





# GIT

Git es un sistema de control de versiones distribuido, diseñado por Linus Torvalds. Está pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando estas 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.



## 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 <Nombre como se verá en los commits>`: 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`: todo lo que hemos hecho históricamente.
- `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`: comparar directorio con standing.
- `git diff <commit1> <commit2>`: compara diferencias entre en cambios confirmados.

# GIT



## Volver en el tiempo con branches y checkout

- `git reset <commit> --soft/hard`: regresa al commit especificado, eliminando todos los cambios que se hicieron después de ese commit.
- `git checkout <commit/branch> <filename>`: 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 - <filePath>`: deshacer cambios en un archivo en estado modified (que ni fue agregado a staging)

## Git rm y git reset. Comandos para corrección en Github

`Git rm`: Este comando nos ayuda a eliminar archivos de Git sin eliminar su historial del sistema de versiones.

- `git rm --cached <archivo/s>`: elimina los archivos del área de Staging y del próximo commit, pero los mantiene en nuestro disco duro.
- `git rm --force <archivo/s>`: 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

`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 <new branch>`: crea una nueva rama.
- `git checkout <branch name>`: se mueve a la rama especificada.
- `git merge <branch name>`: fusiona la rama actual con la rama especificada y produce un nuevo commit de esta fusión. `git merge --abort`: cancelar un merge.
- `git branch`: lista las ramas generadas. `git branch -d`: borrar ramas.
- `git fetch`: traer actualizaciones del servidor remoto y guárdalas en nuestro repositorio local.
- `git pull`: fetch y merge al mismo tiempo.



# ¿QUÉ ES GIT?



**Git** es un software de control de versiones diseñado por **Linus Torvalds**, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.

Sistema operativo: Unix-like, Windows, Linux

Programado en: C, Bourne Shell, Perl

Modelo de desarrollo: Software libre

Escrito en: C, Perl, Tcl, Python

## Importante

### Directarios en Git

Es el lugar donde se almacenan los metadatos y las bases de datos para nuestros proyectos, y es justamente lo que se copia cuando clonamos de un ordenador a otro los archivos.

**GitHub** es una forja para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de ordenador. El software que opera GitHub fue escrito en **Ruby on Rails**. Desde enero de 2010, GitHub opera bajo el nombre de GitHub, Inc.

## GIT

Git es un sistema de control de versiones que originalmente fue diseñado para operar en un entorno Linux. Actualmente Git es multiplataforma, es decir, es compatible con Linux, MacOS y Windows.

### Características de Git

- Git almacena la información como un conjunto de archivos.
- No existen cambios, corrupción en archivos o cualquier alteración sin que Git lo sepa.
- Casi todo en Git es local. Es difícil que se necesiten recursos o información externos, basta con los recursos locales con los que cuenta.
- Git cuenta con 3 estados en los que podemos localizar nuestros archivos: Staged, Modified y Committed

## GITHUB

GitHub es un servicio de alojamiento que ofrece a los desarrolladores repositorios de software usando el sistema de control de versiones, Git.



### Características de Github

- GitHub permite que alojemos proyectos en repositorios de forma gratuita y pública, pero tiene una forma de pago para privados.
- Puedes compartir tus proyectos de una forma mucho más fácil.
- Te permite colaborar para mejorar los proyectos de otros y a otros mejorar o aportar a los tuyos.
- Ayuda 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 nuestros proyectos.

En vez de guardar un mismo archivo varias veces. Git nos ayuda a guardar solo los cambios del mismo, además maneja los cambios que otras personas hagan sobre los mismos archivos, así múltiples personas pueden trabajar en un mismo proyecto sin conflictos. Git permite rastrear que miembro realiza los cambios, además de recuperar una versión antigua de manera precisa. Github nos permite publicar un repositorio para trabajar de forma remota y colaborar con otros miembros dentro y/o fuera de nuestra organización.



git  
Portzi

## Editor de texto o IDE

Es una herramienta que nos brinda muchas ayudas para escribir código, algo así como un bloc de notas muy avanzado.

### 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.

### Más utilizados





# Conceptos importantes de Git

## Bug

Error en el código

## Diff

Se utiliza para mostrar los cambios entre dos versiones del mismo archivo.

## 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.

## 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.

## 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.

## 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”

## 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.

## 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.

## 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.

## Branch

Es una bifurcación del proyecto que se está realizando para anexar una nueva funcionalidad o corregir un bug.

## Repositorio

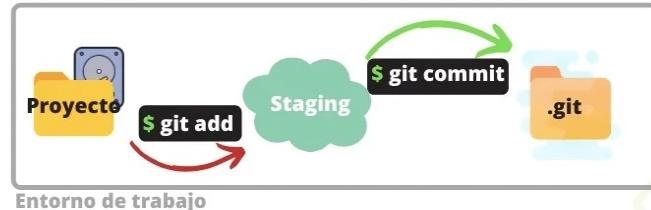
Donde se almacena todo el proyecto, el cual puede vivir tanto en local como en remoto. Guarda un historial de versiones y, más importante, de la relación de cada versión con la anterior para que pueda hacerse el árbol de versiones con las diferentes ramas.

## 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.

## ¿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.



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.

### Archivos unstaged

Entiendelos 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 staged

Son archivos en staging. Viven dentro de git y hay registro de ellos porque han sido afectados por el comando `git add`, aunque no sus últimos cambios. Git ya sabe de la existencia de estos últimos cambios, pero todavía no han sido guardados definitivamente en el repositorio porque falta ejecutar el comando `git commit`.

### Archivos 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.

# Flujo basico de Trabajo



## Working directory

**Git init**

Crea un  
repositorio  
desde 0

**Git clone**

Clonar  
repositorio  
existente

**Git add**

La programadora trabaja en su  
repositorio y envia los commit

## Staging Area

Área de preparación, es un  
espacio de borrador, donde se  
verifica el commit

**Git commit**

## Repositorio remoto

Si usamos git clone

**Git push**

## Repository



# Cómo funcionan las ramas en GIT

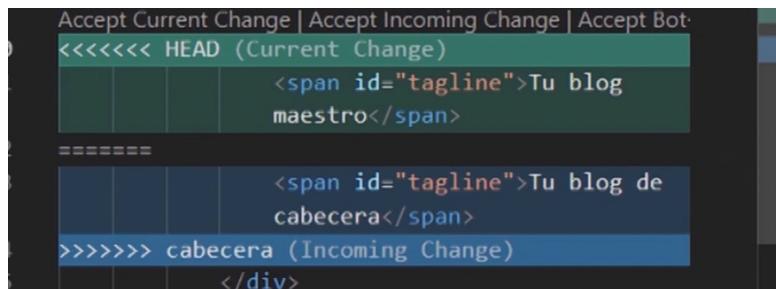
Las ramas son la manera de hacer cambios en nuestro proyecto sin afectar el flujo de trabajo de la rama principal. Esto porque queremos trabajar una parte muy específica de la aplicación o simplemente experimentar.

- 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 brach 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.

## Solución conflictos desde visual code

Accept Current Change: aceptar el cambio actual (HEAD)

Accept Incoming Change: o el cambio que bien de la rama a la que Hulu so el merch



```
Accept Current Change | Accept Incoming Change | Accept Both
<<<<<< HEAD (Current Change)
Tu blog
maestro
-----
>>>>> cabecera (Incoming Change)
Tu blog de
cabecera
```

La rama “master” paso a llamarse rama “main”

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).

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.

## Trabajando con repositorios remotos en GitHub

Cómo conectar un repositorio de GitHub a nuestro documento local

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:

**CAMBIOS IMPORTANTES**

```
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
```

### CAMBIOS IMPORTANTES

git push origin master

Estamos diciéndole a git que envíe a origin(remoto) la rama 'master' de nuestro repositorio local. Por lo tanto, en GitHub se interpreta como adicionar una rama independiente llamada 'master' con su contenido a bordo, pero no se carga en la rama default de GitHub, debido a que su rama default ahora se llama 'main'.

Para alinear el contenido de Freedy y la actualización de GitHub, en la práctica el comando para realizar el push según el nuevo estándar sería:

git push origin master:main

En donde le estamos diciéndole a git que envíe a origin(remoto) la rama 'master' de nuestro repositorio local hasta la rama 'main' del servidor remoto. De igual manera, dentro de esta nueva lógica:

git pull origin main --allow-unrelated-histories

En donde estamos trayendo desde el servidor remoto la rama 'main' fusionando las historias.

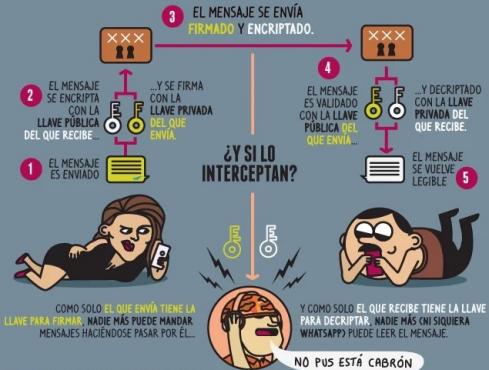
De tal manera que ya podemos continuar con el contenido de Freedy sin llegar a confundirnos.

Al final volvemos a hacer un git push origin master:main para enviar todo y poder visualizarlo en el Github

# Como funcionan las llaves públicas y privadas

## ¿QUÉ TAN FÁCIL (O DIFÍCIL) ES QUE SE FILTREN TUS CONVERSACIONES DEL WHATS\*?

MUY MUY DIFÍCIL. ASÍ ES COMO FUNCIONA:



Generar una nueva llave SSH: (Cualquier sistema operativo)

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

Comprobar proceso y agregarlo (Windows)

- eval \$(ssh-agent -s)
- ssh-add ~/.ssh/id\_rsa

Comprobar proceso y agregarlo (Mac)

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

En mi caso en el proceso de comprobar tube un error aquí encuentre solución:

<http://ligeratecademy.com/powershell/startingssh-agent-on-windows-10-fails-unable-to-start-ssh-agent-service-error-1058/>

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.

¿Private key? ¿Public key?



Para enviar.  
Nunca debes  
compartirla.



Para recibir.  
Está bien que  
la compartas.

## ¿CÓMO FUNCIONA EL CIFRADO ASIMÉTRICO?



¿Usas macOS Sierra 10.12.2 o superior?

Haz lo siguiente:

- cd ~/.ssh
- Crea un archivo config...
- Con Vim vim config
- Con VSCode code config
- Pega la siguiente configuración en el archivo...

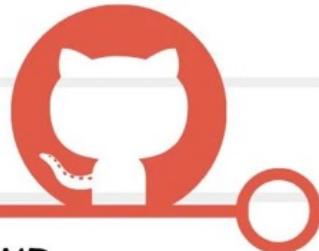
Host \*

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

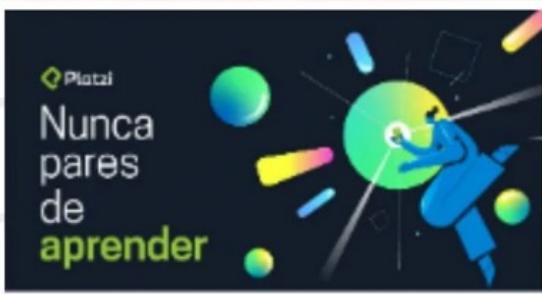
Agrega tu llave

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

# CONEXIÓN A GITHUB CON SSH



Tutorial recomendado



Configurar llaves SSH en Git y GitHub

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

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

ssh -T git@github.com  
#Comando para probar la conexión con Github, se ingresa con la passphrase adicional si se agregó.

git config --global user.email "email"

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

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

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

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

## • AGREGAR UNA CLAVE SSH NUEVA A TU CUENTA DE GITHUB

- 1) Ubicar la carpeta .ssh oculta, abrir el archivo en tu editor de texto favorito, y copiarlo en tu portapapeles.
- 2) En la esquina superior derecha de tu cuenta en Github.com, da clic en tu foto de perfil y después da clic en Configuración.
- 3) En la barra lateral de configuración de usuario, da clic en Llaves SSH y GPG.
- 4) Haz clic en New SSH key (Nueva clave SSH)
  - o Add SSH key (Agregar clave SSH).
- 5) En el campo "Title" (Título), agrega una etiqueta descriptiva para la clave nueva. Por ejemplo, si estás usando tu Mac personal, es posible que llames a esta "Personal MacBook Air". Copia tu clave en el campo "Key" (Clave).
- 6) Haz clic en Add SSH key (Agregar tecla SSH).
- 7) Si se te solicita, confirma tu contraseña GitHub.

## • CAMBIAR LA URL DE UN REMOTO

- 1) Abre la Terminal y cambiar el directorio de trabajo actual en tu proyecto local.
- 2) Enumerar tus remotos existentes a fin de obtener el nombre de los remotos que deseas cambiar.
- 3) Cambiar tu URL remota de HTTPS a SSH con el comando git remote set-url
- 4) Verificar que la URL remota ha cambiado.

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

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

```
Proyecto git/master
> git pull
The authenticity of host 'github.com (140.82.114.3)' cant be established.
RSA key fingerprint is SHA256:TbBg6XUpWg17EII...
Are you sure you want to continue connecting (yes/no/[fingerprint])? y
Please type 'yes', 'no' or the fingerprint: yes
Warning: Permanently added 'github.com,140.82.114.3' (RSA) to the list of known hosts.
No hay información de rastreo para la rama actual.
Por favor especifique a qué rama queres fusionar.
Ver git-pull(1) para detalles.

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

Proyecto git/master*
> git pull origin master
Warning: Permanently added the RSA host key for IP address '140.82.114.3' to the list of known hosts.
Desde github.com:ayenque/hyperblog
 * branch    master      -> FETCH_HEAD
 Ya está actualizado.
```

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

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

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

```
Proyecto git/master*
> git commit -am "Una versión del Hyperblog"
[master 38168c] Una versión del Hyperblog
 1 file changed, 1 insertion(+), 1 deletion(-)

Proyecto git/master
> git pull origin master
Enumerando objetos: 5, listo.
Contando objetos: 100% (5/5), listo.
Comprimiendo objetos: 100% (3/3), listo.
Escribiendo objetos: 100% (3/3), listo.
Total 3 (delta 2), reusado 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:ayenque/hyperblog.git
 7771e35..c38168c master -> master

Proyecto git/master
```

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



## Tags y versiones

Los tags o etiquetas nos permiten asignar versiones a los commits con cambios más importantes o significativos de nuestro proyecto.

Comandos para trabajar con etiquetas:

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

Git nos brinda la posibilidad de poner etiquetas a los commits, esto es útil para tener una referencia histórica de nuestro proyecto.

Ejemplo

`Commit 10 (v 0.1) "primera parte del proyecto"`

## Comandos para el uso de branch

Crear un nuevo tag y asignarlo a un commit

`git tag -a nombre-del-tag -m "mensaje del tag" [id del commit]`

Borrar un tag en el repositorio local:

`git tag -d nombre-del-tag`

Listar los tags

`git tag`

`git show-ref --tags`

Publicar un tag en el repositorio remoto:

`git push origin --tags`

Borrar un tag del repositorio remoto:

`git push origin :refs/tags/nombre-del-tag`

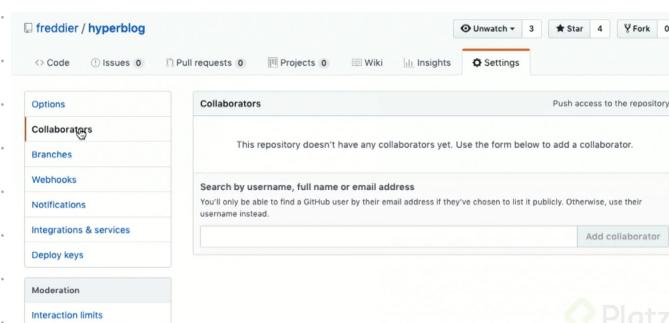
## Cómo agregar colaboradores en Github

- Solo debemos entrar a la configuración de colaboradores de nuestro proyecto.

Se encuentra en:

Repositorio > Settings > Collaborators

Aquí, 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:

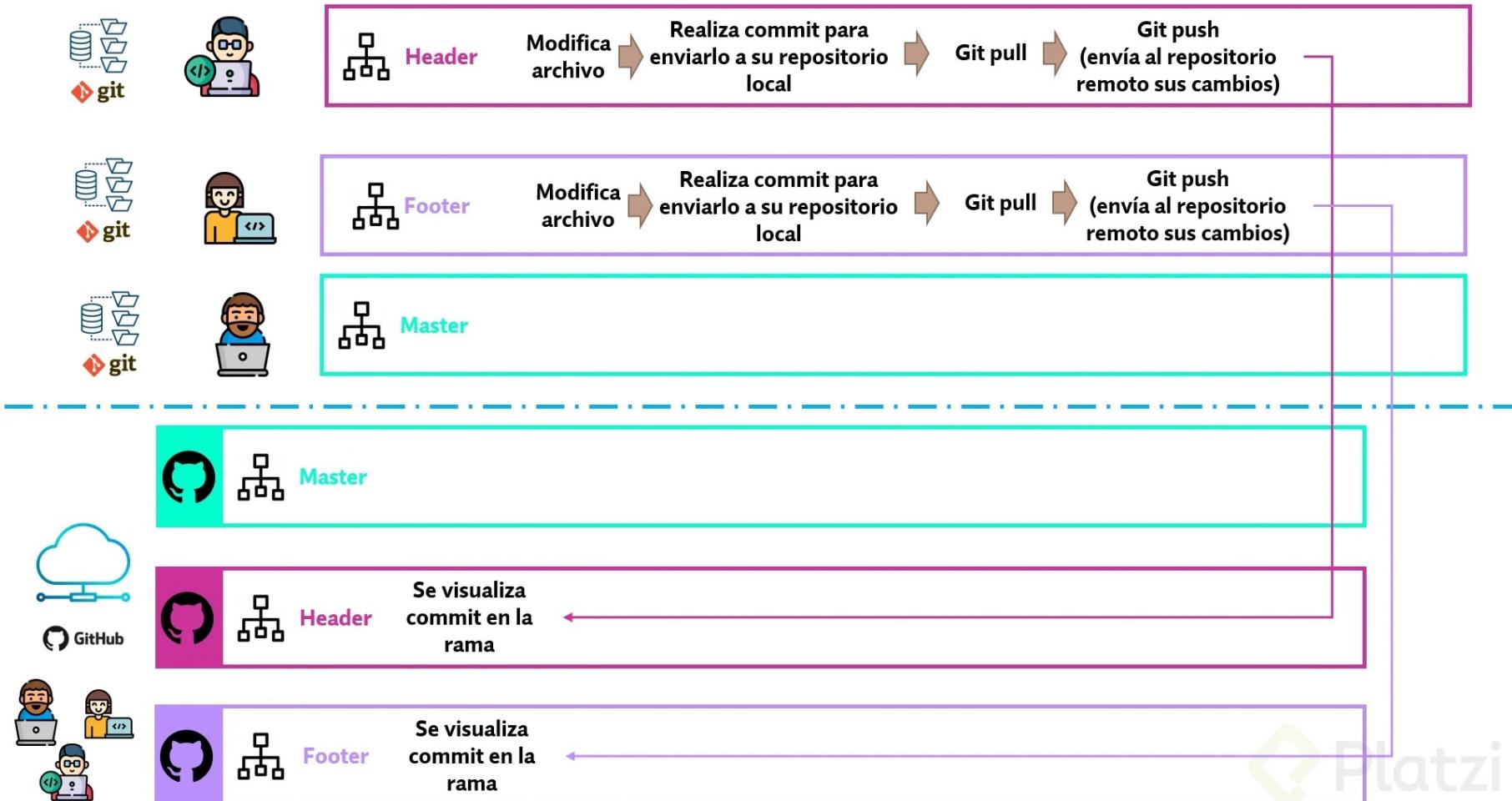
- Hacer un commit con el nuevo mensaje que queremos, esto nos abre el editor de texto de la terminal:  
`git commit —amend`
- Corregimos el mensaje
- Traer el repositorio remoto  
`git pull origin master`
- Ejecutar el cambio  
`git push --set-upstream origin master`

## Cómo agregar un alias solo para git

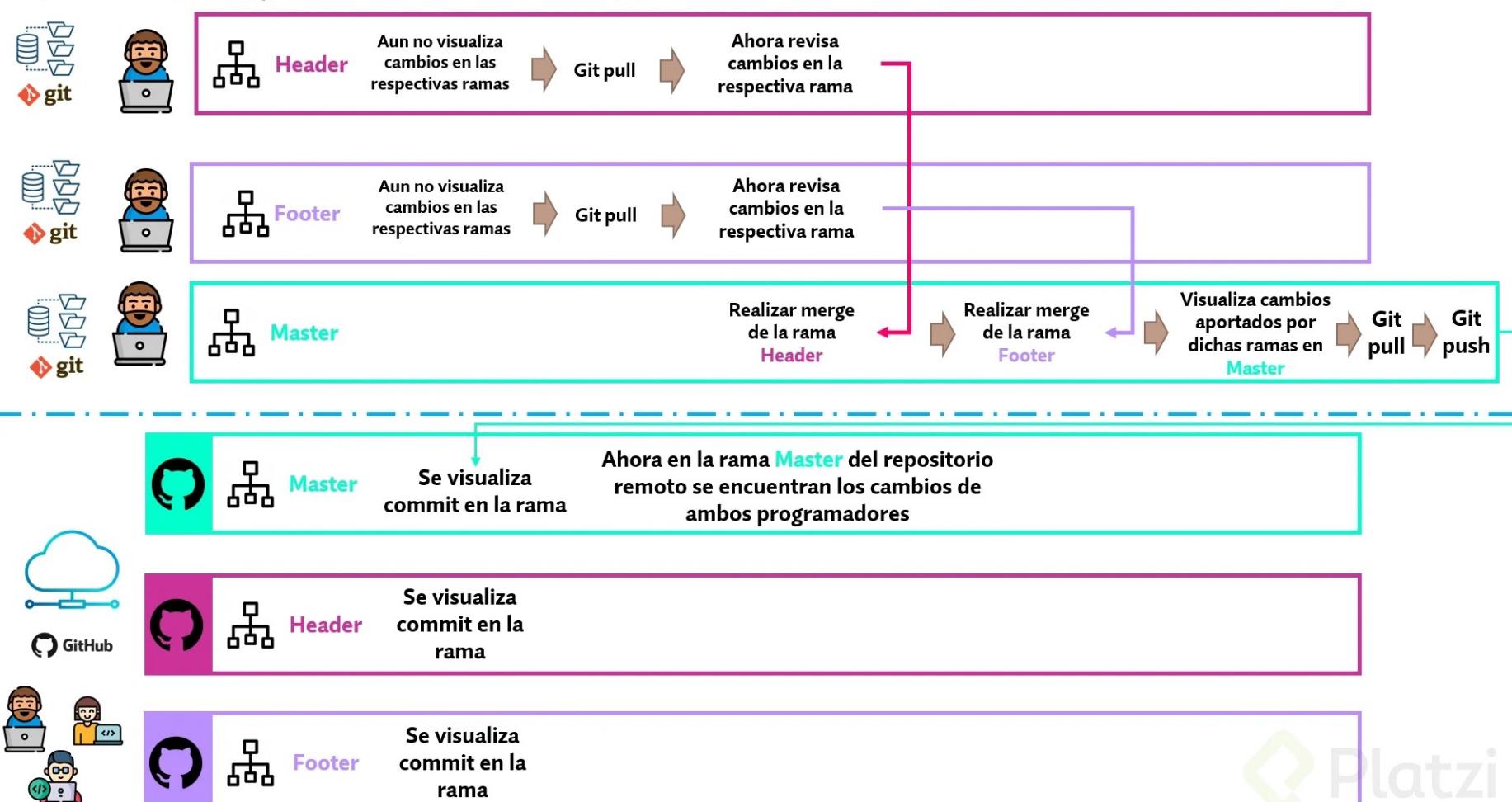
- Para un proyecto:  
`git config alias.arbolito "log --all --graph --decorate --oneline"`
- Global:  
`git config --global alias.arbolito "log --all --graph --decorate --oneline"`
- Para correrlo:  
`git arbolito`

# Flujos de trabajo profesionales

## FLUJO DE TRABAJO EN EQUIPO DENTRO DE UNA EMPRESA (1º)



## FLUJO DE TRABAJO EN EQUIPO DENTRO DE UNA EMPRESA (2º)



# PULL REQUESTS

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

## UTILIZANDO PULL REQUESTS EN GITHUB

Cuando un desarrollador termina de crear (y probar) ya sea una nueva funcionalidad o corrección de bug, solicita integrar su desarrollo al repositorio principal. Esta solicitud se le conoce como pull request (o PR).

**No Olvidar traer los cambios (Git Pull) y enviar los cambios (Git Push) siempre como buena práctica!**

Es una buena práctica crear una nueva rama, cada vez que necesitamos corregir un error puntual.

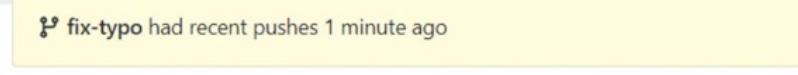
El nombre del Pull Request toma el nombre del ultimo commit.

Una vez aprobado el pull request, el devops u otro integrante puede unir el nuevo desarrollo al desarrollo principal.

Recordar que los pull request no existen del lado de Git, es algo propio de Github, por lo tanto en Git no queda registrado las acciones que realizamos en los Pull Request

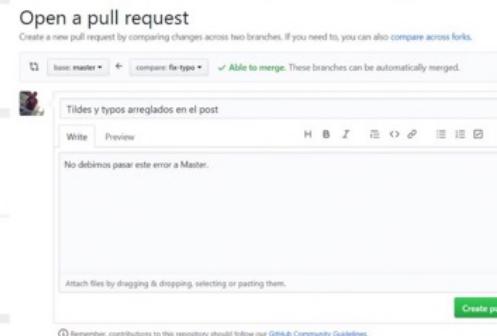
Una vez realizado el Merge, es importante traer los cambios en Git (Git Pull), en caso borramos la rama en Github, podemos borrarla en Git con `git branch -D nombre-rama`

a) Una vez realizada la corrección y enviado los cambios de la rama, Github nos alerta de un push reciente y nos da la opción para comparar y crear un Pull Request

 fix-type had recent pushes 1 minute ago

[Compare & pull request >](#)

b) Para crear un pull request debemos dar click en "Compare & pull request"



c) Una vez creado el pull request, al administrador recibirá una notificación:

Recent activity

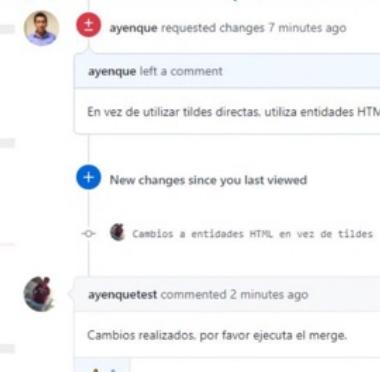
 Tildes y typos arreglados en el post

ayenque/hyperblog · Your review was requested 3 days ago

d) El administrador puede comparar los cambios haciendo click en **Files changed**, y consultar con el equipo para la revisión respectiva.

 Tildes y typos arreglados en el post #1

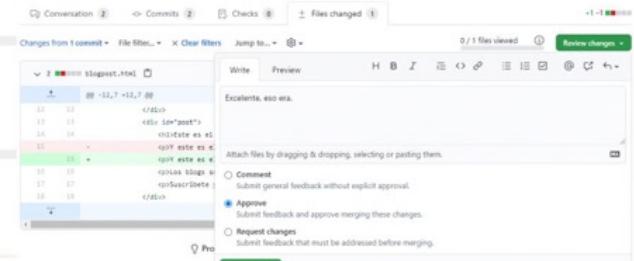
g) Una vez realizados los cambios requeridos responder el Pull Request



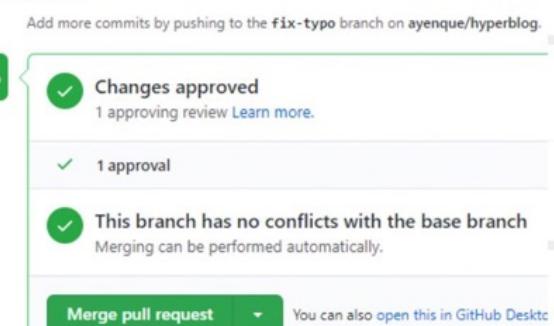
h) Podemos aprobar los cambios, en **Review Changes** y **Approve**

 Tildes y typos arreglados en el post #1

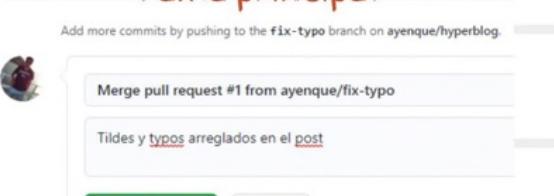
ayenquetest wants to merge 2 commits into master from fix-type



i) El solicitante recibe una notificación de aprobación de su solicitud.



j) Finalmente, el administrador o el Devops debe hacer el **merge** con la rama principal

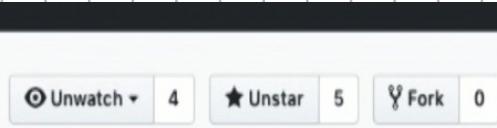


k) Si se requiere, podemos eliminar la rama ya que fue creada solo para solucionar un error (**Delete Branch**)



## FORK

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.



Unwatch/watch para recibir notificaciones de este repositorio.  
Star/Unstar aviso cuando algo cambie.

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

Al hacer clic en **fork** automáticamente trae el proyecto a mi Github y se convierte en un repositorio propio sin perder la historia de este.

Para subir cambios al repositorio original:

- 1 Ubicar el repositorio del proyecto open source con el cual quiero colaborar.
  - Watch → Watching
  - Star
  - Fork: Tomar una copia del estado actual del proyecto y clonarlo, y queda como un proyecto mio.
- 2 Clonamos el repositorio en nuestro local.
- 3 Realizamos los cambios necesarios.
- 4 New Pull Request.

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>  nombre_del_remoto
→ upstream (puede ser cualquier nombre)
```

```
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.

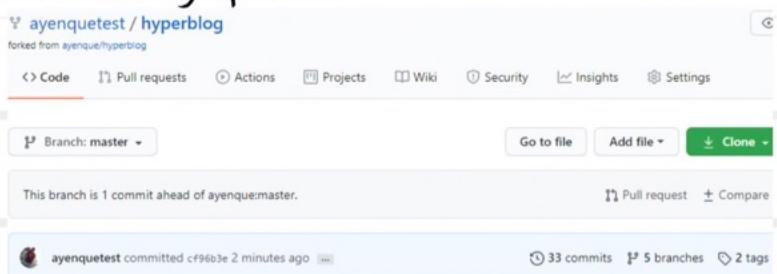
```
git pull upstream master
git push origin master
```

Upstream: repositorio adicional, que es un apuntador al repositorio original.

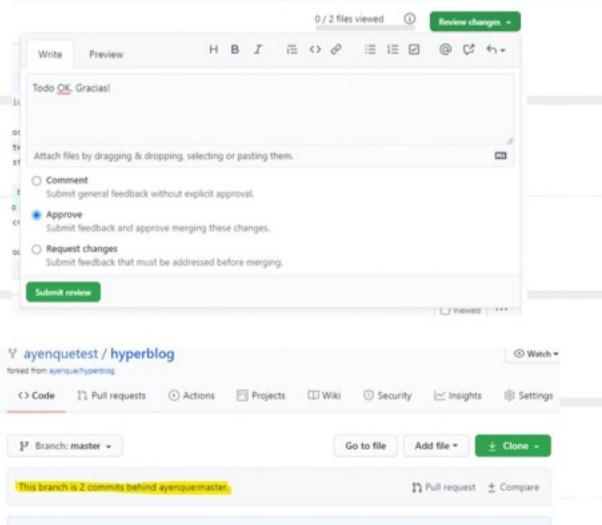
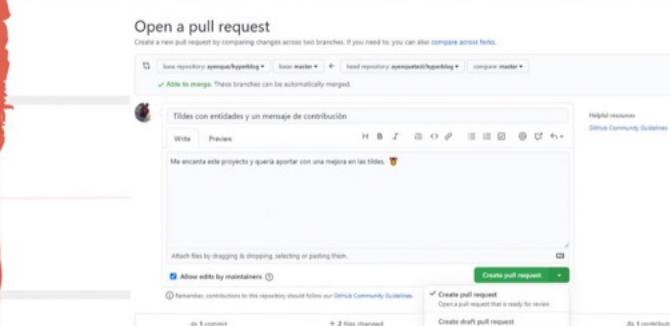
Cuando desde local le hacemos pull a este repositorio nos trae todo actualizado y luego con un push lo mandamos a nuestra rama.

**En unos segundos te redireccionarán a una página nueva de proyecto, en tu cuenta y con tu propia copia del código fuente.**

Con esto podemos clonar el proyecto a nuestro repositorio local (**git clone**) y podemos realizar cualquier cambio y aporte.



Para que nuestros cambios se fusionen en el repositorio remoto original, podemos hacer un **pull request** y esperar que se realice el **merge** con el original.



Github nos advierte, si la rama del repositorio original tiene commits, que el repositorio forkeado no.

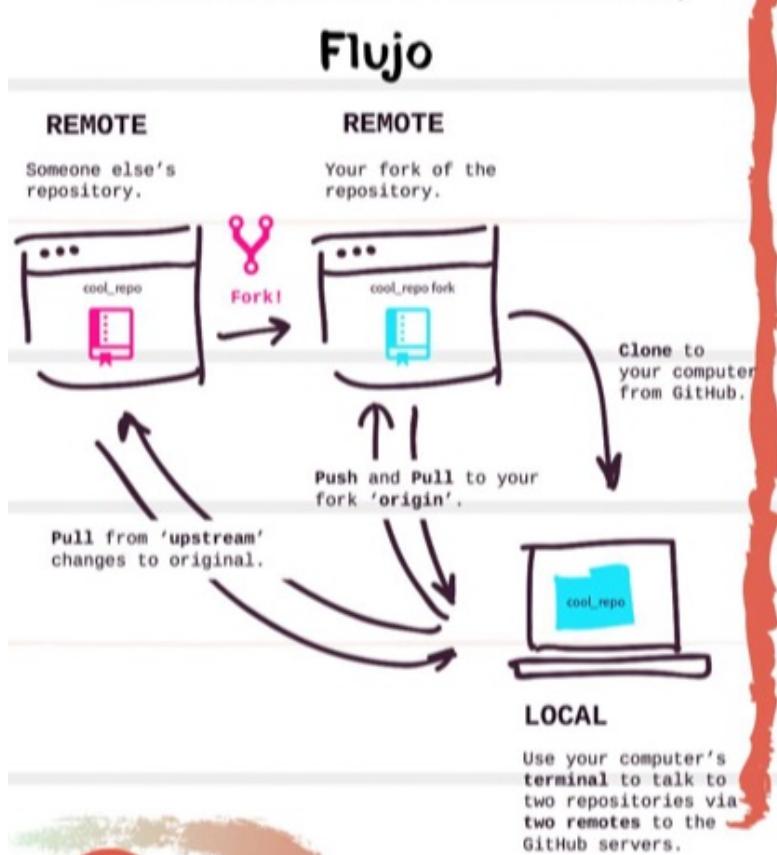
Para traer estos cambios al repositorio forkeado, debemos agregar una nueva fuente remota:

```
git remote add upstream url-repositorio-original
```

Luego traemos los cambios del original, para actualizar en el repositorio forkeado:

```
git pull upstream master # traemos cambios del repositorio original
```

```
git push origin master # enviamos cambios a nuestro repositorio forkeado
```



## DEPLOY

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

.Pasos para hacer deployment en un servidor web:

- Entrar a la carpeta 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 actualizaciones para el servidor en la carpeta de los archivos del servidor.  
`git pull <origin> <master>`

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 Jenkins (Open source).

## .GITIGNORE

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

### Buena práctica

Evitar que los archivos binarios del contenido sean parte de un repositorio.

### Subir imágenes

- imgur
- ftp
- content delivery network

### Buscar inspiración en otros proyectos

- laravel/laravel
- archivo .gitignore
- Vuejs/vue
- TryGhost / Ghost
- arduino/Arduino

### ¿Archivo .gitignore?

Es una lista de los archivos que vamos a ignorar

\* Significa todo tipo de archivos.

```
▷ .gitignore ●
```

```
C: > Users > usuario  
1   *.jpg  
2   *.png
```

### Probemos como funciona

#### Descargar una imagen

- Guardar imagen proyecto1/imágenes algo.jpg

### Visual Studio Code



#### Agregamos imagen al blogpost

- <p></p>

#### También:

- <p></p>

#### Crear archivo para ignorar cosas

- Control N -> Control S -> proyecto1/.gitignore

#### Agregamos archivos a ignorar

- .gitignore
  - \*.jpg -> Esto va a ignorar el 100% de los jpg

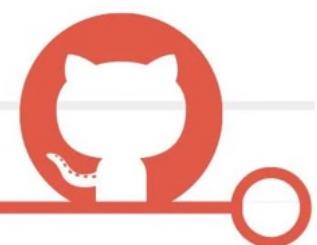
### Git



- git status
- git add .gitignore
- git commit -am "Agregué una imagen al blog"
- git status
- git pull origin master
- git push origin master

**GitHub**   
Vemos los cambios

# README.MD ES UNA EXCELENTE PRÁCTICA



README puede estar en varios formatos. Puede estar con el nombre README, README.md, README.asciidoc y alguno más.

README.md, es el más común. "md" significa Markdown, que es una especie de codificación que te permite cambiar la manera en que se ve un archivo de texto.



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

Editor.md

开源在线 Markdown 编辑器

<https://pandao.github.io/editor.md/>

**Editor.md | Un editor en línea recomendado que nos ayuda a editar nuestro README.md**

Podemos inspirarnos en repositorios open source, ya que es la mejor forma de aprender!



**README** brinda una guía rápida a los que acaban de descubrir nuestro proyecto de cómo empezar a usarlo. Es muy importante que vaya al grano, sea conciso y muy claro. Para extendernos y entrar en detalles está la documentación, a la que siempre podemos enlazar desde dentro de nuestro README.

Puedes agregar un archivo **README** a tu repositorio para comentarle a otras personas por qué tu proyecto es útil, qué pueden hacer con tu proyecto y cómo lo pueden usar.

Un archivo **README** suele ser el primer elemento que verá un visitante cuando entre a tu repositorio.

Esto incluye normalmente cosas como:

- **Para qué** es el proyecto
- **Cómo** se configura y se instala
- **Ejemplo** de uso
- **Licencia** del código del proyecto
- **Cómo** **participar** en su desarrollo



Ejemplos de un archivo README: <https://github.com/vuejs/vue/blob/dev/README.md>



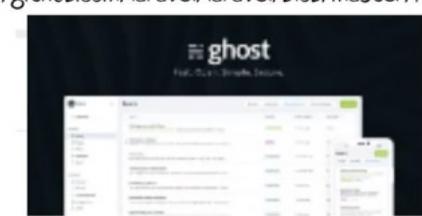
<https://github.com/vuejs/vue/blob/master/README.md>



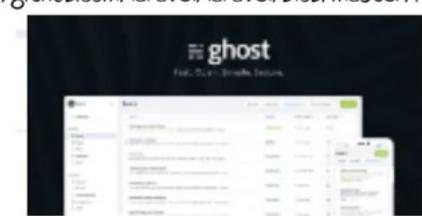
**laravel/laravel**

A PHP framework for web artisans. Contribute to laravel/laravel...

[github.com/laravel/laravel](https://github.com/laravel/laravel)



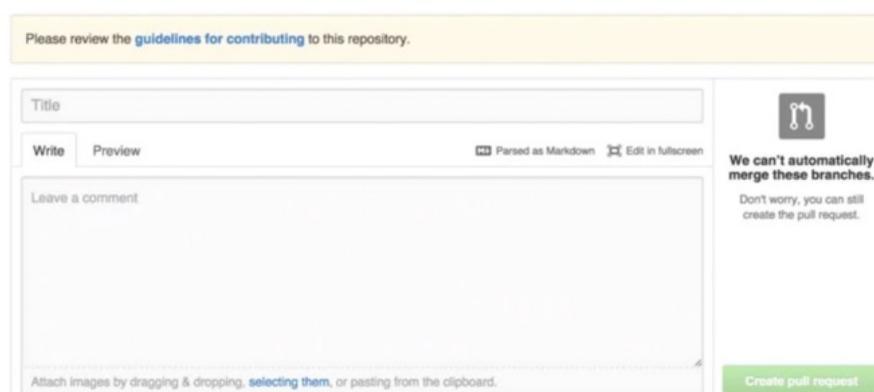
<https://github.com/laravel/laravel/blob/master/README.md>



<https://github.com/TryGhost/Ghost/blob/master/README.md>

## CONTRIBUTING

El otro archivo que GitHub reconoce es **CONTRIBUTING**. Si tienes un archivo con ese nombre y cualquier extensión, GitHub mostrará algo como la imagen, cuando se intente abrir un Pull Request.



La idea es que indiques cosas a considerar a la hora de recibir un Pull Request. La gente lo debe leer a modo de guía sobre cómo abrir la petición.



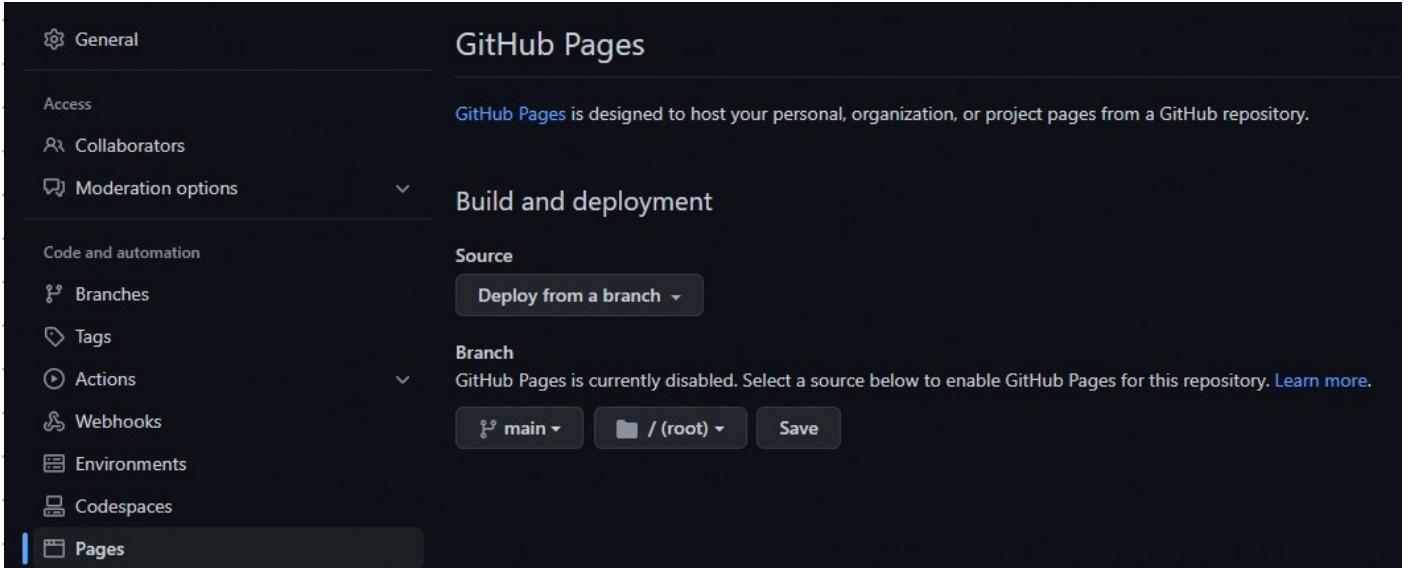
# GITHUB PAGES

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.

Recomiendo seguir 'tutorial proporcionado por GITHUB PAGES para creación del repositorio Pages.Github.com'

Luego de seguir el tutorial, sea para terminal o Github debemos ir a las configuraciones y en una de las actualizaciones recientes al me momento que hago esto es que, ahora en el costado izquierdo hay un menú donde encontraremos "pages" al dar clic configuramos la rama de la cual queremos que se publiquen el repositorio.



En caso sea el primer repositorio:

<https://github.com/new>  
Crear el repositorio

Create a new repository  
A repository contains all project files, including the revision history.  
Owner: ayenque Repository name: sophshep.github.io  
Great repository names are short and memorable. Need inspiration? How about fi  
Description (optional)

Clonar el repositorio

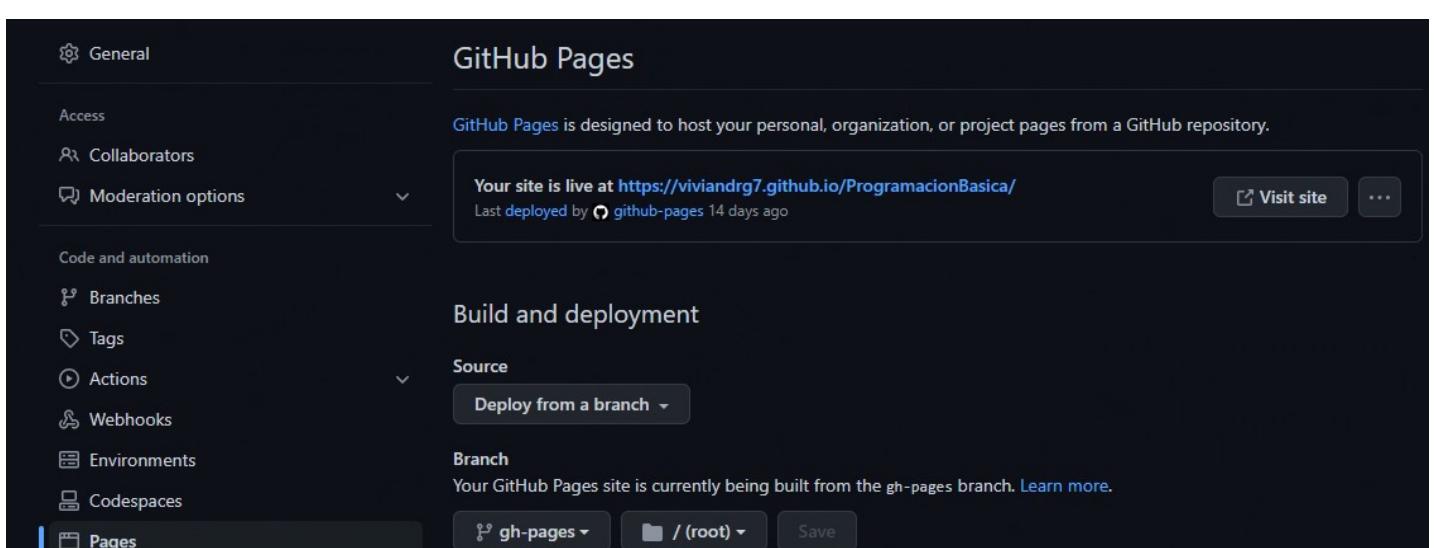
`git clone https://github.com/username/username.github.io`

Ingresar a la carpeta del proyecto y agregar un archivo index.html

`cd username.github.io`  
`echo "Hello World" > index.html`

Ejecutar los comando  
git add, git commit, y git push para aplicar los cambios:

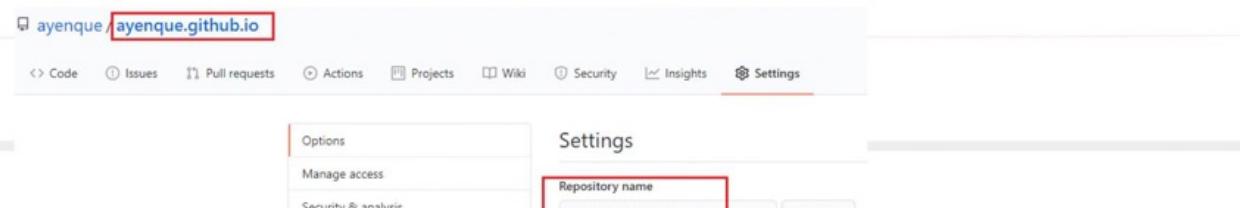
`git add --all`  
`git commit -m "Initial commit"`  
`git push -u origin master`



<https://viviandrg7.github.io/ProgramacionBasica/parte2/Index.html> ese es el enlace a mi proyecto, creado en el curso de programación básica de Platzi.

## GITHUB PAGES PERSONAL

En caso se desea obtener una URL de forma «[usuario.github.io](https://usuario.github.io)», se debe crear un repositorio con el mismo nombre:



Se clona el repositorio y se carga los archivos para agregar una página personal.

No olvidar agregar el archivo "index.html"

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/ayenque.github.io (master)
$ ls -al
total 10
drwxr-xr-x 1 ayenq 197609 0 Aug 31 17:00 .
drwxr-xr-x 1 ayenq 197609 0 Aug 31 17:00 ..
drwxr-xr-x 1 ayenq 197609 0 Aug 31 17:00 .git/
-rw-r--r-- 1 ayenq 197609 406 Aug 31 17:00 index.html
-rw-r--r-- 1 ayenq 197609 64 Aug 31 17:00 README.md
```

Este es un sitio web con GitHub Pages

Esto es un test para el curso de Git y Github



# Multiples entornos de trabajo en Git

## GIT REBASE

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.

### Rebase vs Merge



#### 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.

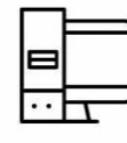
### #3. Conflict Handling

#### 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

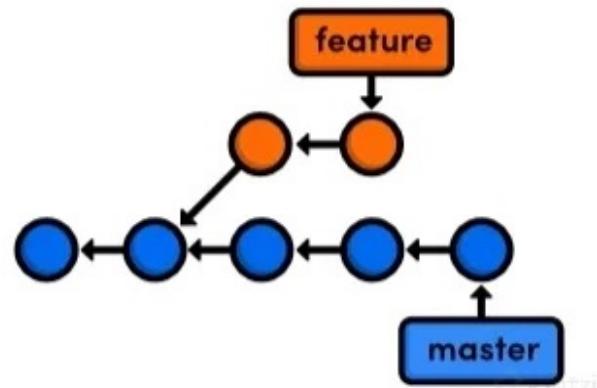


Commits that were once reachable will no longer be reachable after rebase since commit history is changed.

#### Merge



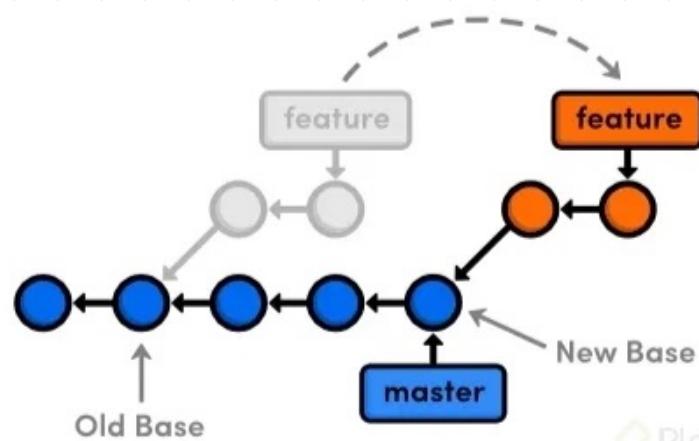
Commits will remain reachable from the source branches.



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
```

# GIT STASH

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

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

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

## 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

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

Para agregar los cambios al stash se utiliza el comando:

```
git stash
```

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

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

## Obtener elementos del stash

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

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

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

```
git stash pop
```

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

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

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

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

Donde el <num\_stash> lo obtienes desde el git stash list

## Listado de elementos en el stash

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

```
git stash list
```

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

## Crear una rama con el stash

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

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

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

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

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

## Eliminar elementos del stash

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

```
git stash drop
```

Pero si en cambio conoces el índice del stash que quieras 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
```

## GIT CLEAN

limpiar el proyecto de archivos no deseados

Permite borrar archivos innecesarios que no estén agregados a un commit, como por ejemplo archivos duplicados, que hayan sido agregados por nosotros recientemente.

Comandos utiles de git clean:

git clean --dry-run # Con este comando es como una simulación, donde te avisa que archivo va a eliminar si aplicas un clean completo.

git clean -n #Versión abreviada de git --dry-run

git clean -f #Eliminar todos los archivos untracked listados ahora si (que no sean carpetas o archivos ignorados)

git clean -f -d # Eliminar los archivos untracked listados, agregando el -d le decimos que también las carpetas que se vuelvan vacias como resultado de eliminar los archivos con el argumento de -f

git clean -x # Borra las copias incluyendo las que están dentro de un .gitignore

git clean -X # Borra solamente los archivos ignorados por git

git clean -q # Muestra los errores que tuvo la ejecución, pero no los archivos que fueron borrados

git clean -i #Modo interactivo de git clean

Para hacer simulaciones:

git clean -fn //Archivos

git clean -dn //Directorios

git clean -xn //Archivos ignorados

Para eliminar de verdad:

git clean -f //Archivos

git clean -df //Directorios

git clean -xf //Archivos ignorados

También se pueden combinar:

git clean -xdf //Eliminar los tres tipos de archivos juntos.

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

## GIT CHERRY-PICK

Es un comando que permite tomar uno o varios commits de otra rama sin tener que hacer un merge completo. Así, gracias a cherry-pick, podríamos aplicar los commits relacionados con nuestra funcionalidad en la rama master sin necesidad de hacer un merge.

Para demostrar cómo utilizar git cherry-pick, supongamos que tenemos un repositorio con el siguiente estado de rama:

a -b - c - d Master  
  |\  
  e - f - g Feature

El uso de git cherry-pick es sencillo y se puede ejecutar de la siguiente manera:

git checkout master

En este ejemplo, commitSha es una referencia de confirmación. Puedes encontrar una referencia de confirmación utilizando el comando git log. En este caso, imaginemos que queremos utilizar la confirmación 'f' en la rama master. Para ello, primero debemos asegurarnos de que estamos trabajando con esa rama master:

git cherry-pick f

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

a -b - c - d - f Master  
  |\  
  e - f - g Feature

La confirmación f se ha introducido con éxito en la rama de funcionalidad

### Atención

Cherry-pick es una mala práctica porque significa que estamos reconstruyendo la historia, usa cherry-pick con sabiduría. Si no sabes lo que estás haciendo, mejor evita emplear este comando.

# Comandos de Git para casos de emergencia

## GIT RESET Y REFLOG; ÚSESE EN CASO DE EMERGENCIA



El reflog no forma parte del repositorio en sí (se almacena por separado) y no se incluye en los push, búsquedas o clones; es puramente local.

Por defecto git guarda las referencias de los últimos 90 días. Se puede cambiar este valor usando el comando

`git reflog expire --expire=[tiempo]`

Ejecutar git reset es equivalente a ejecutar git reset --mixed HEAD (se puede usar cualquier hash de confirmación de Git)

EN TODAS LAS OPCIONES DE GIT RESET, LA PRIMERA ACCIÓN QUE SE REALIZA, ES RESTABLECER EL ÁRBOL DE CONFIRMACIONES AL HEAD O HASH INDICADO.

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

Con **Git Reset** puedo restablecer el apuntador del HEAD y aunque en el log pareciera que la historia de commits efectivamente cambió, en Git Reflog siempre se registra todo, incluso los "borrados".

### • GIT REFLOG

Una de las cosas que Git hace en segundo plano, mientras tu estás trabajando a distancia, es mantener un "reflog" - un log dónde se apuntan las referencias de tu HEAD y tu rama en los últimos meses.

• `git reflog show --all`

Comando para obtener un "reflog" de todas las referencias.

• `git reflog show nombre-rama`

Comando para ver el registro de referencia de una rama concreta.

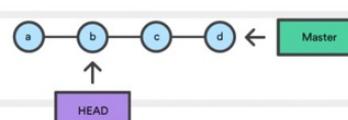
• `git reflog stash`

Comando para ver el registro de referencia de un stash.

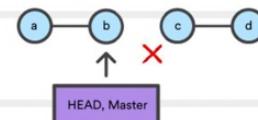
### • GIT RESET

Git reset tiene un comportamiento similar a git checkout. Mientras que git checkout solo opera en el puntero de referencia HEAD, **git reset** moverá el puntero de referencia HEAD y el puntero de referencia de la rama actual.

#### git checkout



#### git reset



• `git reset --hard [HEAD/HASH]`

Esta es la opción más directa, PELIGROSA y que se usa más frecuentemente. Si ejecutamos este comando, pero tengo cambios en el **Working Directory** (sin git add) y/o si tengo agregado archivos o modificaciones en el **Staging Area** (con git add) **se pierden**. Esta pérdida de datos no se puede deshacer.

• `git reset --mixed [HEAD/HASH]`

Si ejecutamos este comando, pero tengo agregado archivos o modificaciones en el **Staging Area** (con git add) **se mueven** al **Working Directory**. Lo que esté en **Working Directory** sigue allí.

• `git reset --soft [HEAD/HASH]`

Si ejecutamos este comando pero tengo cambios en el **Working Directory** (sin git add) y/o si tengo agregado archivos o modificaciones en el **Staging Area** (con git add) **quedan en su mismo estado**.

## Git Amend

`git add -A` :  
# Para hacer uso de amend los archivos deben de estar en staging

`git commit --amend` :  
Remendar último commit

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.

## Git grep

buscar palabras en los archivos en el branch actual

`git grep "palabra a buscar"`

mostrar la linea en la cual la palabra aparece en el archivo

`git grep -n "palabra a buscar"`

mostrar cuantas veces aparece la palabra en cada archivo

`git grep -c "palabra a buscar"`

buscar los commits en los cuales sale una palabra

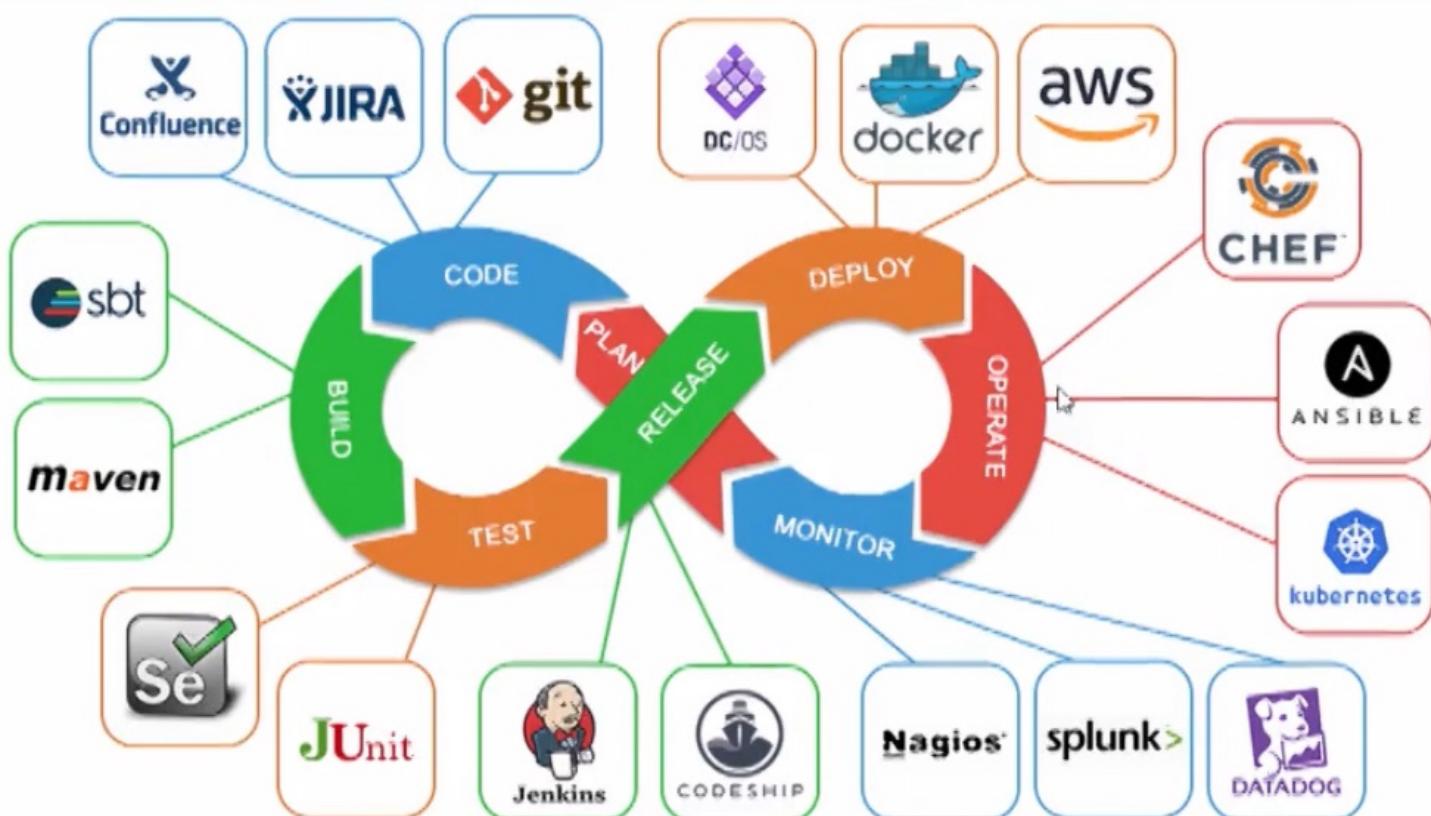
`git log -S "palabra a buscar"`

## Comandos bono



- `git shortlog`: Ver cuantos commits a hecho los miembros del equipo
- `git shortlog -sn`: Las personas que han hecho ciertos commits
- `git shortlog -sn --all`: Todos los commits (también los borrados)
- `git shortlog -sn --all --no-merges`: muestra las estadisticas de los commits del repositorio donde estoy
- `git config --global alias.stats "shortlog -sn --all --no-merges"`: configura el comando "shortlog -sn --all --no-merges" en un Alias en las configuraciones globales de git del pc
- `git blame -c blogpost.html`: Muestra quien ha hecho cambios en dicho archivo identado
- `git blame --help`: Muestra en el navegador el uso del comando
- `git blame archivo -L 35, 60 -c`: Muestra quien escribio el codigo con informacion de la linea 35 a la 60, EJ: `git blame css/estilos.css -L 35, 60 -c`
- `git branch -r`: Muestra las Ramas remotas de GitHub
- `git branch -a`: Muestra todas las Ramas del repo y remotas de GitHub

Utilidades GitHub:  
Panel insights  
pulse merge/pull request/ issues/historia de contribuciones  
contributors contribuidores del proyecto  
community readme, codigo de conducta, licencias, etc  
traffic: trafico del proyecto  
commits historia de commits  
code frequency codigo añadido/borrado  
dependency graph uso de otras librerias  
alerts alertas de GitHub  
network historia de contribucion de los colaboradores  
forks muestra los forks del proyectos



Aprendimos cómo usar Git y GitHub, hacer merge request, investigar quién hizo qué a través de la línea de comandos, cómo utilizar GitHub Pages, cómo revertir cambios y mucho más. Ahora queda de tu parte experimentar, fallar, subir, borrar y por último hacer deploy de tu proyecto y compartirlo con la comunidad.