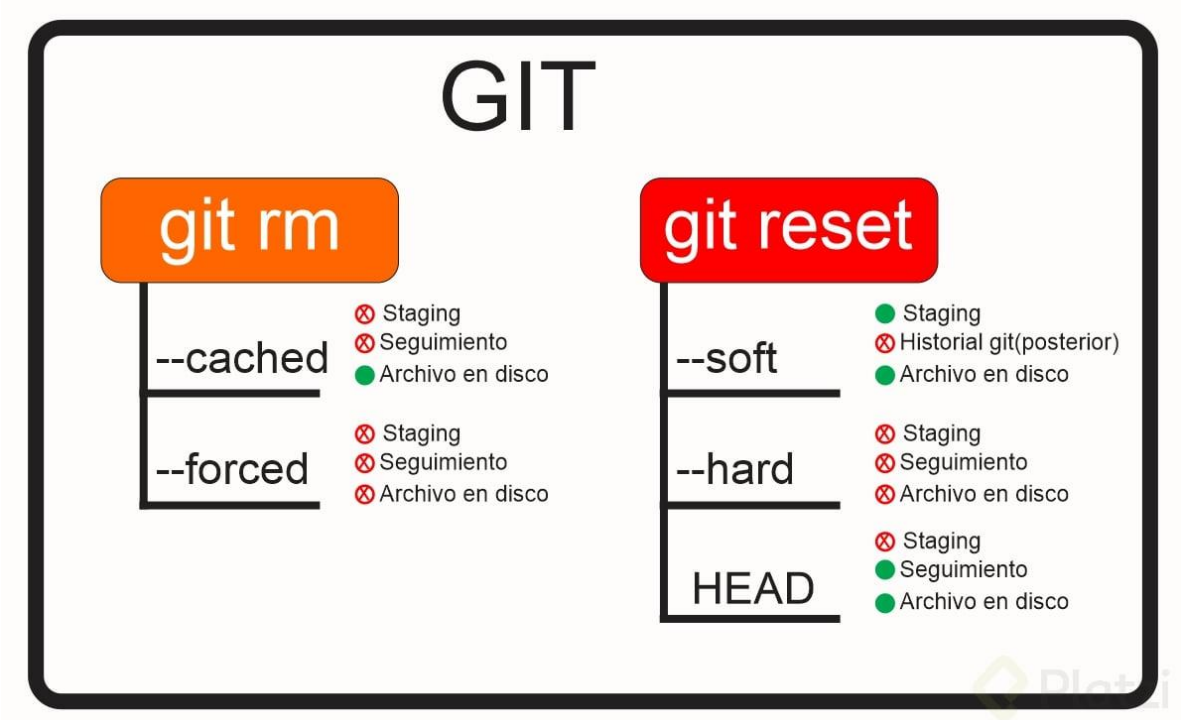
Para evitar problemas en el trabajo, es valioso entender las implicaciones y riesgos de cada comando y elegir el enfoque adecuado según las necesidades y el flujo de trabajo del proyecto.

Con git rm eliminamos un archivo de Git, pero mantenemos su historial de cambios. Si no queremos borrar un archivo, sino dejarlo como está y actualizarlo después, no debemos usar este comando en este commit.

Empleando git reset HEAD, movemos los cambios de Staging a Unstaged, pero mantenemos el archivo en el repositorio con los últimos cambios en los que hicimos commit. Así, no perdemos nada relevante.

Siguientes pasos

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.



FLUJO DE TRABAJO BÁSICO EN GIT

FLUJO DE TRABAJO BÁSICO CON UN REPOSITORIO REMOTO

Cuando empiezas a trabajar en un entorno local, el proyecto vive únicamente en tu computadora. Esto significa que no hay forma de que otros miembros del equipo trabajen en él.

Para solucionar esto, utilizamos 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. Así, el equipo podrá 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

Comandos para trabajo remoto con GIT

- **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.

Adicionalmente, tenemos otros comandos que nos sirven para trabajar en proyectos muy grandes:

- **git log --oneline**:Te muestra el id commit y el título del commit.
- **git log --decorate**: Te muestra donde se encuentra el head point en el log.
- **git log --stat**: Explica el número de líneas que se cambiaron brevemente.
- **git log -p**: Explica el número de líneas que se cambiaron y te muestra que se cambió en el contenido.
- **git shortlog**: Indica que commits ha realizado un usuario, mostrando el usuario y el título de sus commits.
- **git log --graph --oneline --decorate** y
- **git log --pretty=format:"%cn hizo un commit %h el dia %cd"**: Muestra mensajes personalizados de los commits.
- **git log -3**: Limitamos el número de commits.
- **git log --after="2018-1-2"**
- **git log --after="today"** y
- **git log --after="2018-1-2" --before="today"**: Commits para localizar por fechas.
- **git log --author="Name Author"**: Commits hechos por autor que cumplan exactamente con el nombre.
- **git log --grep="INVIE"**: Busca los commits que cumplan tal cual está escrito entre las comillas.
- **git log --grep="INVIE" –i**: Busca los commits que cumplan sin importar mayúsculas o minúsculas.
- **git log – index.html**: Busca los commits en un archivo en específico.
- **git log -S "Por contenido"**: Buscar los commits con el contenido dentro del archivo.
- **git log > log.txt**: guardar los logs en un archivo txt

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 titulo de sus commits.
6. git log --graph --oneline --decorate y
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" y
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

GIT

FLUJO DE TRABAJO BÁSICO CON UN REPOSITORIO REMOTO

IDEAS

Estos servidores remotos pueden estar alojados en GitHub, GitLab, BitBucket, entre otros.

Estos repositorios pueden ser publicos o privados, que todo el mundo lo pueda ver o que solo ciertas personas

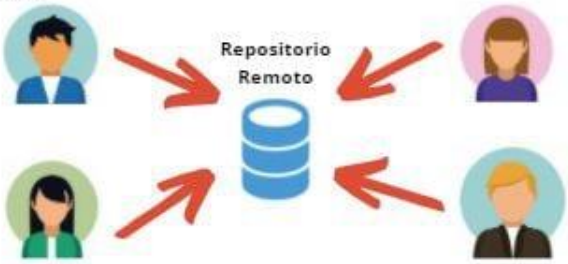
Para que otra persona pueda hacer cambios en enuestro repositorio remoto le tenemos que dar permisos (el creador del repositorio)

NOTAS DE CLASE

Repositorios remotos

Un repositorio remoto es un lugar en internet donde almacenamos nuestro repositorio para que no solo viva en local sino que cualquier persona del equipo (que tenga permisos) pueda verlo y contribuir a nuestro proyecto entre otras cosas mas.

Estos repositorios remotos facilitan el trabajo colaborativo puesto que cada uno de los integrantes de nuestro equipo podrá entrar y ver los cambios hechos , aportar a estos cambios y demas cosas



RESUMEN

Cuando tenemos un equipo de trabajo es necesario usar un repositorio remoto en donde todos podamos trabajar.Estos repositorios pueden ser públicos o privados , dentro de los repositorios remotos mas famosos esta github

INTRODUCCIÓN A LAS RAMAS O BRANCHES DE GIT

Las ramas (branches) 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).

Repasa: ¿Qué es Git?

Cómo funcionan las ramas en GIT

Las ramas son la manera 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.

- **git branch -nombre de la rama-**: Con este comando se genera una nueva rama.
- **git checkout -nombre de la rama-**: Con este comando puedes saltar de una rama a otra.
- **git checkout -b rama**: Genera una rama y nos mueve a ella automáticamente, Es decir, es la combinación de git branch y git checkout al mismo tiempo.
- **git reset id-commit**: Nos lleva a cualquier commit no importa la rama, ya que identificamos el id del tag., eliminando el historial de los commit posteriores al tag seleccionado.
- **git checkout rama-o-id-commit**: Nos lleva a cualquier commit sin borrar los commit posteriores al tag seleccionado.

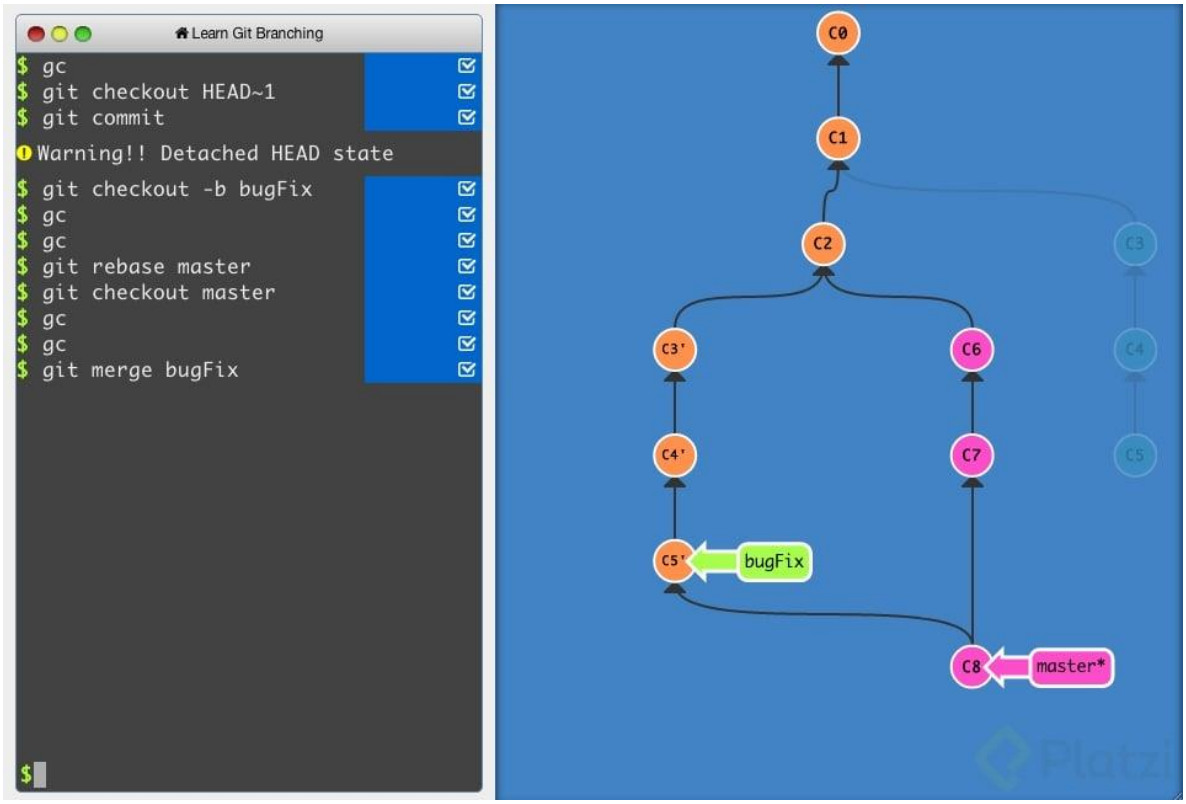
FUSIÓN DE RAMAS CON GIT MERGE

La **fusión en Git** es la forma en que este sistema une un historial bifurcado. El comando git merge permite integrar líneas de desarrollo independientes generadas por git branch en una sola rama. Con este comando, podemos crear un nuevo *commit* que combina dos ramas o branches: la rama actual y la rama que se indica después del comando.

Estos comandos de fusión del merge afectan solo a la rama actual y no a la rama de destino. Por lo tanto, te recomendamos utilizar git checkout para seleccionar la rama actual y git branch -d para eliminar la rama de destino obsoleta.

Funcionamiento de Git merge

Git merge fusiona secuencias de confirmaciones en un solo historial, generalmente para combinar dos ramas. Busca una confirmación de base común y genera una confirmación de fusión que representa la combinación de las dos ramas hasta el resultado final.



¿Cómo unir dos ramas en git?

Ahora bien, para combinar ramas en tu repositorio local, usa git checkout para cambiar a la rama donde deseas fusionar. Por lo general, esta es la rama principal. Luego, emplea git merge y especifica el nombre de la otra rama que deseas traer a esta rama. Ten en cuenta que esto es una combinación de avance rápido.

¿Cómo realizar un merge en git?

Para hacer un merge en Git, primero asegúrate de estar en la rama correcta. Después, usa el comando git merge seguido del nombre de la rama que quieres combinar. Por ejemplo, si quieres crear un nuevo commit en la rama master con los cambios de la rama cabecera, usa este comando:

git checkout **master**

git merge cabecera

Es importante tener en cuenta que en caso de haber conflictos, debes guardar tus cambios antes de hacer git checkout para evitar perder tu trabajo. También es recomendable emplear los comandos básicos de GitHub, como git fetch, git push y git pull, para mantener actualizado tu repositorio.

En este ejemplo, vamos a crear un nuevo *commit* en la rama *master* combinando los cambios de una rama llamada *cabecera*: Otra opción es crear un nuevo *commit* en la rama *cabecera* combinando los cambios de cualquier otra rama:

Git es asombroso porque puede saber cuáles cambios deben conservarse en una rama y cuáles no. En casos de conflictos, asegúrate de guardar tus cambios antes de hacer git checkout para evitar perder tu trabajo.

Comandos básicos de GitHub

- git init: crear un repositorio.
- git add: agregar un archivo a staging.
- git commit -m “mensaje”: guardar el archivo en git con un mensaje.
- git branch: crear una nueva rama.

- git checkout: moverse entre ramas.
- git push: mandar cambios a un servidor remoto.
- git fetch: traer actualizaciones del servidor remoto y guardarlas en nuestro repositorio local.
- git merge: tiene dos usos. Uno es la fusión de ramas, funcionando como un *commit* en la rama actual, trayendo la rama indicada. Su otro uso es guardar los cambios de un servidor remoto en nuestro directorio.
- git pull: fetch y merge al mismo tiempo.
- git checkout "codigo de version" "nombre del archivo": volver a la última versión de la que se ha hecho *commit*.
- git reset: vuelve al pasado sin posibilidad de volver al futuro, se debe usar con especificaciones.
- git reset --soft: vuelve a la versión en el repositorio, pero guarda los cambios en staging. Así, podemos aplicar actualizaciones a un nuevo *commit*.
- git reset --hard: todo vuelve a su versión anterior
- git reset HEAD: saca los cambios de staging, pero no los borra. Es lo opuesto a git add.
- git rm: elimina los archivos, pero no su historial. Si queremos recuperar algo, solo hay que regresar. se utiliza así:git rm --cached elimina los archivos en staging pero los mantiene en el disco duro.git rm --force elimina los archivos de git y del disco duro.
- git status: estado de archivos en el repositorio.
- git log: historia entera del archivo.
- git log --stat: cambios específicos en el archivo a partir de un commit.
- git show: cambios históricos y específicos hechos en un archivo.
- git diff "codigo de version 1" "codigo de version 2": comparar cambios entre versiones.
- git diff: comparar directorio con *staging*.

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 Visual Studio Code nos ayudan a resolver estos conflictos sin necesidad de borrar o escribir líneas de texto, basta con hacer clic en 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.

Cómo revertir un merge

Si nos hemos equivocado y queremos cancelar el merge, debemos usar el siguiente comando:

git merge --abort

Conflictos en repositorios remotos

Al trabajar con otras personas, es necesario utilizar un repositorio remoto.

-Para copiar el repositorio remoto al directorio de trabajo local, se utiliza el comando git clone <url>, y para enviar cambios al repositorio remoto se utiliza git push.
-Para actualizar el repositorio local se hace uso del comando git fetch, luego se debe fusionar los datos traídos con los locales usando git merge.

- Para traer los datos y fusionarlos a la vez, en un solo comando, se usa git pull.
  - Para crear commits rápidamente, fusionando git add y git commit -m "", usamos git commit -am "".
  - Para generar nuevas ramas, hay que posicionarse sobre la rama que se desea copiar y utilizar el comando git branch <nombre>.
- Para saltar entre ramas, se usa el comando git checkout <branch>
  - Una vez realizado los cambios en la rama, estas deben fusionarse con git merge.
- El merge ocurre en la rama en la que se está posicionado. Por lo tanto, la rama a fusionar se transforma en la principal.
- Los merges también son commits.
- Los merges pueden generar conflictos, esto aborta la acción y pide que soluciones el problema manualmente, aceptando o rechazando los cambios que vienen.

Repasa qué es un branch

TRABAJANDO CON REPOSITORIOS REMOTOS EN GITHUB

CÓMO FUNCIONAN LAS LLAVES PÚBLICAS Y PRIVADAS

Las **llaves públicas y privadas**, conocidas también como cifrado asimétrico de un solo camino, sirven para mandar mensajes privados entre varios nodos con la lógica de que firmas tu mensaje con una llave pública vinculada con una llave privada que puede leer el mensaje.

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.

Cómo funciona un mensaje cifrado con llaves públicas y privadas

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 emplear su llave privada para descifrar el mensaje y ver los archivos.Nota: puedes compartir tu llave pública, pero nunca tu llave privada.

CONFIGURA TUS LLAVES SSH EN LOCAL

En este ejemplo, aprenderemos **cómo configurar nuestras llaves SSH en local**.

Cómo generar tus llaves SSH

1. 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"

2. 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**

Configurar llaves SSH en Git y GitHub

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.

Cómo conectar un repositorio de GitHub a nuestro documento local

Si queremos conectar el repositorio de GitHub con nuestro repositorio local, que creamos usando el comando git init, debemos ejecutar las siguientes instrucciones:

1. Guardar la URL del repositorio de GitHub con el nombre de origin

git remote add origin URL

2. Verificar que la URL se haya guardado correctamente:

git remote

git remote -v

3. 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 git pull con el *flag* --allow-unrelated-histories:

git pull origin **master --allow-unrelated-histories**

4. Por último, ahora sí podemos hacer git push para guardar los cambios de nuestro repositorio local en GitHub:

git push origin **master**

Cómo autenticarte en GitHub 2022

Antes de empezar debemos renombrar la rama 'máster' a 'main', este es el nuevo estándar en GitHub, para esto:

1. Primero nos posicionamos en la rama a la que queremos cambiarle el nombre.
2. Ejecutamos el siguiente comando: git branch -M main

Pasos para crear un token de acceso personal.

Desde el 2022 GitHub ya no deja hacer el push con la contraseña del propio GitHub, para esto tenemos que crear un token, y este token es la contraseña que vamos a colocar cuando nos pida clave.

Descubre el uso de tags en Github

Seguir la secuencia: Ingresamos a nuestra cuenta de GitHub.

1. Buscamos Settings
2. Click en Developer settings
3. Click en Personal access tokens
4. Click en Generate new token aquí se puede colocar un nombre, la fecha de expiración.
5. Tildar en repo y luego click en el botón verde Generate token

COMANDOS GIT

Comando	Descripción
$ git merge nombre_rama	Fusiona una rama nombre_rama con la actual (con la rama en la que nos encontramos posicionados en el HEAD).
$ git merge rama2 --allow-unrelated-histories	Obliga a fusionar la rama rama2 con la actual (con la rama en la que nos encontramos posicionados en el HEAD).
$ git remote add origin url	Conecta un repositorio en la url y da un origen de **Repositorio Remoto (GitHub)** a nuestro equipo local en el **Git Repository**.
$ git remote -v	Lista las conexiones existentes. Donde hacer fetch y push
$ git remote remove origin	Elimina una conexión con **Repositorio Remoto (GitHub)**.
$ git push origin master	Enviar la rama master a él **Repositorio Remoto (GitHub)**.
$ git push origin master --tags	Enviar los tags la rama master a él **Repositorio Remoto (GitHub.**
$ git pull origin master	Descarga cambios de **Repositorio Remoto (GitHub)** y los fusiona automáticamente la rama espejo master de nuestro entorno local **Git Repository**.
$ git pull origin master --allow-unrelated-histories	Obliga a fusionar la rama de origin master del **Repositorio Remoto (GitHub)** con la actual (con la rama en la que nos encontramos posicionados en el HEAD), que esta en el entorno local **Git Repxository.**

Notas:

- **Working Directory**: Son los archivos Untracked O Unstaged, que No están dentro de Git o esta desactualizado ya que no ha sido afectado por el comando *git add* y su existencia o sus últimas actualizaciones solo están en Disco Duro.
- **Staging Area: Son los archivos Staged, que SI están dentro de Git han sido afectados por el comando *git add* tienen cambios pendientes, pero aun no han sido guardados en el repositorio falta ejecutar el comando *git commit*.**
- **Git Repository**: Son los archivos Tracked, que Si están dentro de Git, no tienen cambios pendientes y sus ultimas actualizaciones han sido guardadas con el comando *git commit*.
- **Repositorio Remoto**: Son los archivos que se encuentran en un repositorio remoto y que pueden ver y trabajar todos los miembros del equipo estos servidores pueden ser GitHub, GitLab, BitBucket, entre otros.

Paso a paso para subir tu proyecto a GitHub

```
1  # Colocamos el URL de nuestro repositorio de GitHub
   $ git remote add origin (URL)

   # Verificamos que la URL se haya guardado:
2  $ git remote
   $ git remote -v

   # Traemos la versión del repositorio remoto y hacenos merge
     para crear un commit con los archivos de ambas partes.
     Podemos usar git fetch y git merge o solo lo sigueinte:
3  $ git pull origin main --allow-unrelated-histories

   # Ahora sí podemos hacer git push para guardar los cambios de
     nuestro repositorio local en GitHub:
4  $ git push origin master:main
```

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

TU PRIMER PUSH

La creación de las SSH es necesario solo una vez por cada computadora. Aquí conocerás **cómo conectar a GitHub usando 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:

ssh

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

Comandos para copiar la llave SSH:

-**Mac**:

pbcopy < ~/.ssh/id_rsa.pub

- **Windows (Git Bash)**:

clip < ~/.ssh/id_rsa.pub

- **Linux (Ubuntu)**:

cat ~/.ssh/id_rsa.pub

Descubre cómo funcionan los Git Tags.

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	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>	Configura git para conectar con el repositorio remoto a través de SSH (en lugar de HTTPS)

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>	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.

[Configurar llaves SSH en Git y GitHub]

GIT TAG Y VERSIONES EN GITHUB

En Git, las etiquetas o **Git tags** tienen un papel importante al asignar versiones a los commits más significativos de un proyecto. Aprender a utilizar el comando git tag, entender los diferentes tipos de etiquetas, cómo crearlas, eliminarlas y compartirlas, es esencial para un flujo de trabajo eficiente.

Creación de etiquetas en Git

Para crear una etiqueta, ejecuta el siguiente comando:

git **tag <tagname**>

Sustituye **<tagname>** con un identificador semántico que refleje el estado del repositorio en el momento de la creación. Git admite etiquetas anotadas y ligeras.

Las etiquetas anotadas almacenan información adicional como la fecha, etiquetador y correo electrónico, y son ideales para publicaciones

públicas. Las etiquetas ligeras son más simples y se emplean como "marcadores" de una confirmación específica.

Listado de etiquetas

Para obtener una lista de etiquetas en el repositorio, ejecuta el siguiente comando:

git **tag**

Esto mostrará una lista de las etiquetas existentes, como:

v1.0

v1.1

v1.2

Para perfeccionar la lista, puedes utilizar opciones adicionales, como **-l** con una expresión comodín.

Uso compartido de etiquetas

Compartir etiquetas requiere un enfoque explícito al usar el comando **git push**. Por defecto, las etiquetas no se envían automáticamente. Para enviar etiquetas específicas, utiliza:

git push origin <tagname>

Para enviar varias etiquetas a la vez, usa:

git push origin --tags

Eliminación de etiquetas

Para eliminar una etiqueta, usa el siguiente comando:

git **tag -d** <tagname>

Esto eliminará la etiqueta identificada por **<tagname>** en el repositorio local.

En resumen, las etiquetas en Git son esenciales para asignar versiones y capturar instantáneas importantes en el historial de un proyecto. Aprender a crear, listar, compartir y eliminar etiquetas mejorará tu flujo de trabajo con Git.

MANEJO DE RAMAS EN GITHUB

Si no te funciona el comando gitk es posible no lo tengas instalado por defecto.
Para instalar gitk debemos ejecutar los siguientes comandos:
sudo apt-get update
sudo apt-get install gitk

Repasa: ¿Qué es Git?

Las ramas nos permiten hacer cambios a nuestros archivos sin modificar la versión principal (master). 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 crucial es que aprendas a manejarlas para trabajar profesionalmente.

Si, estando en otra rama, modificamos los archivos y hacemos *commit*, tanto el historial(git log) como los archivos serán afectados. La ventaja que tiene usar ramas es que las modificaciones solo afectarán a esa rama en particular. Si luego de "guardar" los archivos(usando commit) nos movemos a otra rama (git checkout otraRama) veremos como las modificaciones de la rama pasada **no aparecen** en la otraRama.

Comandos para manejo de ramas en GitHub

- Crear una rama:
  git branch branchName
- Movernos a otra rama:
  git checkout branchName
- 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 utilizando el comando gitk. Gitk fue el primer visor gráfico que se desarrolló para ver de manera gráfica el historial de un repositorio de Git.

Repasa: Qué es branch.

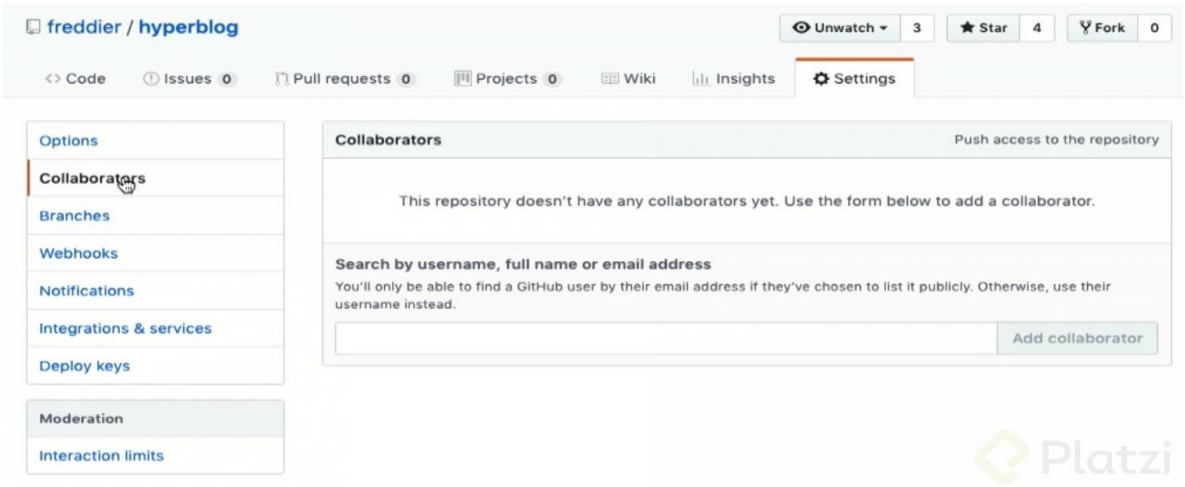
CONFIGURAR MÚLTIPLES COLABORADORES EN UN REPOSITORIO DE GITHUB

Por defecto, cualquier persona puede clonar o descargar tu proyecto desde GitHub, pero no pueden crear commits, ni ramas. Esto quiere decir que pueden copiar tu proyecto pero no colaborar con él. 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.

Cómo agregar colaboradores en Github

- Solo debemos entrar a la configuración de colaboradores de nuestro proyecto. Se encuentra en:
  Repositorio > Settings > Collaborators

Ahí, debemos añadir el email o username de los nuevos colaboradores.



Si, como colaborador, agregaste erróneamente el mensaje del *commit*, lo puedes cambiar de la siguiente manera:

- Hacer un *commit* con el nuevo mensaje que queremos, esto nos abre el editor de texto de la terminal:
  git commit —amend
- Corregimos el mensaje
- Traer el repositorio remoto
  git pull origin master
- Ejecutar el cambio
  git push --set-upstream origin master

FLUJOS DE TRABAJO PROFESIONALES

FLUJO DE TRABAJO PROFESIONAL: HACIENDO MERGE DE RAMAS DE DESARROLLO A MASTER

Para poder desarrollar software de manera óptima y ordenada, necesitamos tener un flujo de trabajo profesional, que nos permita trabajar en conjunto sin interrumpir el trabajo de otros desarrolladores. Una buena práctica de flujo de trabajo sería la siguiente:

1. Crear ramas
2. Asignar una rama a cada programador
3. El programador baja el repositorio con git pull origin master
4. El programador cambia de rama
5. El programador trabaja en esa rama y hace *commits*
6. El programador sube su trabajo con git push origin #nombre_rama
7. El encargado de organizar el proyecto baja, revisa y unifica todos los cambios

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*.

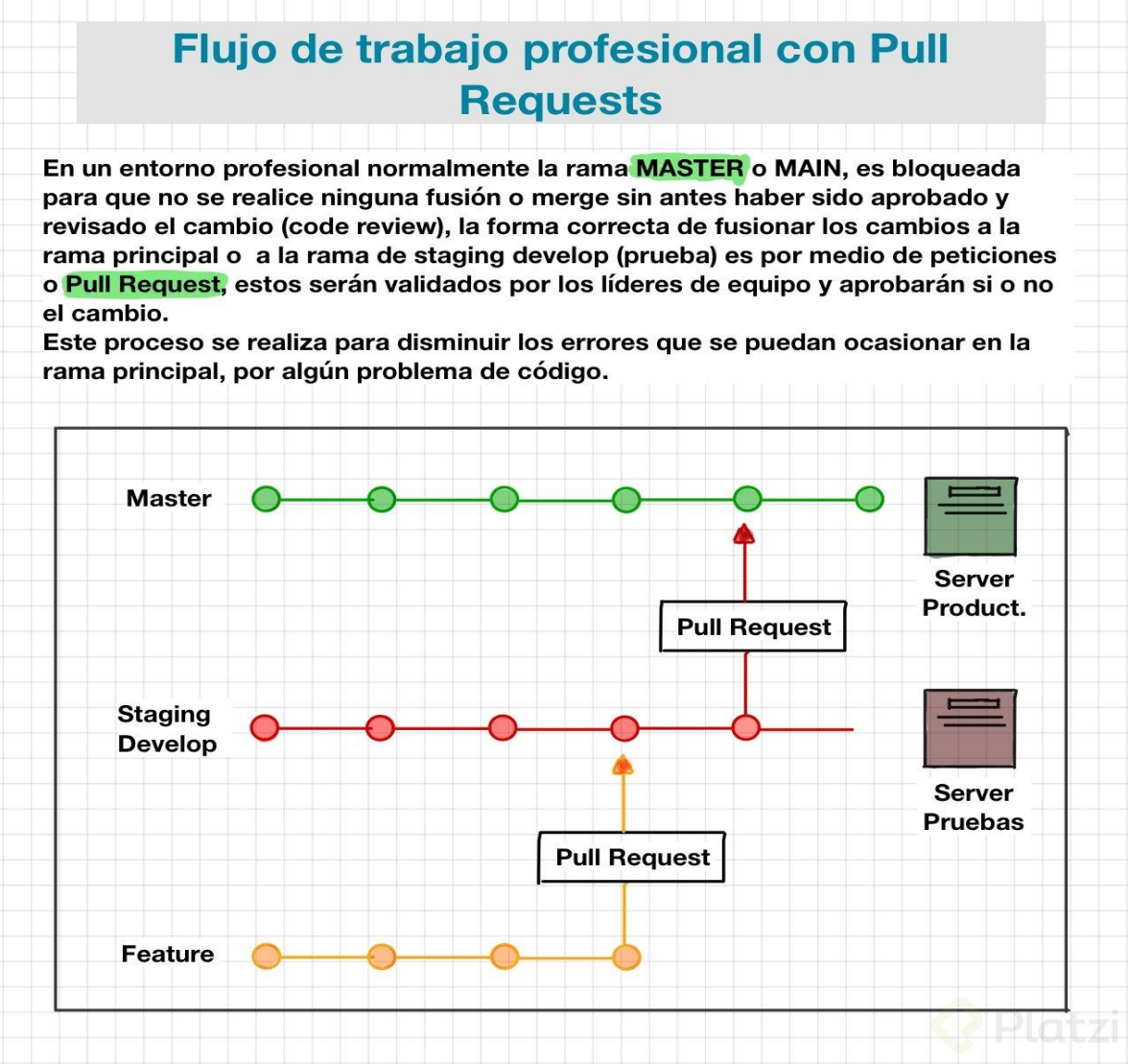
Para realizar pruebas enviamos el código a servidores que normalmente los llamamos *staging develop* (servidores de pruebas) luego de que se realizan las pruebas pertinentes tanto de código como de la aplicación estos pasan al servidor de producción con el ya antes mencionado *merge request*.

Los PR (pull requests) son la base de la colaboración a proyectos Open Source, si tienen pensando colaborar en alguno es muy importante entender esto y ver cómo se hace en las próximas clases. Por lo general

es forkear el proyecto, implementar el cambio en una nueva rama, hacer el PR y esperar que los administradores del proyecto hagan el merge o pidan algún cambio en el código o commits que hiciste.

Proceso de un pull request para trabajo en producción:

- Un pull request es un estado intermedio antes de enviar el merge.
- El pull request permite que otros miembros del equipo revisen el código y así aprobar el merge a la rama.
- Permite a las personas que no forman el equipo, trabajar y colaborar con una rama.
- La persona que tiene la responsabilidad de aceptar los pull request y hacer los merge tienen un perfil especial y son llamados DevOps



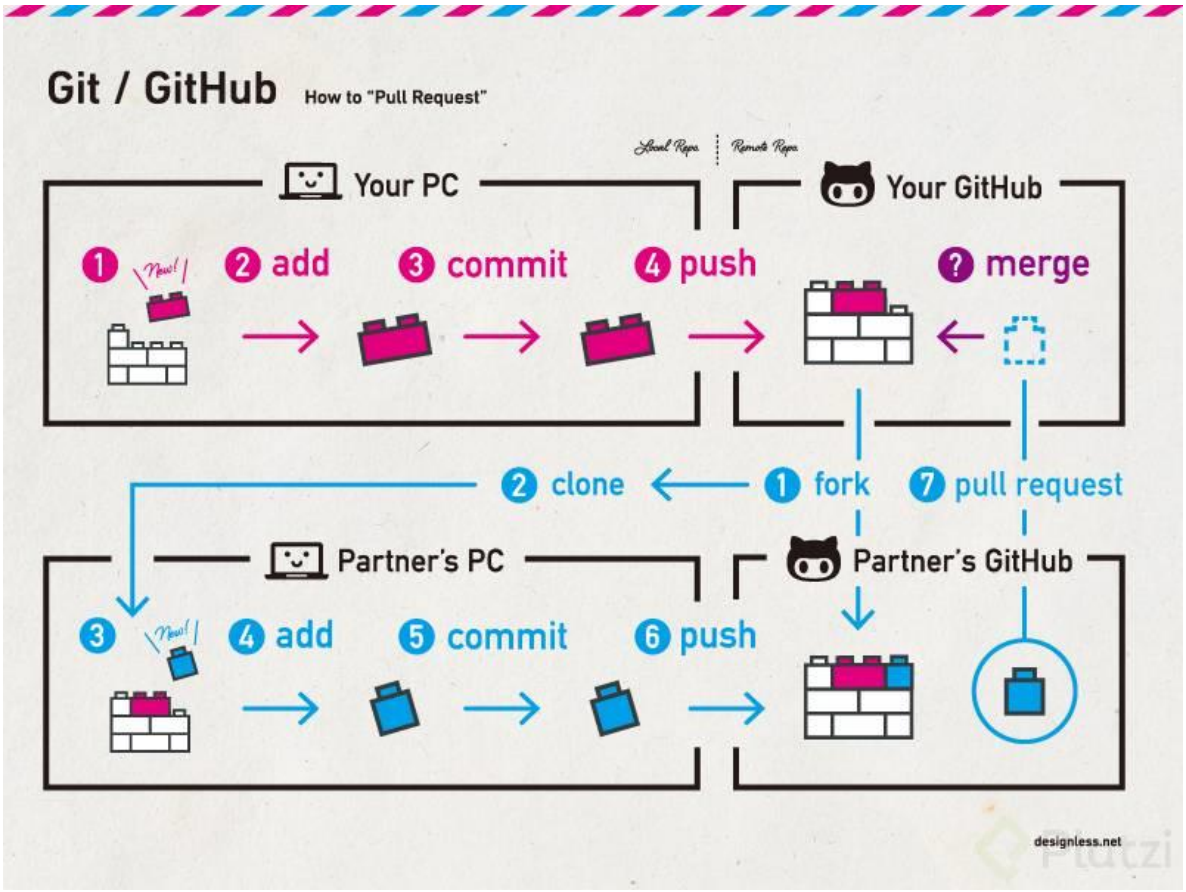
UTILIZANDO PULL REQUESTS EN GITHUB

Un **Pull Request** es una función de GitHub que permite a tu equipo solicitar la revisión y aprobación de sus cambios antes de fusionarlos en la rama principal de desarrollo, denominada “master” o “main”.

Al crear un PR se genera una conversación en la que los demás miembros pueden seguir y comentar los cambios propuestos en el código del repositorio. Esto permite detectar cualquier error o problema potencial antes de integrarlos en la rama principal, lo que ayuda a mantener el proyecto más limpio y estable.

Estructura de la incorporación de cambios

El proceso para hacer un pull request consiste en indicar la rama y repositorio de origen y destino. De esta forma, un desarrollador podrá incluir tus cambios en su proyecto.

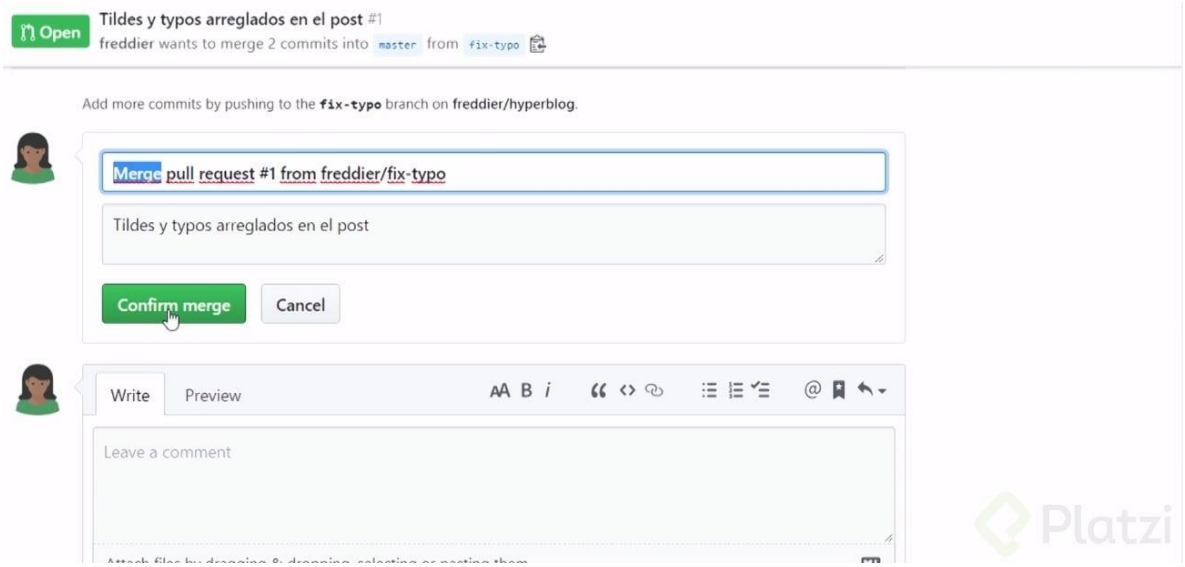


Bitbucket tiene una configuración predeterminada que funciona para la mayoría de los equipos, pero cada equipo puede personalizarla para ajustarla a su propio flujo de trabajo.

Cómo hacer un pull request

Un **PR** es un proceso crucial para facilitar la revisión y la integración efectiva del código. A continuación, veremos el paso a paso.

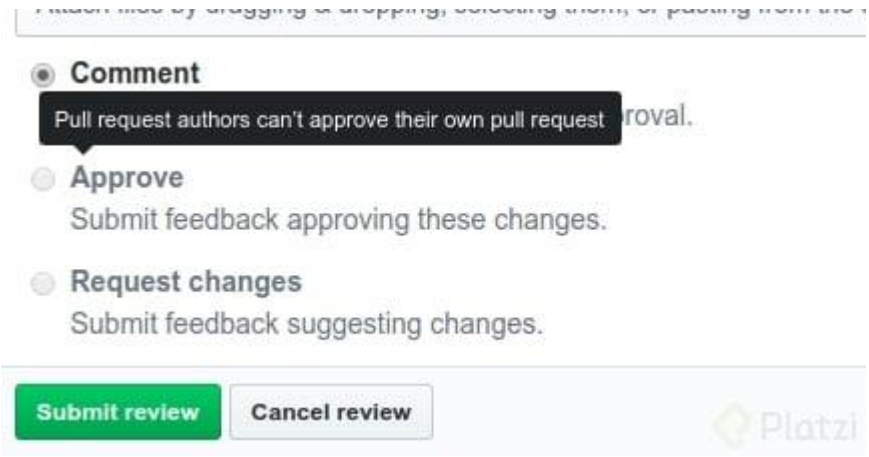
Solicitando un pull request



1. **Crea una rama paralela**: Antes de hacer cambios en el código, utiliza el comando **git checkout -b <rama>** para crear una nueva rama. Así, podrás hacer tus modificaciones sin afectar la rama principal (por ejemplo, **master**).
2. **Realiza commits**: Después de hacer cambios en los archivos, usa **git commit -am '<Comentario>'** para hacer un commit con un mensaje descriptivo.
3. **Sube los cambios**: Usa **git push origin <rama>** para subir tus cambios de la rama local al repositorio remoto.
   Reemplaza **<rama>** con el nombre de tu rama.
4. **Crea un pull request**: En el repositorio remoto (como GitHub), crea un nuevo pull request. Selecciona la rama principal como destino y tu rama con los cambios como comparación.
5. **Feedback**: Los revisores examinarán los cambios. Usa la sección de comentarios del pull request para discutir los cambios y proporcionar feedback adicional.

6. **Realiza los cambios solicitados**: Si se solicitan cambios, regresa a tu rama local y haz las modificaciones necesarias. Luego, sube los cambios al repositorio remoto usando **git push origin <rama>**.

Aceptando un pull request



1. **Acepta los cambios en GitHub**: Si estás satisfecho con los cambios propuestos en el pull request y consideras que están listos para ser fusionados con la rama principal, acepta el pull request en GitHub. De esta forma, los cambios se fusionarán en la rama principal del repositorio.
2. **Realiza el merge en la rama principal**: Después de aceptar el pull request, selecciona la opción para realizar el merge en GitHub. Esto combinará los cambios de la rama con los cambios existentes en la rama principal (**master**).

Para solicitar y aceptar pull requests de manera efectiva, sigue estos sencillos pasos que te ayudarán a facilitar la colaboración y la mejora continua del código en tu proyecto.

¿Cómo corregir un Pull Request?

Al revisar un pull request, es posible encontrar problemas o áreas que necesiten corrección antes de que los cambios se puedan unir con la rama principal. Aquí se explica cómo corregir un pull request de forma efectiva.

1. **Lee los comentarios y feedback**: Comprueba cuidadosamente todos los comentarios y feedback proporcionados por los revisores en el pull request. Toma nota de los problemas señalados y las sugerencias de mejora.

2. **Regresa a tu rama local**: Utiliza el comando **git checkout <rama>** para volver a la rama en la que realizaste los cambios y necesitas revisar.
3. **Realiza las modificaciones**: Basándote en los comentarios y feedback recibidos, efectúa las revisiones necesarias en los archivos relevantes. Asegúrate de abordar todos los problemas señalados y seguir las sugerencias de mejora proporcionadas.
4. **Realiza un nuevo commit**: Después de efectuar las correcciones, utiliza el comando **git commit -am '<Comentario>'** para crear un nuevo commit que refleje las correcciones realizadas. Asegúrate de proporcionar un mensaje claro y descriptivo para indicar las modificaciones realizadas.
5. **Sube los cambios al repositorio remoto**: Utiliza el comando **git push origin <rama>** para subir los cambios corregidos a la rama correspondiente en el repositorio remoto.
6. **Actualiza el pull request**: Una vez que los cambios corregidos se hayan subido al repositorio remoto, el pull request se actualizará automáticamente para reflejar las modificaciones procedidas en la rama. Los revisores podrán ver los nuevos cambios y comentarios.
7. **Comunica las correcciones realizadas**: Si consideras que has abordado adecuadamente los problemas señalados y las sugerencias de mejora, puedes comunicar a los revisores que has realizado las correcciones necesarias y que el pull request está listo para ser revisado nuevamente.

Es hora de administrar tus pull requests

Al seguir estos pasos, podrás revisar un pull request de manera efectiva, asegurándote de abordar los problemas identificados y proporcionar una versión mejorada de los cambios propuestos. La comunicación abierta y clara con los revisores es fundamental durante este proceso para garantizar una colaboración exitosa.

Recuerda que cualquier modificación que realices en una rama también modificará el pull request, así que es importante que procedas con los cambios necesarios y los subas de nuevo al repositorio remoto antes de aceptar el pull request.

CREANDO UN FORK, CONTRIBUYENDO A UN REPOSITORIO

Los ***forks* o bifurcaciones** son una característica única de GitHub en la que se crea una copia exacta del estado actual de un repositorio directamente en GitHub. Este repositorio podrá servir como otro origen y se podrá clonar (como cualquier otro repositorio). En pocas palabras, lo podremos utilizar como un nuevo repositorio git cualquiera

Un fork es como una bifurcación del repositorio completo. Comparte una historia en común con el original, pero de repente se bifurca y pueden aparecer varios 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 poryecto en GitHub, te conviertes en dueñ@ del repositorio fork, puedes trabajar en este con todos los permisos, pero es un repositorio completamente diferente que el original, teniendo solamente alguna historia en común (como crédito al creado o creadora original).

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 contribuír al mismo, haciendo mejor software que pueda ser utilizado por cualquiera.

Cómo se hace un fork remoto desde consola en GitHub

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.

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 esto, puedes generar 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

Este 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 *origin* para pasar a hacer pull request.

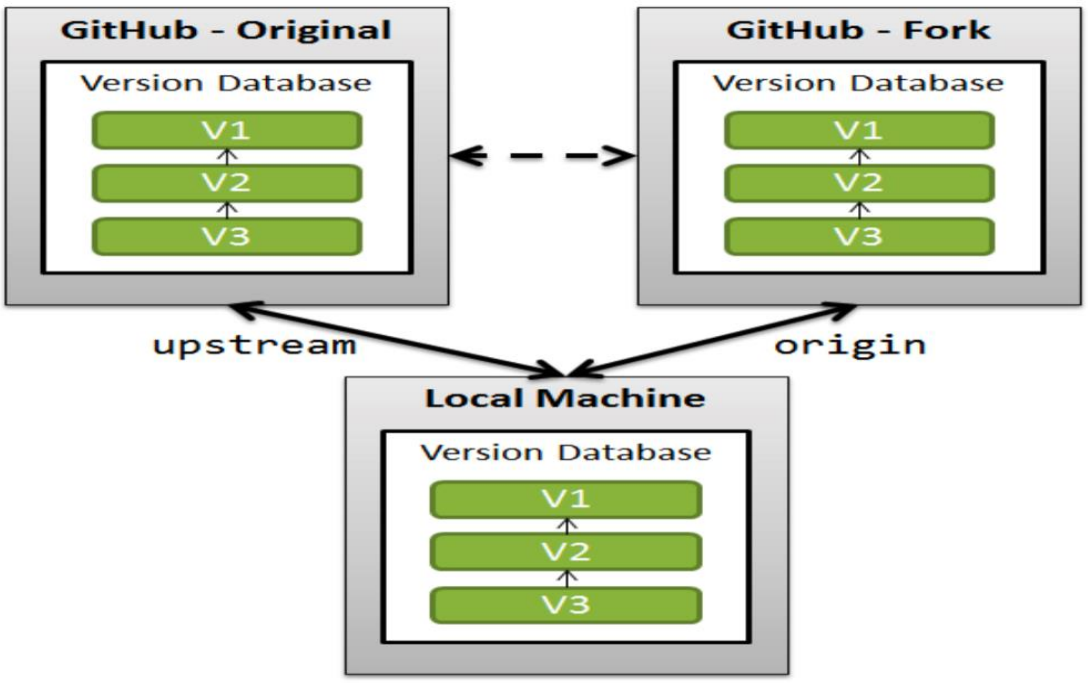
git pull upstream **master**

git push origin **master**

Solo para resaltar algo de este video, despues de hacer el clone podemos tratar de consultar las ramas que se descargaron al repositorio local con "git branch" y nos damos cuenta que solo aparece la rama master. pero y entonces… ¿que pasó con las otras ramas?

Solución: Si hacemos un "git branch --all" vamos a ver todas las ramas tanto locales como remotas y basta con cambiar de rama, por ejemplo con: "git checkout cabecera" y automaticamente se crea una copia de esta rama en el repositorio local.

Para confirmarlo haces nuevamente un "git branch" y veran que ahora si aparacen la rama master y la rama cabecera.



HACIENDO DEPLOYMENT A UN SERVIDOR

Deploy es el proceso que permite enviar al servidor uno o varios archivos. Este servidor puede ser de prueba, desarrollo o producción.

En el siguiente ejemplo veremos cómo se realiza el deployment de un documento en un servidor web básico.

Pasos para hacer deployment en un servidor web:

- Entrar a la capeta de los archivos del servidor.
- Copiar link en *clone*, elegir entre HTTPS o SSH del repositorio a contribuir.
  -En la carpeta deseada se clona el repositorio:

git **clone url**

Deploy:

- Realizar cambios y *commit* en GitHub.
- Traer al Repositorio local las actualizacion para el servidor en la capeta de los archivos del servidor.

git pull ramaRemota main

Nota: Siempre se debe proteger el archivo .git. Dependiendo del software para el servidor web, existen diferentes maneras. La conexión entre GitHub y el servidor se puede realizar mediante: Travis (pago) o Jenkis (Open source).

git

Curso Profesional de Git y GitHub

30

Haciendo deployment a un servidor

Ideas Clave

Esta es una forma sencilla de hacer un Deployment.
El problema con esto es que no es una buena practica, porque alguien podría tener acceso al archivo .git y a toda nuestra base de datos de cambios, Entonces, tenemos que proteger el .git . Si usas apache hay unas formas, o nginx hay otras. Esto lo aprendes en la carrera de administración de servidores

Cursos importantes

platzi.com/servidores

Preguntas

Formas avanzadas de hacer Deployment?

Existe una pagina llamada travis-ci.org es como un GitHub que conecta automáticamente tus ramas de GitHub con tus servidores. Solo le das permisos a tu servidor, permisos de GitHub y cuando le haces push a una rama ejemplo llamada deployment, automáticamente detecta esto y lo va a enviar.

Notas de Clase

GitHub

freddier/hyperblog

- Sabiendo que se tienen varios archivos ...

En la web:

- freddier.com

GitHub

Copiar URL para conectar en local

- Code -> Clone
  - Copiar SSH

Git

Traer a local el repositorio de GitHub

- git clone URL SSH
- ls -al

En la web

- www.freddier.com/hyperblog/blogpost.html
  - Muestra la web creada con html y CSS

- cd hyperblog/
- ls -al
- git status
- Your branch is up to date with 'origin/master'
- git pull origin master

En la web

- ls -al

Git

cd /www/freddier.com/html/

- ls -al
- Archivos
  - index.html / test.html
- cat index.html: Solo es un test del sistema
- cat text.html: Es una prueba

En la Web:

- freddier.com/test.html
  - Es una prueba

GitHub

Hacer cambios

- blogpost -> edit
- <h1> Aquí inicia la historia de un gran proyecto </h1>
- Commit changes
  - Titulo: Cambio del título visible para probar un deploy
- Commit directly to the master branch
- Commit changes
- Home hyperblog: Vemos los cambios

Git

Traer los cambios hechos en GitHub para poder ver los cambios en el navegador.

Resumen

Para hacer deploy de nuestra aplicación, no necesitas comprar un servidor, puedes instalar un servidor local en tu computadora para realizar pruebas y testear tu aplicación.

@Alevrs01

¿QUÉ ES DEPLOY EN DESARROLLO WEB?

EDteam

Es el proceso de llevar los cambios o nuevas funcionalidades de los equipos locales (de los desarrolladores) **al ambiente de producción** (el que usan los usuarios finales).

¿QUÉ NECESITAS?

DOMINIO

Es un nombre **fácil de recordar** asociado a una dirección IP de Internet. **Ej: ed.team**

SERVIDOR

Una computadora conectada a internet donde publicas **toda los recursos** de tu app web.

¡SERVIDORES PARA HACER DEPLOY GRATIS!

P.D. ¡Nunca hagas deploy en viernes!

ed.team/cursos/web

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**.

Por diversas razones, no todos los archivos que agregas a un proyecto deberían guardarse en un repositorio. Esto es porque hay archivos que no todo el mundo debería de ver, y hay archivos que al estar en el repositorio ralentizan el proceso de desarrollo (por ejemplo: los binary large objects, blob, que tardan en descargarse).

Para que no se suban estos archivos no deseados se puede crear un archivo con el nombre .gitignore en la raíz del repositorio con las reglas para los archivos que no se deberían subir: Aquí puedes ver la sintaxis de los .gitignore.

Las razones principales para tomar la decisión de no agregar un archivo a un repositorio son:

- Es un archivo con contraseñas (normalmente con la extensión .env)
- Es un blob (binary large object, objeto binario grande), mismos que son difíciles de gestionar en git.
- Son archivos que se generan corriendo comandos, por ejemplo la carpeta node_modules, que genera **npm** al correr el comando npm install

README.MD ES UNA EXCELENTE PRÁCTICA

README.md es el lugar donde se explica de qué trata el proyecto, cómo utilizarlo y demás información que se considere que se deba conocer cualquier persona que vaya a trabajar de alguna forma con el proyecto.
.
Los archivos README son escritos en un lenguaje llamado **markdown**, por eso la extensión .md, mismo que es un estándar de escritura en diversos sitios (como Platzi, Wikipedia y el mismo GitHub). Aquí puedes ver las reglas de markdown.

Los README.md pueden estar en todas las carpetas, pero el más importante es el que se encuentra en la raíz. Este documento ayuda a que los colaboradores sepan información relevante del proyecto, módulo o sección. Puedes crear cualquier archivo con la extensión .md pero solo los README.md los mostrará por defecto GitHub.

TU SITIO WEB PÚBLICO CON GITHUB PAGES

GitHub tiene un servicio de hosting gratis llamado **GitHub Pages**. Con él, puedes tener un repositorio alojado en GitHub y hacer que el contenido se muestre en la web en tiempo real.

Este es un sitio para nuestros proyectos donde lo único que tenemos que hacer es tener un repositorio alojado. En la página, podemos seguir las instrucciones para crear este repositorio

Pasos para subir un repositorio a GitHub Pages

- Debemos tomar la llave SSH y hacer un git clone #SSHexample en mi computador local (Home).
- Luego, accederemos a la carpeta nueva que aparece en nuestra máquina local.
- Creamos un nuevo archivo que se llame index.html
- Guardamos los cambios, hacemos un git pull y seguido de esto un git push a master.

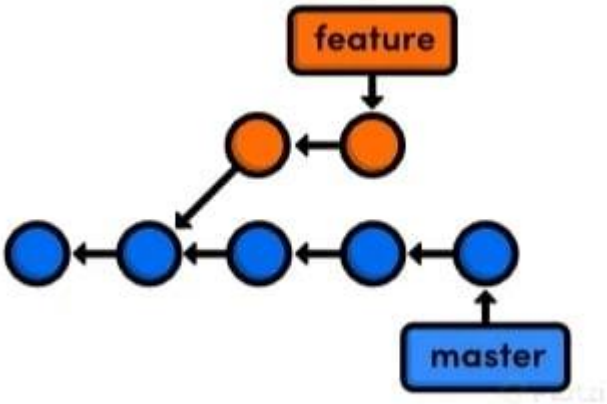
- Vamos a las opciones de *settings* de este repositorio y, en la parte de abajo, en la columna Github Pages, configuramos el *source* o fuente para que traiga la rama *master*
- Guardamos los cambios.

Después de esto, podremos ver nuestro trabajo en la web como si tuviéramos nuestro propio servidor.

MULTIPLES ENTORNOS DE TRABAJO EN GIT

GIT REBASE: REORGANIZANDO EL TRABAJO REALIZADO

Rebase es el proceso de mover o combinar una secuencia de confirmaciones en una nueva confirmación base. La reorganización es muy útil y se visualiza fácilmente en el contexto de un flujo de trabajo de ramas de funciones. El proceso general se puede visualizar de la siguiente manera.

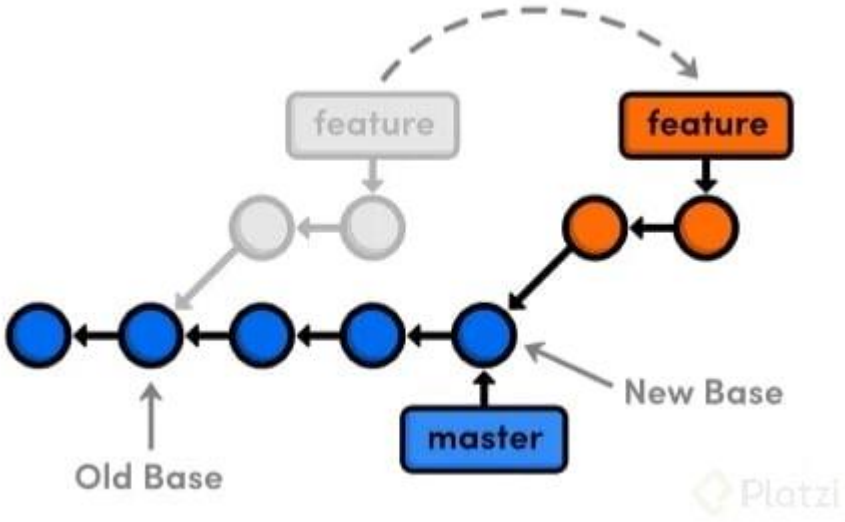


Para hacer un rebase en la rama feature de la rama master, correrías los siguientes comandos:

git checkout feature

git rebase **master**

Esto *trasplanta* la rama feature desde su locación actual hacia la punta de la rama master:



Ahora, falta fusionar la rama feature con la rama master

git checkout **master**

git rebase feature

No reorganices el historial público

Nunca debes reorganizar las confirmaciones una vez que se hayan enviado a un repositorio público. La reorganización sustituiría las confirmaciones antiguas por las nuevas y parecería que esa parte del historial de tu proyecto se hubiera desvanecido de repente.

El comando ***rebase*** es **_una mala práctica, sobre todo en repositorios remotos. Se debe evitar su uso, pero para efectos de práctica te lo vamos a mostrar, 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 vs Merge

#1. Commits

Rebase

It changes and rewrites the history by creating a new commit for each commit in the source branch.

Merge

It incorporates all the changes to the source but maintains ancestry of each commit history which incorporates all the changes to the source but maintains ancestry of each commit history.

#2. Selection

Rebase

Here we first check out the branch that needs to be rebased then select the rebase command to add updates to others.

Merge

Here we first check out the branch that needs to be merged then perform the merge operation and latest commit of the source will be merged with the latest commit of the master.

#3. Conflict Handling

Rebase

Since commit history will be rewritten so it will be difficult to understand the conflict in some cases.

Merge

Merge conflict can be easily handled understanding the mistake that was performed while merging.

#4. Golden Rule

Rebase

Should be used on public branches since commit history can cause confusion.

Merge

Not much harm while performing public branches.

#5. Reachability

Rebase

Commits that were once reachable will no longer be reachable after rebase since commit history is changed.

Merge

Commits will remain reachable from the source branches.

www.educba.com

CÓMO USAR GIT STASH: GUARDA CAMBIOS TEMPORALMENTE

El comando git stash te permite almacenar temporalmente (o guardar en un stash), los cambios que hayas efectuado en el código en el que estás trabajando para que puedas trabajar en otra cosa y, más tarde, regresar y volver a aplicar los cambios.

Guardar los cambios en stashes resulta práctico si tienes que cambiar rápidamente de contexto y ponerte con otra cosa, pero estás en medio de un cambio en el código y no tienes todo listo para confirmar los cambios.

Git stash lo puedes usar sin necesidad de crear una nueva rama o hacer un commit. Además, no pierdes tus cambios.

- **git stash:** guarda los cambios temporalmente en memoria cuando no quieres hacer un commit aun
- **git stash save "mensaje":** guarda un stach con mensaje
- **git stash list:** muestra la lista de cambios temporales
- **git stash pop:** trae de vuelta los cambios que teníamos guardados en el ultimo stash
- **git stash apply stash@{n}:** trae el stash que necesites con indicar su número dentro de las llaves
- **git stash drop:** borra el ultimo stash
- **git stash clear:** borra todos los stash

Cómo funciona el comando git stash

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"

Ejemplo de uso de git stash

El stashed nos permite cambiar de rama o branch, 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.

Por ejemplo:

$ git status

On branch master

Changes **to** be committed:

new file: style.css

Changes **not** staged **for** commit:

modified: **index**.html

$ git stash

Saved working directoryandindex state WIP onmaster: 5002d47 our **new** homepage

HEADisnowat5002d47 our **new** homepage

$ git status

On branch master

nothingtocommit, working tree clean

Cómo ver los stash en git

Utiliza git stash pop. Este muestra los cambios guardados en el stash, también podemos mostrar los cambios de un stash determinado usando su índice que nos muestra el git 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

Cómo ver el 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.

Cómo crear una rama con el stash

Para crear una rama y aplicar el stash más 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**.

Cómo 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.

git stash : Guarda el trabajo actual de manera temporal. (Archivos modificados o eliminados)
git stash -u : Crea un stash con todos los archivos. (Añadiendo los creados Untracked)
git stash save "mensaje" : Crea un stash con el mensaje especificado.
git stash list : Permite visualizar todos los stash existentes.
git stash clear : Elimina todos los stash existentes.
git stash drop : Elimina el stash más reciente. El que tiene num_stash=0.
git stash drop stash@{num_stash} : Elimina un stash específico.
git stash apply : Aplica el stash más reciente. El que tiene num_stash=0.
git stash apply stash@{num_stash} : Aplica los cambios de un stash específico.
git stash pop : Aplica el stash más reciente y lo elimina. El que tiene num_stash=0.
git stash pop stash@{num_stash} : Aplica los cambios de un stash específico y elimina lo stash.
git stash branch nombre_de_rama : Crea una rama y aplica el stash mas reciente.
git stash branch nombre_de_rama stash@{num_stash} : Crea una rama y aplica el stash especificado.

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.
- Al aplicar un stash este no se elimina, es buena práctica eliminarlo.

GIT CLEAN: LIMPIAR TU PROYECTO DE ARCHIVOS NO DESEADOS

Mientras estamos trabajando en un repositorio podemos añadir archivos a él, que realmente no forma parte de nuestro directorio de trabajo, archivos que no se deberían de agregar al repositorio remoto.

El comando clean actúa en archivos sin seguimiento, este tipo de archivos son aquellos que se encuentran en el directorio de trabajo, pero que aún no se han añadido al índice de seguimiento de repositorio con el comando add.

$ git clean

La ejecución del comando predeterminado puede producir un error. La configuración global de Git obliga a usar la opción force con el comando para que sea efectivo. Se trata de un importante mecanismo de seguridad ya que este comando no se puede deshacer.

Revisar que archivos no tienen seguimiento.

$ git clean --dry-run

Eliminar los archivos listados de no seguimiento.

$ git clean -f

Git clean tiene muchísimas opciones adicionales, que puedes explorar al ver su documentación oficial.

¿Cómo usar git stash?

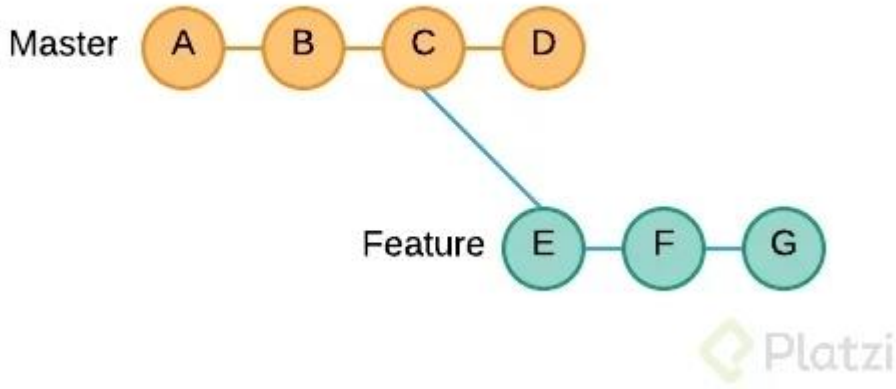
Quiero dejar bien en claro ¿cuando funciona git clean?
Git clean solo detecta archivos nuevos, no es necesario que se trate de una copia de otro archivo, suficiente con que sea un archivo nuevo que **ustedes** hayan creado.

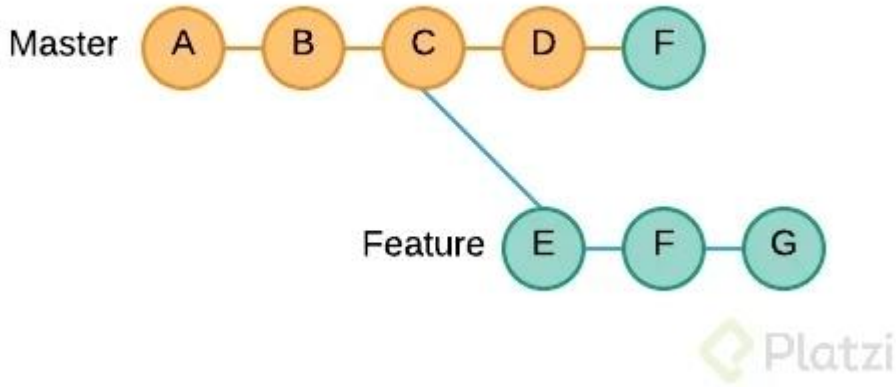
**git clean --dry-run **
Simula la eliminación de archivos, *¿tienes dudas de que archivos eliminará?*, **ejecuta git clean** --dry-run, y cuando estes seguro de que desea eliminarlos, ejecutas **git clean -f**

Archivos que se hayan modificado o editados git clean no interviene aquí.

GIT CHERRY-PICK: TRAER COMMITS ANTIGUOS AL HEAD DEL BRANCH

Git cherry-pick es un comando en Git que selecciona y aplica commits específicos de una rama o branch a otra. Todo, sin tener que hacer un merge completo. Así, podemos copiar un commit específico y aplicarlo de forma aislada en la rama de destino, conservando su historial.





Escenarios de uso de Git Cherry Pick

Este comando facilita la incorporación precisa de cambios, optimizando la colaboración y el mantenimiento en proyectos de desarrollo de software. Veamos sus casos de uso.

1. Colaboración en equipo

Antes que nada, puede ser útil implementarlo cuando diferentes miembros del equipo trabajan en áreas similares del código, pues permite seleccionar y aplicar commits específicos a cada rama, facilitando el progreso individual.

2. Solución de bugs o errores

Cuando encuentras un error o bug, es importante solucionarlo y entregar la corrección a los usuarios lo más rápido posible. Con Git Cherry Pick, puedes aplicar rápidamente un commit de verificación en la rama principal, evitando que afecte a más usuarios.

3. Deshacer cambios y recuperar commits perdidos

A veces, una rama de funcionalidad puede ser obstoleta y no ser fusionada con la rama principal. O puede suceder que una solicitud de extracción (pull request) sea cerrada sin ser fusionada.

Git nunca pierde esos commits y, a través de comandos como **git log** y **git reflog**, puedes encontrar y aplicar los commits utilizando Git Cherry Pick.

¿Cómo funciona Git Cherry Pick? Ejemplos

Imaginemos que tienes un repositorio con el siguiente estado de las ramas:

a - b - c - **d** Rama Principal

\\

e - f - **g** Rama **de** Características

El uso de Git Cherry Pick es bastante sencillo y se ejecuta de la siguiente manera:

1. Primero, asegúrate de estar en la rama principal empleando el comando **git checkout rama-principal**.
2. Luego, ejecuta el siguiente comando:

git cherry-pick commitSha

Aquí, **commitSha** es una referencia al commit que deseas aplicar. Puedes encontrar la referencia del commit utilizando el comando **git log**. Supongamos que deseas usar el commit 'f' en la rama principal.

Una vez ejecutado el comando, el historial de Git se verá así:

a - b - c - **d** - f Rama Principal

\\

e - f - **g** Rama **de** Características

De esta forma, el commit 'f' se ha incorporado correctamente a la rama principal.

El uso de Git Cherry Pick debería aplicarse con sabiduría, ya que puede generar duplicación de commits y complicaciones en el historial de cambios. Sin embargo, si sabes lo que estás haciendo, ¡adelante! Solo asegúrate de evitar su utilización si no estás seguro.

¿Cómo deshacer este comando en caso de conflicto?

Supongamos que estás usando GitHub para colaborar con un equipo en un proyecto y has realizado un cherry-pick de un commit de otra rama en tu rama local, pero ocurren conflictos durante este proceso y deseas detenerlo y volver al estado anterior.

Por suerte, en ese caso, puedes emplear el siguiente comando.

git cherry-pick --abort

Esto significa que puedes hacer las correcciones necesarias en tu rama local y volver a intentar el cherry-pick si así lo deseas.

Pon en práctica lo aprendido

Git Cherry Pick es un comando poderoso y conveniente que resulta especialmente útil en ciertas situaciones. Sin embargo, si abusas de él, podría considerarse una mala práctica en Github. Recuerda utilizarlo correctamente y comprender sus implicaciones en el historial de cambios.

Si necesitas aplicar commits específicos, no te preocupes. Siempre puedes usar el comando **git log** y Git Cherry Pick.

COMANDOS DE GIT PARA CASOS DE EMERGENCIA

GIT RESET Y REFLOG: ÚSESE EN CASO DE EMERGENCIA

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

¿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.

Atención

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

RECONSTRUIR COMMITS EN GIT CON AMEND

Git amend es una forma que tienes para hacer cambios a tu commit más recientes sin tener que hacer un nuevo commit.

Recomendaciones en el uso de git amend

El comando de git amend se usa para modificar el último commit. Es decir, te permite "revisar" o "corregir" el último cambio confirmado que hayas hecho en tu proyecto.

Por ejemplo, imagínate que acabas de hacer un commit, pero te diste cuenta de que no querías enviarlo porque faltaba algo más. En lugar de hacer un nuevo commit, puedes usar git commit --amend para agregar esos cambios al commit más reciente. Esto es útil porque mantiene tu historial de commits limpio y organizado.

Usar amend es considerado una mala práctica, especialmente después de haber hecho push o pull al repositorio remoto. Al hacer amend con algún commit que ya esté en remoto, se generará un conflicto que deberá resolverse con un commit adicional. En este proceso, se perderá el beneficio del amend.

No utilizar --amend para reconstruir commits que ya se encuentran en el repositorio remoto. Esto sería una mala práctica.

¿Cómo hacer un git amend?

Utilizar amend para *remendar* un commit puede modificar el commit más reciente (enmendar) en la misma rama. Se ejecuta de la siguiente manera:

gitadd -A # Para hacer uso de amend los archivos deben de estar en staging

git commit --amend # Remendar último commit

Para qué sirve git commit amend

Este comando sirve para agregar archivos nuevos o actualizar el commit anterior y no generar commits innecesarios. También es una forma sencilla de **editar o agregar comentarios al commit anterior** porque abrirá la consola para editar este commit anterior.

Reconstrucción de commits

Si el último commit que hicimos tenía un error, por ejemplo, de ortografía o quizá se nos olvidó agregar algo al código de ese commit podemos darle solución con el siguiente comando.

→ Modificar el mensaje del commit más reciente.

$ git **commit** --amend

→ Modificar el commit más reciente y su mensaje en la misma línea.

$ git **commit** --amend -m

Recordar que -m permite escribir un mensaje desde la línea de comandos sin tener que abrir un editor.

→ Modificar el commit sin modificar el mensaje de dicho commit.

$ git commit **--**amend **--**no**-**edit

El indicador --no-edit permite hacer correcciones en el código sin modificar el mensaje original.

Este comando es una manera práctica de modificar la información más reciente de nuestro repositorio.

BUSCAR EN ARCHIVOS Y COMMITS DE GIT CON GREP Y LOG

A medida que nuestro proyecto en Git se hace más grande, vamos a querer buscar ciertas cosas.

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

Para buscar, empleamos 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>".

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.

BONUS SOBRE GIT Y GITHUB

COMANDOS Y RECURSOS COLABORATIVOS EN GIT Y GITHUB

A continuación veremos una lista de comandos colaborativos para facilitar el trabajo remoto en 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 eliminados.
- git shortlog -sn --all --no-merge: muestra cuantos commit ha hecho cada miembro, quitando los eliminados sin los merges.
- git blame ARCHIVO: muestra quien hizo cada cosa línea por línea.
- git COMANDO --help:muestra como funciona el comando.
- git blame ARCHIVO -Llinea_inicial,linea_final: muestra quien hizo cada cosa línea por línea, indicándole desde qué línea 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.

Repasa: ¿Qué es Git?
Git stash: guarda cambios temporalmente

git config --global user.name "nombre": Configurar Nombre que salen en los commits.

git config --global user.email nombre@gmail.com: Configurar Email.

git config --global color.ui true: Marco de colores para los comando.
git help: Muestra una lista con los comandos más utilizados en GIT.

git init: Podemos ejecutar ese comando para crear localmente un repositorio con GIT y así utilizar todo el funcionamiento que GIT ofrece. Basta con estar ubicados dentro de la carpeta donde tenemos nuestro

proyecto y ejecutar el comando. Cuando agreguemos archivos y un commit, se va a crear el branch master por defecto.

git add + path: Agrega al repositorio los archivos que indiquemos.

git add : Añadimos todos los archivos para el commit.

git add –A: Agrega al repositorio TODOS los archivos y carpetas que estén en nuestro proyecto, los cuales GIT no está siguiendo.

git commit -m "mensaje" + archivos: Hace commit a los archivos que indiquemos, de esta manera quedan guardados nuestras modificaciones.

git commit -am "mensaje": Hace commit de los archivos que han sido modificados y GIT los está siguiendo.

git checkout -b NombreDeBranch: Crea un nuevo branch y automaticamente GIT se cambia al branch creado, clonando el branch desde donde ejecutamos el comando.

git branch: Nos muestra una lista de los branches que existen en nuestro repositorio.

git checkout NombreDeBranch: Sirve para moverse entre branches, en este caso vamos al branch que indicamos en el comando.

git merge NombreDeBranch: Hace un merge entre dos branches, en este caso la dirección del merge sería entre el branch que indiquemos en el comando, y el branch donde estémos ubicados.

git status: Nos indica el estado del repositorio, por ejemplo cuales están modificados, cuales no están siendo seguidos por GIT, entre otras características.

git clone URL/name.git NombreProyecto: Clona un proyecto de git en la carpeta NombreProyecto.

git push origin NombreDeBranch: Luego de que hicimos un git commit, si estamos trabajando remotamente, este comando va a subir los archivos al repositorio remoto, específicamente al branch que indiquemos.

git pull origin NombreDeBranch: Hace una actualización en nuestro branch local, desde un branch remoto que indicamos en el comando.

git tag : Muestra una lista de todos los tags.

git tag -a <verison> - m "esta es la versión x" : Crea un nuevo tags.

Dentro de **GitHub** existe la sección de **Insights** que muestra estadísticas e históricos de los colaboradores del proyecto además de Dashboard del histórico del repositorio.

Pulse: Muestra las principales contribuyentes, versiones desplegadas, pull request entre algunos datos más.

Contributors: Muestra los contribuyentes y en cada uno la línea de tiempo, mostrando la cantidad de aportaciones.

Community: Establece un espacio de discusión y muestra una lista de checkeo sobre los items que mejoran su comunidad como tener descripción, el Readme , código de conducta, licencia, contribuciones, pull request template, entre otros.

Traffic: No es público para todos los proyectos pero muestra el flujo de gente que ha visitado el repositorio en una línea de tiempo.

Commits: Gráfica cuantos commits se han hecho por día.

Code Frecuency: Muestra cuanto código se añade y cuanto se borra por semana

Dependency graph: Muestra la dependencia de otras librerías y a que repositorio pertenecen.

Alerts: Está sección es personal y no se ve en otros proyectos, pero muestra los problemas que detecta github que pueden ser relevantes.

Network: Cronología de las confirmaciones más recientes en este repositorio y su red ordenadas por envío más reciente. La red de repositorios muestra las 100 bifurcaciones impulsadas más recientemente.

Forks: Muestra en una lista quienes han hecho Fork del repositorio.