

# USO DE GITHUB



GitHub Inc.	
<h1>GitHub</h1>	
Tipo	Filial
Industria	Software
Fundación	8 de febrero de 2008 (12 años)
Fundador	Tom Preston-Werner Chris Wanstrath P. J. Hyett Scott Chacon
Sede	San Francisco, California, Estados Unidos
Personas clave	Nat Friedman (CEO)
Propietario	Microsoft
Matriz	Microsoft Corporation
Filiales	Npm, Inc.
Sitio web	github.com

## Comandos útiles

git fetch #actualizar mi copia local del repositorio remoto, no lo copia en el directorio local

git merge #para combinar los últimos cambios del repositorio remoto y nuestro directorio de trabajo

git pull origin master

Comando para traer los cambios realizados en el repositorio de Github a nuestro repositorio local

README.md

Archivo que veremos por defecto al entrar a un repositorio. Sirve para describir el proyecto, los requerimientos y las instrucciones que debemos seguir para contribuir correctamente

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.

Luego de crear nuestra cuenta en <https://github.com> 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.

### • git clone "URL"

Comando 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.

Clone with HTTPS ⓘ  
Use Git or checkout with SVN using the web URL.  
<https://github.com/freddier/hyperblog> ⌂  
Download ZIP

### • Conectar el repositorio de GitHub con nuestro repositorio local:

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

`git remote add origin "URL"`

2) Verificar que la URL se haya guardado correctamente:

`git remote`  
`git remote -v`

\* Si hacemos `git push origin master`, nos mostrará una advertencia debido a la diferencia de archivos de trabajo en ambos repositorios.

```
Proyecto: git@github.com:ayenque/hyperblog.git
> git push origin master
Username for 'https://github.com': ayenque
Password for 'https://ayenque@github.com':
To https://github.com/ayenque/hyperblog.git
! [rejected]          master -> master (fetch first)
error: failed to push some references to 'https://github.com/ayenque/hyperblog.git'
ayuda: Actualizaciones fueron rechazadas porque el remoto contiene trabajo que
ayuda: no existe localmente. Esto es causado usualmente por otro repositorio
ayuda: realizando push a la misma ref. Quizás quiera integrar primero los cambios
ayuda: remotos (ej. 'git pull ...') antes de volver a hacer push.
ayuda: Vea 'Notes about fast-forwards' en 'git push -h' para detalles.
```

3) Debemos traer la versión del repositorio remoto y hacer merge para crear un commit con los archivos de ambas partes. Podemos usar `git fetch` y `git merge` o solo el `git pull`, pero para forzar se debe usar:

`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

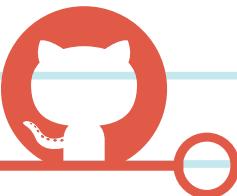
`git push origin master`

GitHub conocido como la "red social de los programadores", es un super-servidor de Git que nos permite alojar nuestros proyectos de tal manera que cualquiera pueda acceder y colaborar de forma remota en el desarrollo de los mismos. De esta forma, podemos tener nuestro portafolio de proyectos y dar a conocer que sabemos y que podemos hacer.



git

# CÓMO FUNCIONAN LAS LLAVES PÚBLICAS Y PRIVADAS



Tambien conocido como "Cifrado asimétrico de un solo camino"

La llaves se crean con un proceso algorítmico y esan vinculadas matematicamente una con la otra, de esta manera estan asociadas.

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

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

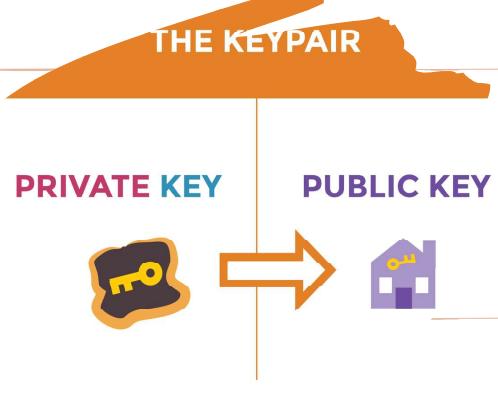
<https://www.s>

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

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

## Llaves públicas y privadas

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



## ¿CÓMO FUNCIONA EL CIFRADO ASIMÉTRICO?

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



Las llaves públicas son visibles para todo el mundo



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



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

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



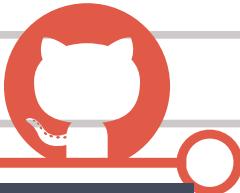
Las llaves públicas y privadas sirven para compartir información en Internet de una forma segura, incluso si el mensaje es interceptado la probabilidad de que se pueda descifrar es casi nulo, si no se tiene la llave privada. Este método se usa incluso en el mundo financiero.

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



git

# CONFIGURA TUS LLAVES SSH EN LOCAL



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 mensaje.

Si conectamos Github por HTTPS nuestro usuario y contraseña se guardando en el entorno local, y eres vulnerable a password cracking.

Si conectamos Github por SSH, además de que la conexión es mas segura, no tenemos que volver a ingresar nuestras credenciales de Github.

Para la conexión con Github, al enviarle nuestra llave publica Github nos envía cifrado su propia llave publica.

Se recomienda cifrar nuestro disco en Windows o Linux



Las llaves SSH no son por repositorio o proyecto si no por personal!

Procedimiento para crear y configurar las llaves en nuestro servidor SSH de nuestra computadora:

1) Revisar las configuraciones globales

```
> git config -l  
~ 20s  
> git config --global user.email "ayenquet@gmail.com"  
~ 6s  
> cd ~
```

2) Ahora se crea la llave SSH, estando en el Home

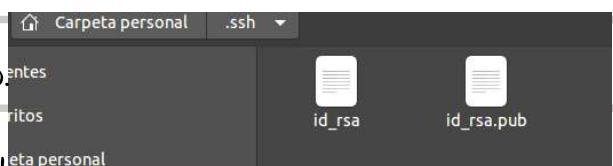
```
ssh-keygen -t rsa -b 4096 -C "ayenquet@gmail.com"
```

```
> ssh-keygen -t rsa -b 4096 -C "ayenquet@gmail.com"  
Generating public/private rsa key pair.  
Enter file in which to save the key (/home/ayenquet/.ssh/id_rsa):  
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:  
Your identification has been saved in /home/ayenquet/.ssh/id_rsa  
Your public key has been saved in /home/ayenquet/.ssh/id_rsa.pub  
The key fingerprint is:  
SHA256:MrPCd3y7xBe1CuX0LcWt3n5 [REDACTED] <ayenquet@gmail.com>  
The key's randomart image is:  
+---[RSA 4096]---
```

#-t que tipo de algoritmo se va a usar para crear esa llave, rsa el más popular  
#-b que tan compleja es la llave desde una perspectiva matematica  
#-C que correo electronico va a estar conectada la llave, email de Github

Tomar nota que es recomendable dar una clave adicional para darle mayor seguridad.

3) Revisamos que nuestras llaves se han generado con éxito.



4) Revisar que el servidor de SSH este corriendo:

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

```
> eval $(ssh-agent -s)  
Agent pid 16358
```

5) Ahora debemos agregar la llave que acabamos de crear

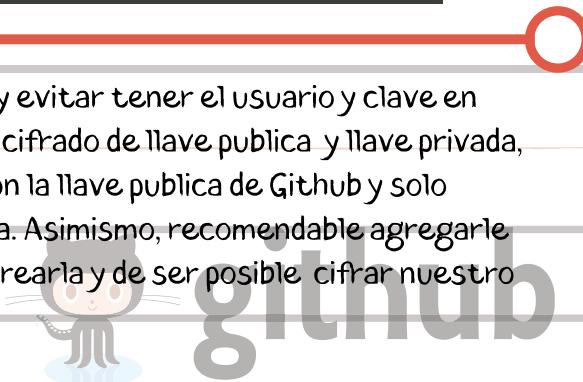
```
ssh-add ~/.ssh/id_rsa
```

```
> ssh-add ~/.ssh/id_rsa  
Enter passphrase for /home/ayenquet/.ssh/id_rsa:  
Identity added: /home/ayenquet/.ssh/id_rsa (ayenquet@gmail.com)
```

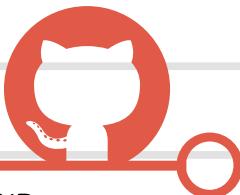
## PARA MAC:

```
# 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
```

Para tener una mayor seguridad en nuestros proyectos y evitar tener el usuario y clave en nuestro repositorio local, debemos utilizar el método de cifrado de llave pública y llave privada, de esta manera la conexión se realizará directamente con la llave pública de Github y solo debemos guardar y nunca compartir nuestra llave privada. Asimismo, recomendable agregarle una contraseña adicional o "passphrase" al momento de crearla y de ser posible cifrar nuestro disco duro.



# 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 mensaje.

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.

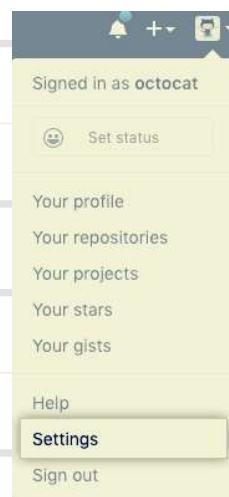
Mensaje que asegura que la IP de GitHub es reconocida como de confianza, no es mensaje de error, solo de advertencia.

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).



6) Haz clic en Add SSH key (Agregar tecla SSH).

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

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).



## • 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.

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 (140.82.113.3)' can't be established.  
RSA key fingerprint is SHA256nThbg6XwJWc17E1Ic  
Are you sure you want to continue connecting (yes/no/fingerprint)? y  
Warning: Permanently added 'github.com,140.82.113.3' (RSA) to the list of known hosts.  
You have no information on this host so it's actually.  
Por favor especifica a qué rama quieres 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* 7ba  
> 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.
```

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

Proyecto git/master

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

4) Verificar que la URL remota ha cambiado.

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 c38168c] Una versión del Hyperblog  
1 file changed, 1 insertion(+), 1 deletion(-)  
  
Proyecto git/master  
> git pull origin master  
Warning: Permanently added the RSA host key for IP address '140.82.114.4' to the list of known hosts.  
Desde github.com:ayenque/hyperblog  
* branch      master    -> FETCH_HEAD  
Ya está actualizado.  
  
Proyecto git/master  
> git push origin master  
Enumerando objetos: 100% (5/5), listo.  
Comprimiendo objetos: 100% (3/3), listo.  
Comprimiendo objetos: 100% (3/3), listo.  
Escribiendo objetos: 100% (3/3), listo.  
Total 3 (delta 0), reusado 0 (delta 0).  
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.  
To github.com:ayenque/hyperblog.git  
7771ea9..c3b168c master -> master  
  
Proyecto git/master
```

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



# TAGS Y VERSIONES EN GIT Y GITHUB



## Comandos útiles

git tag -a nombre-tag -m "mensaje" # si se omite el HASH, el tag se referencia al commit actual

git tag -l # comando para ver la lista de tags

git show nombre-tag # comando para ver la información de la etiqueta junto con el commit que está etiquetado

git push origin nombre-tag # comando para enviar un tag determinado a Github

git config --global alias."nombre del alias" "git log --all --graph --decorate --oneline" # comando para guardar un alias en las configuraciones de git, para ejecutar:  
git "nombre del alias", #En este caso mostrará el log de forma 'grafica'.

Recordar siempre primero hace el git pull origin master, para traernos los cambios del remoto como un buena práctica.

Los tags se pueden usar como releases, por eso tienden a quedar en Github, aun después de eliminarlos y ejecutar un git push.

Como muchos VCS (Version Control Systems), Git tiene la posibilidad de etiquetar puntos específicos del historial como importantes. Esta funcionalidad se usa típicamente para marcar versiones de lanzamiento (v1.0, por ejemplo).

- **git tag -a nombre-del-tag -m "mensaje"** HASH-a-etiquetar

Comando para crear un tag y asignarlo a un determinado commit (hash)

```
Proyecto1 git/master 18s
> git tag -a v0.2 -m "Resultado de las primeras clases del curso" 33bac85
```

- **git tag**

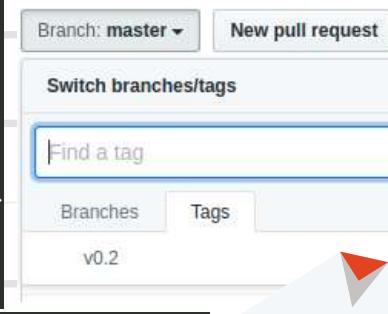
Comando para revisar la lista de tags creados.

Los tags no son cambios que afectan al staging o al repositorio local, sin embargo si se tienen que enviar al repositorio remoto (Github)

- **git push origin --tags**

Comando para enviar los tags creados al repositorio remoto, y si revisamos en Github, comprobamos que el tag ha sido creado.

```
Proyecto1 git/master
> git push origin --tags
Enumerando objetos: 7, listo.
Contando objetos: 100% (7/7), listo.
Comprimiendo objetos: 100% (5/5), listo.
Escribiendo objetos: 100% (5/5), 645 bytes | 645.00 KiB/s, listo.
Total 5 (delta 2), reusado 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:ayenque/hyperblog.git
 * [new tag]          0.1 -> 0.1
 * [new tag]          0.5 -> 0.5
 * [new tag]          v0.2 -> v0.2
```



- **git tag -d nombre-del-tag**

Comando para eliminar un tag

```
Proyecto1 git/master
> git tag -d 0.1
Etiqueta '0.1' eliminada (era 64b7389)

Proyecto1 git/master
> git tag -d 0.5
Etiqueta '0.5' eliminada (era b1d57aa)
```

Luego de eliminar y enviar los cambios a GitHub (git pull y luego git push origin --tags), nos damos cuenta que todavía aparecen los tags en Github, por ello hacemos lo siguiente:

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

Comando para eliminar un tag y su referencia en el repositorio remoto, de esta forma borrar de la lista de tags de Github.

```
Proyecto1 git/master
> git push origin :refs/tags/0.5
To github.com:ayenque/hyperblog.git
 - [deleted]           0.5
```

Los tags sirven cuando necesitamos marcar un punto específico en la historia de nuestro trabajo (para los releases). De esta forma, podemos hacer un seguimiento al progreso de nuestro proyecto e identificar los cambios más fácilmente entre cada versión, incluso podemos hacer un checkout a uno de esos tags.



github

# MANEJO DE RAMAS EN GITHUB



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

En caso de error al momento de ejecutar gitk:

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

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

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



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

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

## • git branch

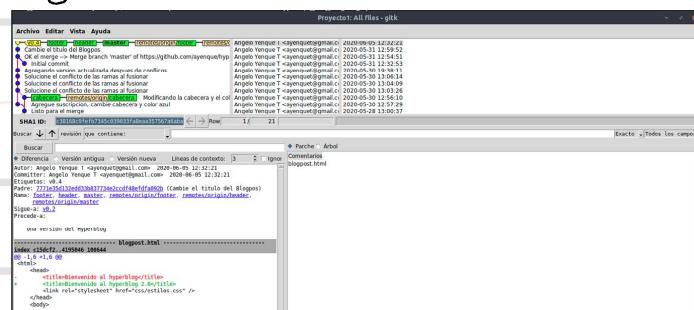
Comando para revisar las ramas creadas, pero si queremos ver más detalles podemos usar los siguientes comandos.

## • git show-branch

## • git show-branch --all

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

## • gitk



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

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

Nos ubicamos en la rama master con git checkout master

Creamos la dos ramas: git branch header y git branch footer

## • git push origin "nombre rama"

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

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

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

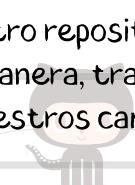
```
Proyecto git/master  
> git push origin header  
Total 0 (delta 0), reusado 0 (delta 0)  
remote:  
remote: Create a pull request for 'header' on GitHub by visiting:  
remote: https://github.com/ayenque/hyperblog/pull/new/header  
remote:  
To github.com:ayenque/hyperblog.git  
 * [new branch] header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

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

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

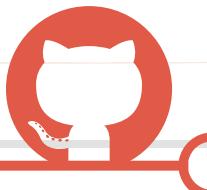


Las ramas en Github son importantes porque representan un área de trabajo independiente de desarrollo dentro de nuestro proyecto. Al igual que en nuestro repositorio local, en Github podemos trabajar con ramas y nos permiten de la misma manera, traernos los cambios realizados en otras ramas y compararlos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



github

# CONFIGURAR MÚLTIPLES COLABORADORES EN UN REPOSITORIO DE GITHUB



- `git clone url-del-remoto`  
Comando que permite clonar el repositorio remoto de forma local. La URL puede ser HTTP o SSH, si tenemos agregado nuestra llave pública a Github.

Repositorio > Settings  
> Collaborators

La ruta que anteriormente se usaba para agregar a los colaboradores, ahora se ingresa por "Manage acces"

En la página principal del repositorio también puedo ver la lista de los colaboradores.

Contributors 2

	ayenque	ayenque
	ayenquetest	ayenquetest

Aporte  
[tucorreo+algo@gmail.com](mailto:tucorreo+algo@gmail.com)  
[tucorreo+algo@hotmail.com](mailto:tucorreo+algo@hotmail.com)

Podemos agregar el símbolo "+" después de nuestro usuario de correo, para crear un alias y así asociarlo a la misma bandeja. Con esto podemos tener "múltiples cuentas" para crear adicionales en Github y poder hacer nuestras pruebas y prácticas.

Por defecto, cualquier persona puede clonar o descargar tu proyecto desde GitHub, pero no pueden crear commits, ni ramas, ni nada. Para solucionar esto podemos añadir a cada persona de nuestro equipo como colaborador de nuestro repositorio.

1. En **GitHub**, visita la página principal **del repositorio**

2. Debajo de tu nombre de repositorio, da clic en **Settings**.

3. En la barra lateral izquierda, da clic en **Manage access**

4. Da clic en **Invite a collaborator**



You haven't invited any collaborators yet

5. Comienza a teclear el nombre de la persona que deseas invitar dentro del campo de búsqueda. Posteriormente, da clic en algún nombre de la lista de coincidencias. Despues click en "**Add [user] to [repository]**"

El usuario recibirá un correo electrónico invitándolo al repositorio.

6. Click en **View invitation** y luego de ingresar a su cuenta **Accept invitation**.

Una vez que el colaborador **acepte la invitación**, tendrá acceso de colaborador a tu repositorio.

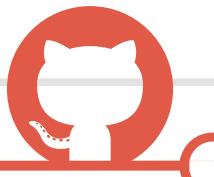
Se verifica que tenga la opción **Edit** en algún archivo para confirmar que el proceso fue exitoso.

Para que otros miembros del equipo puedan colaborar con nuestro repositorio en Github, es necesario agregarlos a la lista de colaboradores, dirigiéndonos a **Repositorio > Settings > Manage acces > Collaborators** y añadir su email o username. De esta manera nos preparamos para un flujo de trabajo profesional.



github

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



Ctrl + Shift + R para forzar la actualización y asegurar de ver los cambios de los archivos binarios.

Siempre es una buena práctica ejecutar el comando «git pull origin master» antes de hacer un git push, esto por si alguien hizo algún cambio en el servidor y evitar errores.

Las mejores prácticas dicen que los archivos binarios no deben estar agregados al repositorio, debe estar ignorados.

Podemos tener situaciones en las que el trabajo tiene que ser dividido en ramas para trabajar de forma eficiente, para esto debemos proceder según los procedimientos Básicos de Ramificación:

## HEADER



Agregamos una imagen al repositorio a modo de ejemplo.

```
git add imágenes/dragon.png
```

```
git commit -m "Logo del Header"
```

La imagen fue agregada.

Ahora traemo y enviamos los cambios al repositorio.

```
git pull origin header  
git push origin header
```

Mejoramos el logo:



```
git commit -am "Logo Mejorado"
```

```
git pull origin header  
git push origin header
```

La imagen fue actualizada en Github.

Ahora realizamos los cambios en nuestro Header.

```
git commit -am "Color de fondo,  
logo nuevo y mejor color de header"
```

Traemos y enviamos los cambios en Github:

```
git pull origin header  
git push origin header
```

Una de las herramientas más útiles que tiene Git son las ramas. Las ramas pueden tomar diferentes caminos como partes del desarrollo del trabajo, pero que en algún momento, debemos unirlos. Para esto, debemos elegir una rama principal y tener claro cuál es el procedimiento para tener éxito en la fusión de todos los avances y/o updates del proyecto.

## FOOTER

Traemos los cambios del footer del repositorio y nos ubicamos en la rama Footer.

```
git pull origin footer  
git checkout footer
```

Realizamos los cambios requeridos en el footer y confirmamos.

```
git commit -am "Footer terminado"
```

```
git pull origin footer  
git push origin footer
```

Con esto enviamos los cambios realizados al repositorio.

## MASTER

Como administrador puedo revisar los cambios realizados en otras ramas:

Nos ubicamos en la rama :

```
git checkout header
```

Traemos los cambios realizados por miembros del equipo del repositorio:

```
git pull origin header
```

Haciendo un code review, puedo fusionar los cambios.

Voy a la rama maestra (Principal):

```
git checkout master
```

Procedemos a fusionar los cambios:

```
git merge header
```

Los cambios de la rama Header ya figuran en la rama Master:

Enviamos la rama actualizada al repositorio remoto:

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

Para traer los cambios del footer

Estando en master, los comandos serían :

```
git checkout footer
```

Traemos los cambios:

```
git pull origin footer
```

Hacemos el code review y procedemos a fusionar los cambios.

Voy a la rama maestra (Principal):

```
git checkout master
```

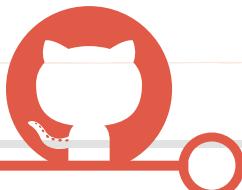
Procedemos a fusionar los cambios:

```
git merge footer
```

Finalmente envío la rama master actualizada a Github

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

# FLUJO DE TRABAJO PROFESIONAL CON PULL REQUESTS



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

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

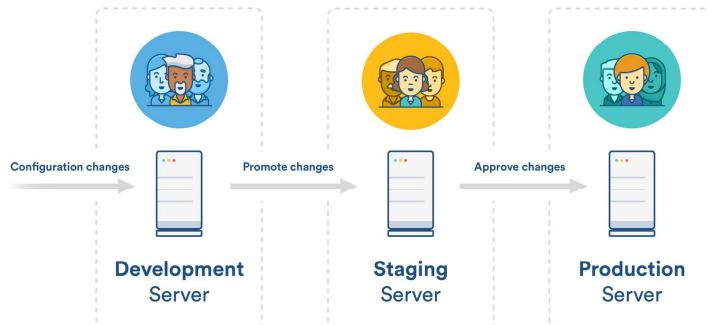
Los **pull request** tambien son importantes porque permiten a personas que no son colaboradores, trabajar y apoyar en nuestro proyecto.

Equivalencia en otras plataformas:

Bitbucket	GitHub	GitLab
Pull Request	Pull Request	Merge Request

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

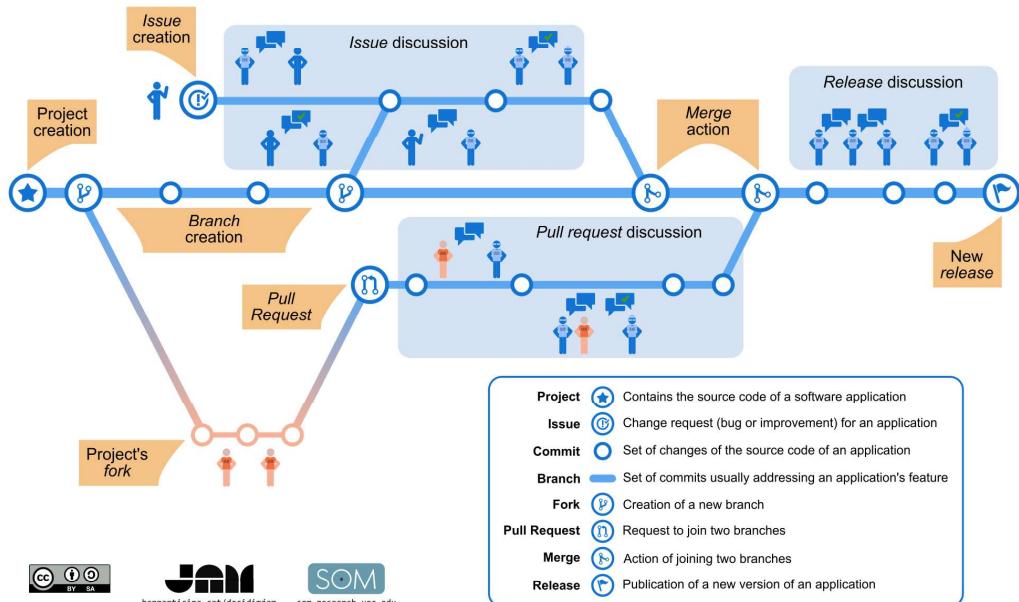
Para realizar pruebas enviamos el código a servidores que normalmente los llamamos **staging server**, luego de que se realizan las pruebas pertinentes tanto de código como de la aplicación estos pasan a el **servidor de producción**.



## PULL REQUESTS

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

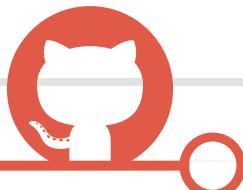
How GitHub projects are developed?  
Where are the main discussion points?



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

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

# 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 último 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

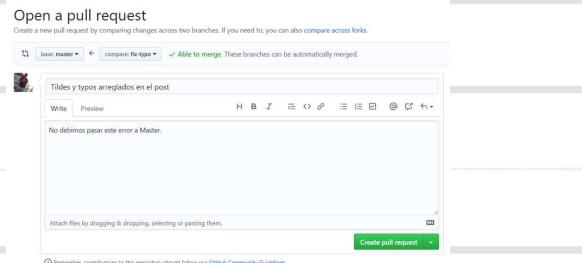
Una vez realizado el Merge, es importante traer los cambios en Git (Git Pull), en caso borramos la rama en Github, podemos borrarla en Git con `git branch -D nombre-rama`

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-typo 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:

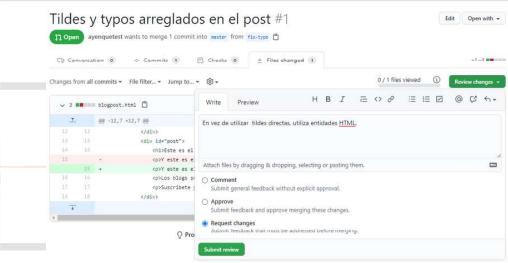
Recent activity

Tildes y tipos arreglados en el post  
ayenque/hyperblog · Your review was requested 3 days ago

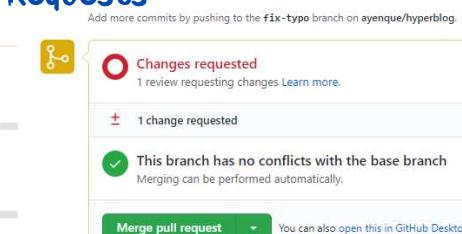
d) El administrador puede comparar los cambios haciendo click en **Files changed**, y consultar con el equipo para la revisión respectiva.



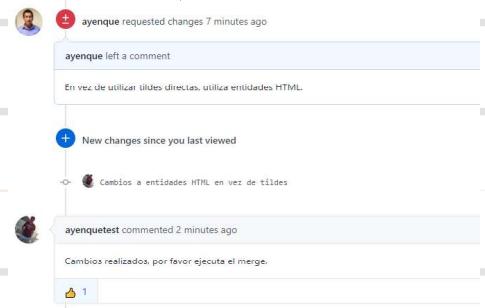
e) Dar click en **Review Changes**, en caso de requerir cambios al solicitante.



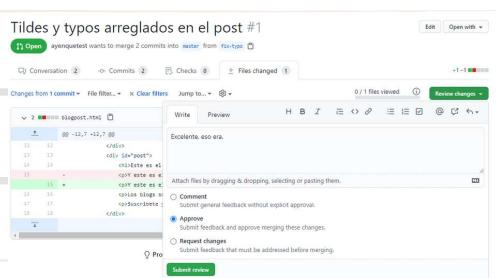
f) Al solicitante, se le notifica del cambio requerido ingresando en **Pull Requests**



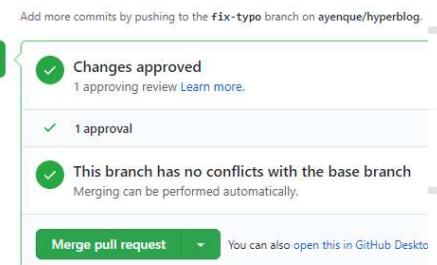
g) Una vez realizados los cambios requeridos responder el **Pull Request**



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



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**)



Github y sus similares, nos ofrecen una serie de herramientas que permiten al equipo realizar un seguimiento para la revisión de cada merge que se realiza a la rama principal.

# CREANDO UN FORK, CONTRIBUYENDO A UN REPOSITORIO



Watch 35 Star 45 Fork 37

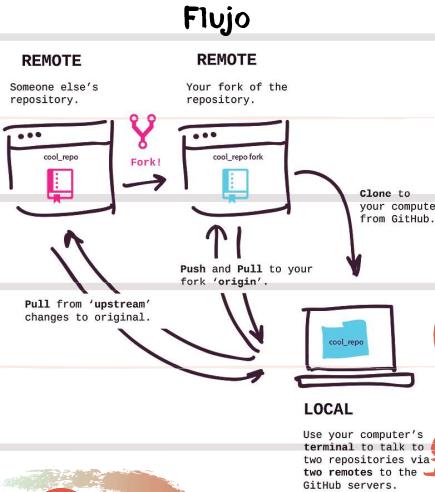
En un proyecto público podemos dar click en **Watching** y **Star** para seguir el repositorio, de esta manera apoyamos y recibimos avisos cada vez que hay cambios en dicho repositorio.

Bifurcar (fork) un proyecto público es una característica única de Github (no de Git).

Github nos muestra el repositorio original en el repositorio forkeado (**forked from**)

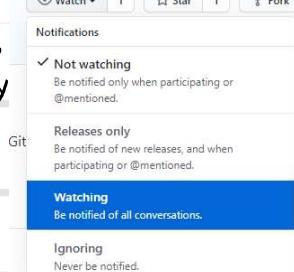
ayenquetest / hyperblog  
forked from ayenque/hyperblog

Una vez que hacemos fork en un repositorio podemos clonarlo en nuestro local con **git clone url-propia-del-repositorio-fork** para empezar a aplicar nuestros cambios.



Siquieres participar en un proyecto existente, en el que no tengas permisos de escritura, puedes bifurcarlo (hacer un "fork"). Esto consiste en crear una copia completa del repositorio totalmente bajo tu control: se almacenará en tu cuenta y podrás escribir en él sin limitaciones.

Fork



Forking ayenque/hyperblog

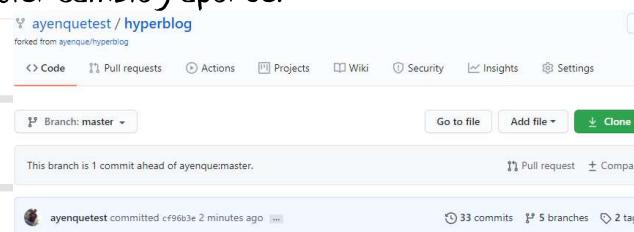
It should only take a few seconds.



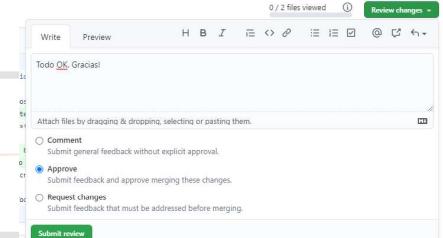
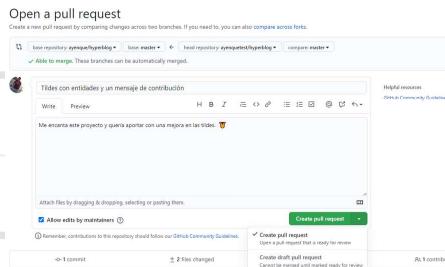
Para bifurcar un proyecto, visita la página del mismo y pulsa sobre el botón "Fork" del lado superior derecho de la página.

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:

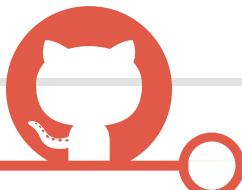
**git pull upstream master # traemos cambios del repositorio original**

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

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

Cuando hacemos un fork de un repositorio, se hace una copia exacta del repositorio original que podemos utilizar como un repositorio git cualquiera. Despues de hacer fork tendremos dos repositorios git idénticos pero con distinta URL. Finalizado el proceso, tendremos dos repositorios independientes que pueden cada uno evolucionar de forma totalmente autónoma, asimismo, Github nos permite comparar los cambios con el proyecto original para poder aportar mediante un pull request.

# IGNORAR ARCHIVOS EN EL REPOSITORIO CON .GITIGNORE



Si tienes archivos que no pueden ser públicos, como archivos de configuración con contraseñas, lo ideal es que no los subas a tu repositorio, estos archivos los puedes poner en el archivo `.gitignore`

Una buena práctica es evitar que los **archivos binarios** del contenido, sean parte del repositorio.

Es importante que el archivo se nombre `".gitignore"`, con punto al inicio.

Cuando se crea el archivo `.gitignore`, se debe confirmar con `git add .gitignore` y luego el `commit` respectivo para que se agregue al repositorio

Los archivos binarios que no se suben al repositorio se pueden referenciar por FTP o con un CDN, o por otro servicio.

Inspirarte en proyectos Open Source, es la forma en la que nos podemos ayudar en la industria del software y aprender de forma correcta.

A veces, tendrás algún tipo de archivo que no quieres que Git añada automáticamente o más aun, que ni siquiera quieras que aparezca como no rastreado. Este suele ser el caso de archivos generados automáticamente como trazas o archivos creados por tu sistema de compilación.

En estos casos, puedes crear un archivo llamado `.gitignore` que liste patrones a considerar.

Las reglas sobre los patrones que puedes incluir en el archivo `.gitignore` son las siguientes:

- Ignorar las líneas en blanco y aquellas que comiencen con #.
- Aceptar patrones glob estándar.
- Los patrones pueden terminar en barra (/) para especificar un directorio.
- Los patrones pueden negarse si se añade al principio el signo de exclamación (!).

Ejemplo de un archivo `.gitignore`:

<https://gitignore.io/>

• [.gitignore.io](#)

Create useful .gitignore files for your project

x Node x OSX x Polymer x SublimeText x Windows Generate

```
# ignora los archivos terminados en .a
*.a

# pero no lib.a, aun cuando había ignorado los archivos terminados en .a en la línea anterior
!lib.a

# ignora únicamente el archivo TODO de la raíz, no subdir/TODO
/TODO

# ignora todos los archivos del directorio build/
build/

# ignora doc/notes.txt, pero no este: doc/server/arch.txt
doc/*.*txt

# ignora todos los archivos .txt del directorio doc/
doc/**/*.*txt
```

**gitignore**



**vuejs/vue**  
Vue.js is a progressive, incrementally-adoptable JavaScript...  
[github.com](https://github.com/vuejs/vue/blob/dev/.gitignore)

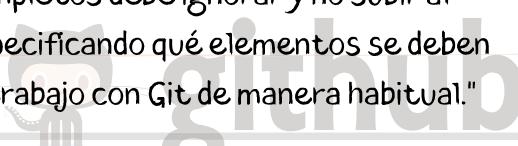


**laravel/laravel**  
A PHP framework for web artisans.  
Contribute to laravel/laravel...  
[github.com](https://github.com/laravel/laravel/blob/master/.gitignore)

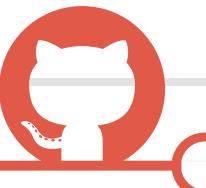


**TryGhost/Ghost**  
The #1 headless Node.js CMS for professional publishing - TryGhost/Ghost  
[github.com](https://github.com/TryGhost/Ghost/blob/master/.gitignore)

".Gitignore sirve para decirle a Git qué archivos o directorios completos debe ignorar y no subir al repositorio de código. Únicamente se necesita crear un archivo especificando qué elementos se deben ignorar y, a partir de entonces, realizar el resto del proceso para trabajar con Git de manera habitual."



# 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.

M 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!



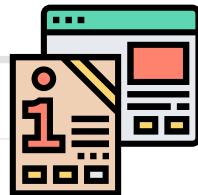
README brinda una guía rápida a los que acaban de descubrir nuestro proyecto de cómo empezar a usarlo. Es muy importante que vaya al grano, sea conciso y muy claro. Para extendernos y entrar en detalles está la documentación, a la que siempre podemos enlazar desde dentro de nuestro README.

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>



Vue.js is an MIT-licensed open source project with its ongoing development made possible entirely by the support of these awesome [backers](#). If you'd like to join them, please consider:

- Become a backer or sponsor on [Patreon](#).
- Become a backer or sponsor on [Open Collective](#).
- One-time donation via [PayPal](#) or crypto-currencies.

What's the difference between Patreon and OpenCollective?  
Funds donated via Patreon go directly to support Evan You's full-time work on Vue.js. Funds donated via OpenCollective are managed with transparent expenses and will be used for compensating work and expenses for core team members or sponsoring community events. Your name/logo will receive proper recognition and exposure by donating on either platform.



**laravel/laravel**

A PHP framework for web artisans.  
Contribute to [laravel/laravel...](#)

[github.com](#)

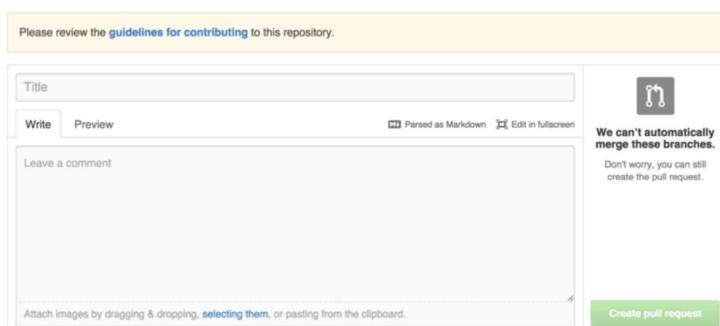
<https://github.com/laravel/laravel/blob/master/README.md>



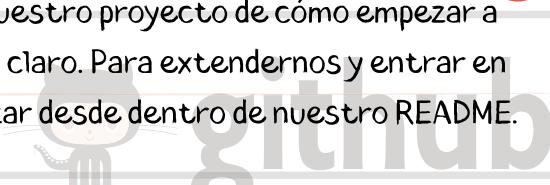
<https://github.com/TryGhost/Ghost/blob/master/README.md>

## CONTRIBUTING

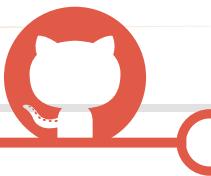
El otro archivo que GitHub reconoce es CONTRIBUTING. Si tienes un archivo con ese nombre y cualquier extensión, GitHub mostrará algo como la imagen, cuando se intente abrir un Pull Request.



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.



# TU SITIO WEB PÚBLICO CON GITHUB PAGES



## Dominio personalizado para Github Pages

Puedes personalizar el nombre de dominio de tu sitio de Github Pages. En **Settings - Github Pages - Custom domain**, ingresar la url personalizada y guardar. Luego se debe configurar los DNS en la configuración del dominio.

<https://docs.github.com/en/github/working-with-github-pages/managing-a-custom-domain-for-your-github-pages-site>

## Temas para Github Pages

Puedes personalizar el tema de tu sitio de Github Pages con **Jekyll**. En **Settings - Github Pages - Theme Chooser**, Seleccionar **Choose a theme**, elegir el Tema a aplicar y seleccionar **Select Theme**.

ayenque Set theme jekyll-theme-cayman  
README.md Creando Readme.md  
.config.yml Set theme jekyll-theme-cayman  
index.html Agregando index  
<https://docs.github.com/en/github/working-with-github-pages/about-github-pages-and-jekyll>

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 \* sophashelp.github.io  
Great repository names are short and memorable. Need inspiration? How about fi  
Description (optional)  
Public Anyone can see this repository. You choose who can commit.  
Private You choose who can see and commit to this repository.

## 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
```

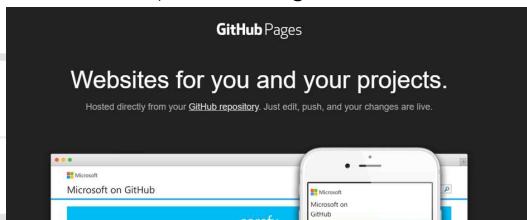
Ejecutar los comando git add, git commit, y git push para aplicar los cambios:

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

## GITHUB PAGES

Puedes usar Páginas de GitHub para albergar un sitio web sobre ti mismo, tu organización o tu proyecto directamente desde un repositorio GitHub.

Abrir la página oficial de Github para más ayuda: <https://pages.github.com/>



Para agregar una url de Github Pages a un repositorio existente, ir a "Settings" del repositorio en la pestaña "Options", en la sección "GitHub Pages".

ayenque / hyperblog  
Code Issues Pull requests Actions Projects Wiki Security Insights Settings  
Options Manage access Security & analysis Branches  
Repository name hyperblog Rename  
hyperblog.github.io/hyperblog/

Elegir en "Source", la rama principal donde se encuentra el código actualizado. Esto genera en automático la Url de la web alojada en Github.

GitHub Pages  
GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.  
✓ Your site is published at <https://ayenque.github.io/hyperblog/>  
Source  
Your GitHub Pages site is currently being built from the master branch. Learn more.  
Branch: master / (root) Save  
<https://ayenque.github.io/hyperblog/>

hyperblog

Hyperblog 💚

Un blog increíble para el curso de Git y Github de Platzi

El curso de Git y Github de Platzi es lo que me hacía falta

## En este curso vemos de todo

- Todos los comandos de Git
- El flujo de trabajo en Github
- El verdadero amor por las buenas prácticas
- Trucos muy locos del profesor
- Las personalidades múltiples de Freddy
- Creado por el increíble Platzi Team
- Incluye ejemplos en Windows, Linux y Mac
- Disponible para todas las edades

## GITHUB PAGES PERSONAL

En caso se desea obtener una URL de forma `<> usuario.github.io <>`, se debe crear un repositorio con el mismo nombre:

ayenque / ayenque.github.io  
Code Issues Pull requests Actions Projects Wiki Security Insights Settings  
Options Manage access Security & analysis  
Repository name ayenque.github.io Rename  
ayenque.github.io

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

Este es un test para el curso de Git y Github



**Github Pages** es un beneficio adicional que te permite publicar tu sitio web de forma gratuita y desde la CDN global de Github, basado en una rama que puede ser diferente a la master. La ventaja de esto es que la web se puede actualizar con cambios en el repositorio y ejecutando un git commit/git push, además de contar con certificado HTTPS para proporcionar mayor transparencia y seguridad para los usuarios.



# GIT REBASE: REORGANIZANDO EL TRABAJO

## REALIZADO



### REBASE ES UNA MALA PRACTICA



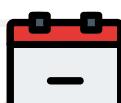
Rebase es solo para repositorio locales, porque reescribe la historia del repositorio.

Rebase es una forma de hacer cambios silenciosos en otras ramas y volver a insertar la historia de esa rama a una anterior.

Primero se aplica rebase a la rama que tiene los cambios y luego rebase a la rama final, donde queremos que queden los cambios.

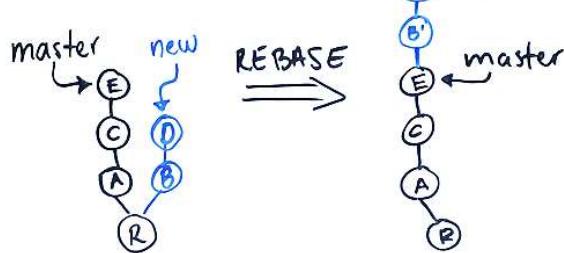
#### Problemas:

- No queda historia, de los cambios originales
- No se sabe el autor real de los commits.
- Si la rama principal avanzó varios commits, puede generar varios conflictos que se tienen que arreglar de forma manual.



```
* f43b9fa - (hace 8 minutos) Experimento 2 - Angelo Yenque T (HEAD -> master)
* 86ef416 - (hace 8 minutos) Experimento 1 - Angelo Yenque T
* c74b930 - (hace 6 minutos) Master 2 - Angelo Yenque T
* e33e175 - (hace 9 minutos) Master 1 - Angelo Yenque T
* 8d40083 - (hace 4 días) Readme modificado y mejorado - Angelo Yenque T (origin)
* 5495686 - (hace 7 días) Referenciando una imagen desde un servidor externo
```

La manera más sencilla de integrar ramas, tal y como hemos visto, es el comando git merge. Sin embargo, también hay otra forma de hacerlo: con el comando **git rebase**, puedes capturar todos los cambios confirmados en una rama y reaplicarlos sobre otra.



#### Ejemplo:

Vamos a cambiar el archivo `historia.txt` de nuestro proyecto.

- Realizamos un cambio del archivo en la rama `master`
- Creamos una rama `experimento`, nos movemos a la rama y realizamos un cambio en el mismo archivo.

```
Proyecto1 git/experimento
> git commit -am "Experimento 1"
[experimento a82ae9d3] Experimento 1
 1 file changed, 2 insertions(+)
```

```
Proyecto1 git/master*
> git commit -am "Master 1"
[master e33e175] Master 1
 1 file changed, 2 insertions(+)
```

```
Proyecto1 git/experimento
> git commit -am "Experimento 1"
[experimento a82ae9d3] Experimento 1
 1 file changed, 2 insertions(+)
```

- Realizamos otro cambio en el archivo `Historia.txt` y lo aplicamos como "Experimento 2"



Estando en la rama `Experimento`, si ejecutamos el comando **git rebase master** nos dirá que la rama está actualizada ya que no se han realizado cambios en `master`.

```
Proyecto1 git/experimento
> git rebase master
La rama actual experimento está actualizada.
```

- Entonces nos movemos en la rama `Master` y realizamos un cambio "Master 2".

```
Proyecto1 git/experimento 225
> git rebase master
En primer lugar, rebobinando HEAD para después reproducir tus cambios encima de ésta...
Aplicando: Experimento 1
Usando la información del índice para reconstruir un árbol base...
M historia.txt
Retocando para parchar base y fusión de 3-vías...
Auto-fusionando historia.txt
Aplicando: Experimento 2
Usando la información del índice para reconstruir un árbol base...
M historia.txt
Retocando para parchar base y fusión de 3-vías...
Auto-fusionando historia.txt
```

```
Proyecto1 git/master
> git commit -am "Master 2"
[master c74b930] Master 2
 1 file changed, 2 insertions(+)
```

- Regresamos a la rama `Experimento` y ejecutamos **git rebase master**
- Regresamos a la rama `Master` y ejecutamos **git rebase experimento**

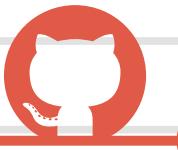
```
Proyecto1 git/master
> git rebase experimento
En primer lugar, rebobinando HEAD para después reproducir tus cambios encima de ésta...
Avance rápido de master a experimento.
```

Ahora si revisamos el log de la rama `master` parecerá que todos los cambios se hicieron en la rama `master` y no queda registro de la rama `experimento`.

```
historia.txt X
historia.txt
1 Esta es la historia de Freddy Vega
2 Freddy Vega tiene 32 años y nació en Colombia
3 AngeloTest es una robot que contribuye a este proyecto Open Source.
4
5 Experimento 1
6 Experimento 2
7 Master 1
8 Master 2
9 Master 3
10 Master 4
11 Master 5
12 Master 6
13 Master 7
14 Mañana nos enfocaremos en su vida laboral
15
16
```

Rebase es una de las dos utilidades de Git que se especializa en integrar cambios de una rama a otra. Es el proceso de mover o combinar una secuencia de confirmaciones a una nueva confirmación base. Pero debido a que sobrescribe la historia del proyecto, se debe tener cuidado al usarlo para evitar conflictos, sobretodo cuando hay releases previos.

# GIT STASH: GUARDAR CAMBIOS EN MEMORIA Y RECUPERARLOS DESPUÉS



## NOTA:

Por defecto git stash NO almacena los archivos no preparados (untracked o que no se hayan ejecutado con git add) y los archivos ignorados.

Si se necesita guardar en el stash estos archivos, ejecutar los siguientes comandos:

```
git stash -u #El stash considera los Untracked files
```

```
git stash -a #El stash considera los archivos ignorados
```

```
git stash save -u "mensaje"  
#Considera los Untracked y se agrega un comentario
```

## Comandos importantes

```
git stash save "mensaje"  
#Comando para comentar los stashes con una descripción
```

```
git stash apply #Aplicar los cambios en el código en el que estás trabajando y conservarlos en tu stash
```

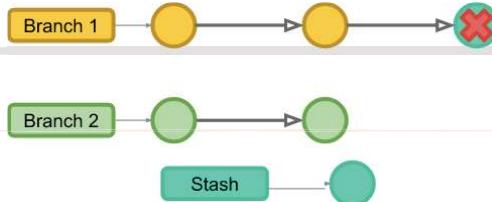
```
git stash show  
#Comando para visualizar un resumen de un stash
```

```
git stash show -p  
#Comando para ver todas las diferencias de un stash
```

```
git stash pop stash@{n}  
#Comando para elegir que número "n" de stash se desea volver a aplicar
```

```
git stash drop stash@{n}  
#Comando para eliminar un determinado número "n" de stash.
```

El comando **git stash** almacena temporalmente (o guarda 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.



### • git stash

El comando **git stash** coge los cambios sin confirmar (tanto los que están preparados como los que no los están), los guarda aparte para usarlos más adelante y, acto seguido, los deshace en el código en el que estás trabajando.

```
Proyecto1 git/master  
> git stash  
Directorio de trabajo guardado y estado de índice WIP on master: f43b9fa Experimento 2
```

### • git stash pop

Puedes volver a aplicar los cambios de un stash mediante el comando **git stash pop**. Al hacer **pop** del stash, se eliminan los cambios de este y se vuelven a aplicar en el código en el que estás trabajando. De forma predeterminada, **git stash pop** volverá a aplicar el último stash creado.

```
Proyecto1 git/master  
> git stash pop  
En la rama master  
Cambios no rastreados para el commit:  
(usa "git add <archivo>..." para actualizar lo que será confirmado)  
(usa "git restore <archivos>..." para descartar los cambios en el directorio modificado: blogpost.html)
```

sin cambios agregados al commit (usa "git add" y/o "git commit -a")  
Botado refs/stash@{0} (347226b85dbe2d7c42910200e6bc7886c97e8a9d)

### • git stash branch nombre-rama

```
Proyecto1 git/master  
> git stash branch english-version  
Cambiado a nueva rama 'english-version'  
En la rama english-version  
Cambios no rastreados para el commit:  
(usa "git add <archivo>..." para actualizar lo que será confirmado)  
(usa "git restore <archivos>..." para descartar los cambios en el directorio modificado: blogpost.html)
```

sin cambios agregados al commit (usa "git add" y/o "git commit -a")  
Botado refs/stash@{0} (2a09318919d16f73e08c89df63ea27f50bc93f50)

Puedes usar **git stash branch** para crear una rama nueva basada en la confirmación a partir de la cual creaste el stash y, después, se hace **pop** en ella con los cambios del stash.

### • git stash list

Se puede listar los stash realizados con el comando **git stash list**. Este comando muestra el número de stash @{n}

```
stash@{0}: WIP on master: f43b9fa Experimento 2  
(END)
```

### • git stash drop

Si decides que ya no necesitas algún stash en particular, puedes eliminarlo mediante el comando **git stash drop**.

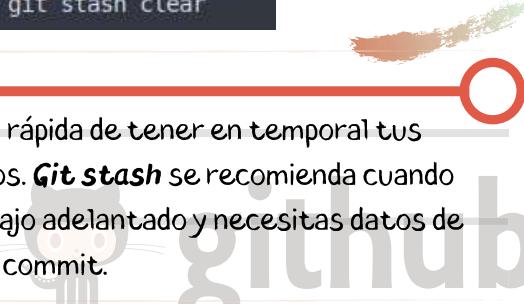
```
Proyecto1 git/master  
> git stash drop  
Botado refs/stash@{0} (ce8c16a81a67d846d6741d9760fe622ccffc6164)
```

### • git stash clear

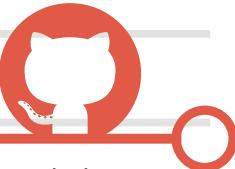
Comando para eliminar todos los stashes.

```
Proyecto1 git/master  
> git stash clear
```

**Git stash** es uno de los comandos más útiles de Git. Es una forma rápida de tener en temporal tus cambios, poder moverte entre ramas y luego recuperar esos cambios. **Git stash** se recomienda cuando haces pequeños cambios que no merecen ramas o cuando llevas trabajo adelantado y necesitas datos de otra rama pero no estás listo para hacer commit.



# GIT CLEAN; LIMPIAR TU PROYECTO DE ARCHIVOS NO DESEADOS



GIT CLEAN ACTÚA EN ARCHIVOS SIN SEGUIMIENTO.



Los archivos sin seguimiento son aquellos que se encuentran en el directorio del repositorio, pero que no se han añadido al índice del repositorio con `git add`.

Los archivos o carpetas sin seguimiento especificados como `.gitignore` no se eliminarán si ejecutamos `git clean -f`

## Comandos Importantes

`git clean -dn` #Comando para verificar que elementos se eliminan incluyendo directorios

`git clean -df` #Comando para eliminar archivos incluyendo directorios

`git clean -xdn` #Comando para verificar que elementos se eliminan incluyendo los archivos ignorados y directorios

`git clean -xdf` #Comando para eliminar archivos incluyendo los archivos ignorados y directorios

`git clean -q` #Nos muestra en pantalla solo los mensajes de errores, no imprime los nombres de los archivos eliminados

`git clean -Xf` #Comando para eliminar solo los archivos ignorados incluidos en `.gitignore`

`git clean -i` #Comando interactivo de git clean, para ejecutar el proceso por medio de opciones, se puede combinar con otros comandos.

```
git clean -i  
would remove the following items:  
 READMEv0.md  
*** Commands ***  
 1: clean 2: filter by pattern 3: select by numbers  
 4: ask each 5: quit 6: help  
 what now?
```

Git Clean es un método adecuado para eliminar los archivos sin seguimiento en un directorio de trabajo del repositorio.

## • git clean

Si ejecutamos `git clean` por defecto, la consola mostrará un error.

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)  
$ git clean  
fatal: clean.requireForce defaults to true and neither -i, -n, nor -f given; ref using to clean
```

Esto porque por defecto y por seguridad Git Clean esta configurado para ser ejecutado con el parametro `-f` (force).

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)  
$ git config --global -l  
user.name=Angelo Yenque T.  
user.email=avennuet@gmail.com  
filter.lfs.clean=git-lfs clean -- %F  
filter.lfs.smudge=git-lfs smudge -- %F  
filter.lfs.process=git-lfs filter-process  
filter.lfs.required=true
```



## • git clean --dry-run

Si ejecutamos `git clean --dry-run` o `git clean -n`, Git realizará un simulacro de borrado y podrás ver los archivos que se van a eliminar sin que se eliminen realmente.

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)  
$ git clean -n  
Would remove css/estilos2.css  
Would remove historia para borrar.txt
```

## • git clean -f

Si ejecutamos `git clean -f`, Git inicia la eliminación real de los archivos sin seguimiento del directorio actual.

## • git clean -d

Ya que se ignora los directorios de forma predeterminada, podemos agregar la opción `-d` a `git clean`, para indicar a Git que también quieras eliminar los directorios sin seguimiento,

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)  
$ git clean -dn  
Would remove Error/  
Would remove README2.md  
Would remove historia2.txt
```

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)  
$ git clean -df  
Removing Error/  
Removing README2.md  
Removing historia2.txt
```

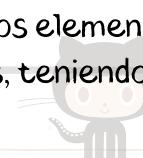
## • git clean -x

`git clean -x` indica a Git que incluya también los archivos ignorados. Al igual que ocurría con las anteriores invocaciones de `git clean`, se recomienda realizar un simulacro antes de la eliminación final. La opción `-x` actuará en los archivos ignorados.

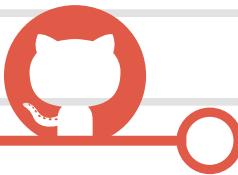
```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)  
$ git clean -xdn  
Would remove Error/  
Would remove blogpostv0.html  
Would remove historiav2.txt  
Would remove imagenes/dragon - Copy.png
```

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)  
$ git clean -xdf  
Removing Error/  
Removing blogpostv0.html  
Removing historiav2.txt  
Removing imagenes/dragon - Copy.png
```

Con `git clean`, podemos eliminar archivos y/o carpetas, para tener una versión limpia de nuestro proyecto, debido a que en ocasiones tenemos elementos que han sido generados por herramientas de combinación o externas, teniendo siempre cuidado de eliminar algo por error.



# GIT CHERRY-PICK; TRAER COMMITS VIEJOS AL HEAD DE UN BRANCH



git cherry-pick es una herramienta útil, pero no siempre es una práctica recomendada, ya que puede generar duplicaciones de confirmaciones.

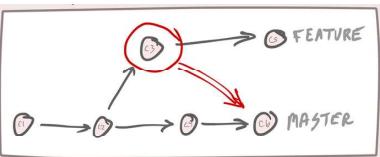
git log --oneline  
#Comando para buscar rápidamente el Hash que necesitamos para ejecutar cherry-pick

```
ayenq@ANGELO MINGW64 /  
mejorado)  
$ git log --oneline  
-d0370e (HEAD -> readme  
f854a03 Ejemplos en Wi  
a9615ca (origin/master  
6c79b73 Cambio al tag!  
648e0db Actualizando r
```

## Comandos Importantes

git cherry-pick HASH -e  
#Comando para ejecutar cherry-pick pero permite editar el mensaje del commit original

git cherry-pick HASH -n  
#Comando para ejecutar cherry-pick pero solo trae los cambios y no ejecuta un commit



El comando **cherry-pick** es útil en algunos casos, por ejemplo, cuando se necesita corregir algún error explícito. Permite ejecutar un commit puntual de una rama en otra, pero no se debe aplicar equivocadamente en lugar de merge o rebase.

## • GIT CHERRY-PICK [HASH]

Este comando permite elegir una confirmación de una rama y aplicarla a otra. Permite que las confirmaciones arbitrarias de Git se elijan por referencia y se añadan al actual HEAD de trabajo.



Para usar cherry-pick lo único que necesitamos saber es el commit específico que queremos aplicar en nuestra rama.

### Ejemplo:

Queremos traer el commit **f854a03** de la rama "readme-mejorado". Nos ubicamos en la rama donde vamos a llevar esa confirmación, en este caso, **master** y ejecutamos:

```
git cherry-pick f854a03
```

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/P  
mejorado)  
$ git checkout master  
Switched to branch 'master'  
Your branch is up to date with 'origin/master'.
```

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/P  
$ git cherry-pick f854a03  
[master 49417de] Ejemplos en Windows, Linux y Mac  
Date: Thu Sep 10 16:42:25 2020 -0500  
1 file changed, 2 insertions(+)
```

Ahora si revisamos el log de la rama **master**, verificamos que se ha creado el mismo commit y que se aplicaron todos los cambios como si lo hubieramos realizado en el mismo **master**.

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)  
$ git log --oneline  
49417de (HEAD -> master) Ejemplos en Windows, Linux y Mac  
a9615ca (origin/master, origin/HEAD) Actualizando Readme
```

Hay que tener en cuenta que este comando crea un nuevo commit, por lo que antes de usar **cherry-pick** no debemos de tener ningún archivo modificado que no haya sido incluido en un **commit**.

También, en caso de querer aplicar más de un commit, nos basta con indicárselos a **cherry-pick**.

```
git cherry-pick [HASH1] [HASH2]
```

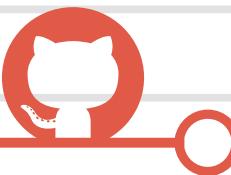
Podemos usar la opción **-x** en caso deseamos añadir una referencia al commit original

```
git cherry-pick [HASH] -x
```

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/D  
$ git cherry-pick 267b2ec -x  
[master 4afc8ee] Ignorando .bak  
Date: Thu Sep 10 17:46:45 2020 -0500  
1 file changed, 2 insertions(+)
```

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)  
$ git log --pretty="%h %s %b"  
4afc8ee Ignorando .bak (cherry picked from commit 267b2ec504700ee683af390d287ab344aeee00b)
```

# RECONSTRUIR COMMITS EN GIT CON AMEND



Con el comando `git commit --amend`, Git sustituye por completo al commit a corregir, y esta será una entidad nueva con su propia referencia.

Procura no corregir una confirmación en la que otros desarrolladores hayan basado su trabajo, ya que crea una situación confusa para ellos de la que resulta complicado recuperarse.



No se recomienda 'modificar' un commit ya enviado a un repositorio, pero en caso se tenga que volver a enviar, podemos hacerlo adicionando la opción `--force` a `git push`. Posiblemente luego debemos solucionar algún conflicto de forma manual. Tener precaución en su uso.

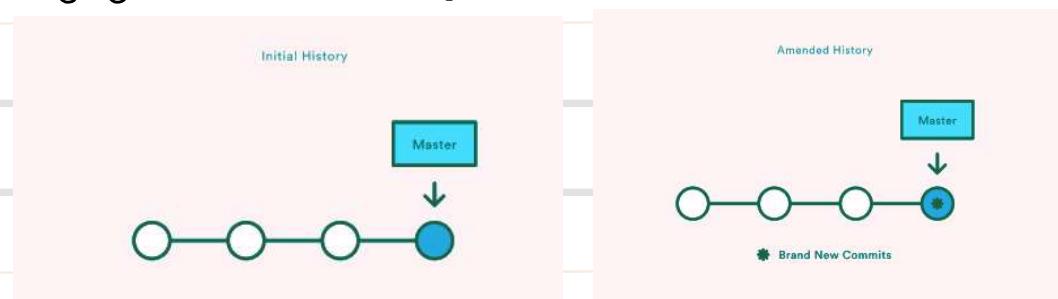
`git push origin master --force`

`git commit --amend --no-edit`  
Adicionando el comando `--no-edit` permite hacer las correcciones en la confirmación sin cambiar el mensaje del commit original.

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)
$ git commit --amend --no-edit
[master 64717c7] Personalizando el archivo historia
Date: Thu Sep 10 21:55:00 2020 -0500
1 file changed, 1 insertion(+), 1 deletion(-)
```

## GIT COMMIT --AMEND

El comando `git commit --amend` es una manera práctica de modificar la confirmación más reciente. Sirve para realizar dos cosas: cambiar la confirmación del mensaje, o cambiar la parte instantánea que acabas de agregar sumando, cambiando y/o removiendo archivos.



### Ejemplo:

Supongamos que acabas de realizar una confirmación y te das cuenta de que hay un error en el mensaje de la confirmación.

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)
$ git commit -am "Personalizando archivo readme.md"
[master a2d27bf] Personalizando archivo readme.md
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)
$ git commit --amend
[master 8faa220] Personalizando archivo historia.txt
Date: Thu Sep 10 21:15:59 2020 -0500
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)
$ git log --oneline
a2d27bf (HEAD -> master) Personalizando archivo readme.md
8faa220 Personalizando archivo historia.txt
```

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)
$ git log --oneline
8faa220 (HEAD -> master) Personalizando archivo historia.txt
```

Notar que el commit "corregido" tiene un nuevo HASH (cambio de a2d27bf a 8faa220).

### git commit --amend -m

Añadir la opción `-m` te permite escribir un nuevo mensaje desde la línea de comandos sin tener que abrir un editor.

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)
$ git commit --amend -m "Personalizando el archivo historia"
[master cd3d535] Personalizando el archivo historia
Date: Thu Sep 10 21:15:59 2020 -0500
1 file changed, 1 insertion(+), 1 deletion(-)
```

### IMPORTANTE

Las confirmaciones no se pueden editar de ninguna manera.

`git commit --amend` te permite "enmendar" una confirmación, pero en realidad hace una nueva confirmación, por lo que tu antigua confirmación aún estará disponible, y puedes volver a ella con `git checkout`.

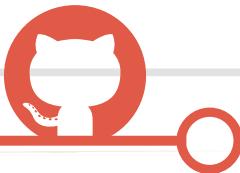
```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)
$ git push origin master
Enter passphrase for key '/c/Users/ayenq/.ssh/id_rsa':
To github.com:ayenque/hyperblog.git
! [rejected]          master -> master (non-fast-forward)
error: failed to push some refs to 'github.com:ayenque/hyperblog.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Mensaje de error cuando se realiza --amend a un commit ya enviado al repositorio remoto.

`git commit --amend` permite añadir nuevos cambios preparados a la confirmación más reciente, incluso permite editar el mensaje de confirmación enviado previamente. Debemos tener cuidado al usar `--amend` en confirmaciones compartidas con otros miembros del equipo, puede que conlleve aplicar resoluciones del conflicto de fusión largas y confusas.



# GIT RESET Y REFLG; 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 REFLG

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.

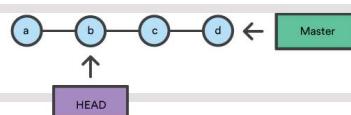
- `git reflog stash`

Comando para ver el registro de referencia de un stash.

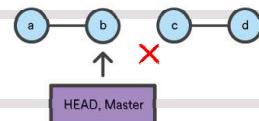
## • GIT RESET

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.

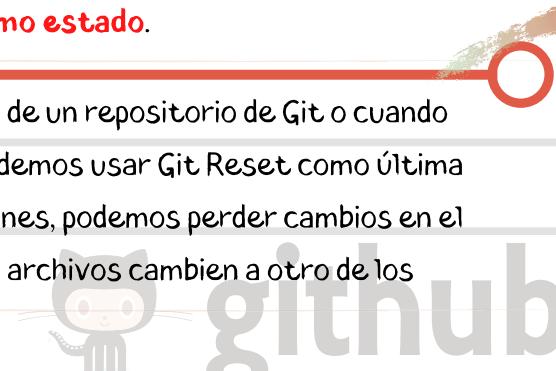
- `git reset --soft [HEAD/HASH]`

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**.

Si necesitamos deshacer los cambios locales en el estado de un repositorio de Git o cuando



tenemos un issue que resulta complicado de solucionar, podemos usar Git Reset como última instancia. Tomando en cuenta que dependiendo de las opciones, podemos perder cambios en el **Working Directory** o en el **Staging Area** o que nuestros archivos cambien a otro de los estados de Git.



# BUSCAR EN ARCHIVOS Y COMMITS DE GIT CON GREP Y LOG



## Comandos útiles

git grep --untracked "palabra" # Busca la palabra en los archivos del directorio de trabajo incluyendo también los archivos untracked (sin seguimiento).

git grep -i "Palabra" # Busca la palabra en los archivos del directorio de trabajo e ignora las diferencias entre mayúsculas y minúsculas

git grep -w "palabra" # Busca la palabra en los archivos del directorio de trabajo considerando la "palabra" exacta y obviando coincidencias.

git grep -v "palabra" # Muestra solo las líneas que no coincidan o no contengan la "palabra" buscada.

git grep -l "palabra" # Lista los archivos que contienen la "palabra" coincidente.

git grep -o "palabra" # Lista todas las coincidencias sin mostrar el detalle del archivo.

git log --oneline --grep "palabra" # Adicionando --grep a las demás opciones de git log, busca la "palabra" en los mensajes de cada commit y muestra las confirmaciones que la contengan.

En Git también se pueden realizar búsquedas con expresiones regulares. Las expresiones regulares son patrones que se utilizan para hacer coincidir combinaciones de caracteres en cadenas.

## GIT GREP

Git tiene un comando llamado **grep** que le permite buscar fácilmente a través de cualquier árbol o directorio de trabajo con **commit** por una cadena de texto. Por defecto, este comando buscará a través de los archivos en el directorio de trabajo.

### GIT GREP "PALABRA-A-BUSCAR"

```
$ git grep "<p>"  
blogpost.html:  
blogpost.html:  
blogpost.html:  
blogpost.html:  
Y este es el p&acute;rrafo de inicio donde vamos a explicar las cosas incre;&iacute;bles  
<p>"  
blogpost.html:4
```

```
$ git grep --count color  
css/estilos.css:3
```

## BUSQUEDA DE REGISTROS

El comando **git log** tiene varias herramientas potentes para encontrar commits específicos por el contenido de sus mensajes o incluso el contenido de las diferencias que introducen.

- git log -S "palabra"

Muestra los commit que contiene la "palabra" tanto en los mensajes de los commits como en el contenido de los cambios. Podemos hacer **git show** a los HASH del resultado para corroborar la búsqueda.

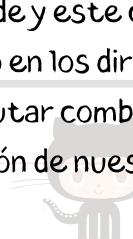
```
$ git log --oneline --grep "cabecera"  
6c79b73 Cambio al tagline y el color del footer :)  
97bf6c5 Logo nuevo y mejor color de Header  
bb6eade Agregando version actualizada despues de conflictos  
3dfe630 Solucion el conflicto de las ramas al fusionar  
d08775a Agregue suscripcion, cambie cabecera y color azul  
23d0066 (origin/cabecera) Modificando la cabecera y el color del texto  
741a2a2 Finalizada la cabecera con diseño azul  
caf6e44 Commit al master del blogspot en su version mas reciente
```

- git log --grep "palabra"

Muestra los commit que contiene la expresión "palabra" solo en los mensajes de cada confirmación de los commits.

```
$ git log --oneline --grep "cabecera"  
d08775a Agregue suscripcion, cambie cabecera y color azul  
23d0066 (origin/cabecera) Modificando la cabecera y el color del texto  
741a2a2 Finalizada la cabecera con diseño azul  
429caa5 Estructura inicial de la cabecera
```

- A medida que nuestro proyecto se va haciendo más grande y este conformado de múltiples archivos, Git nos ofrece herramientas de búsqueda tanto en los directorios de trabajo como en el log de confirmaciones. Las opciones se pueden ejecutar combinadas, que lo vuelve una opción muy útil y poderosa para la administración de nuestro proyecto.



github

# COMANDOS Y RECURSOS COLABORATIVOS EN GIT Y GITHUB



## Comandos útiles

git comando --help

#La opción --help de cualquier comando de Git ejecuta un documento donde se puede leer y revisar todo el funcionamiento "por dentro" de dicho comando.

[HTTP://WEBMITEDU/GIT/www/GIT-SHORTLOGHTML](http://WEBMITEDU/GIT/www/GIT-SHORTLOGHTML)

git blame -e "archivo"

#La opción -e muestra la dirección de correo electrónico de los autores en lugar de su nombre de usuario.

git blame -c "archivo" #Ejecuta el comando blame pero "identa" cada línea del archivo para mostrarlo más ordenado.

git blame -M "archivo"  
#Detecta las líneas que se han movido o copiado dentro del mismo archivo. Esto indicará el autor original de las líneas en lugar de indicar el último autor que las ha movido o copiado.

**GitHub Insights** proporciona métricas e informes analíticos para ayudar a los equipos de ingeniería a comprender y mejorar su proceso de entrega de software.

Pulse
Contributors
Community
Traffic
Commits
Code frequency
Dependency graph
Network
Forks

El propósito de cualquier sistema de control de versiones es registrar cambios en su código. Esto te da el poder de volver a la historia de tu proyecto para ver quién contribuyó con qué, averiguar dónde se introdujeron los errores. Pero, tener toda esta historia disponible es inútil si no sabes cómo navegar por ella. Git ofrece herramientas colaborativas y de inspección como git **shortlog**, **blame**, etc.



## • GIT SHORTLOG

Agrupa cada confirmación por autor y muestra la primera línea de cada mensaje de confirmación. Es una forma fácil de ver quién ha estado trabajando en qué. Tiene opciones que se pueden ejecutar combinadas:

git shortlog -n

Ordena la salida según el número de confirmaciones por autor en lugar de hacerlo por orden alfabético.

git shortlog -s

Proporciona sólo un resumen del recuento de la confirmación.

git shortlog -e

Muestra la dirección de correo electrónico de cada autor.

git shortlog --all --merges

Muestra solo las confirmaciones de fusión o merge.

git shortlog --all --no-merges

Muestra todas las confirmaciones y excluye las que son por merge o fusión.

## • GIT "ALIAS"

Si no quieres teclear el nombre completo de cada comando de Git, puedes establecer fácilmente un alias para cada comando mediante **git config**.

git config --global alias.alias-que-quiero-usar 'comando-a-ejecutar'

```
$ git config --global alias.stats "shortlog -sn --all --no-merges"
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/h
$ git stats
 54 Angelo Yenque T
  4 Angelo Yenque / Test
  2 Angelo Y. T
```

## • GIT BLAME

El funcionamiento de **git blame** es la visualización de metadatos de autor adjuntos a líneas específicas confirmadas en un archivo. Esto se usa para examinar puntos específicos del historial de un archivo y poner en contexto quién fue el último autor que modificó la línea.

git blame "nombreArchivo"

```
git blame historia.txt
cf2fde68 (Angelo Yenque T 2020-09-10 21:11:59 -0500 1) Esta es la historia de Angelo Yenque
f54b7389 (Angelo Yenque T 2020-05-21 18:10:20 -0500 2) Freddy Vega tiene 32 años y nació en Colom...
f54b7389 (Angelo Yenque T 2020-05-21 18:10:20 -0500 3) Freddy Vega tiene 32 años y nació en Colom...
cf96b166 (Angelo Yenque / Test 2020-06-21 13:01:40 -0500 5) AngeloTest es una robot que contribuye a es...
cf96b166 (Angelo Yenque / Test 2020-06-21 13:01:40 -0500 6) AngeloTest es una robot que contribuye a es...
86gef4166 (Angelo Yenque T 2020-07-06 11:09:35 -0500 7) Experimento 1
86gef4166 (Angelo Yenque T 2020-07-06 11:09:35 -0500 8) Experimento 2
F43b9Fab (Angelo Yenque T 2020-07-06 11:09:36 -0500 9) Experimento 3
F43b9Fab (Angelo Yenque T 2020-07-06 11:09:36 -0500 10) Experimento 4
e33e175f (Angelo Yenque T 2020-07-06 11:09:36 -0500 11) Master 1
e33e175f (Angelo Yenque T 2020-07-06 11:09:36 -0500 12) Master 2
C749e930 (Angelo Yenque T 2020-07-06 11:11:18 -0500 13) Master 3
C749e930 (Angelo Yenque T 2020-07-06 11:11:18 -0500 14) Master 4
4f2e7fc3 (Angelo Yenque / Test 2020-06-11 18:27:45 -0500 15) Malina nos enfocaremos en su vida laboral
```

git blame "nombreArchivo" L(linea\_inicio),(linea\_fin)

Agregando la opción **L**, permite filtrar por una cantidad de líneas indicando el **inicio** y **fin**.

```
$ git blame historia.txt -L5,10
cf96b166ee (Angelo Yenque / Test 2020-06-21 13:01:40 -0500 5) AngeloTest es
cf96b166ee (Angelo Yenque / Test 2020-06-21 13:01:40 -0500 6) AngeloTest es
86gef4166 (Angelo Yenque T 2020-07-06 11:09:35 -0500 7) Experimento 1
86gef4166 (Angelo Yenque T 2020-07-06 11:09:35 -0500 8) Experimento 2
F43b9Fab (Angelo Yenque T 2020-07-06 11:09:36 -0500 9) Experimento 3
F43b9Fab (Angelo Yenque T 2020-07-06 11:09:36 -0500 10) Experimento 4
```

## • GIT BRANCH

El comando **git branch** tiene opciones adicionales para listar las ramas existentes en nuestro repositorio. Con la opción **-v**, lista las ramas locales. La opción **-r** lista las ramas remotas, permite comparar que ramas se han enviado a Github. Con la opción **-a** puedes listar todas las ramas.

```
$ git branch -v
  footer          738b613 Footer terminado
  gitignore       5deab7e Ignorando archivos de e
  header          64493b0 Logo mejorado
  list            d908e1b Personalizando historia
* master          a7c76ab [ahead 1] Cambios para
                  readme-mejorado ad03706 Disponible para todas
```

```
$ git branch -r
origin/HEAD -> origin/master
origin/cabecera
origin/fix-typo
origin/footer
origin/header
origin/master
```

```
$ git branch -a
footer
gitignore
header
list
* master
readme-mejorado
remotes/origin/HEAD -> origin/master
remotes/origin/cabecera
remotes/origin/fix-typo
remotes/origin/footer
remotes/origin/header
remotes/origin/master
```

