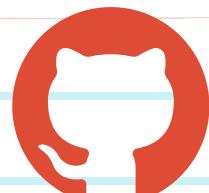


¿QUÉ ES GIT?



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

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, Mac OS 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

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 a reducir significativamente los errores humanos, a tener un mejor mantenimiento de distintos entornos y a detectar fallos de una forma más rápida y eficiente.
- Es la opción perfecta para poder trabajar en equipo en un mismo proyecto.
- Ofrece todas las ventajas del sistema de control de versiones, Git, pero también tiene otras herramientas que ayudan a tener un mejor control de nuestros proyectos.

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.

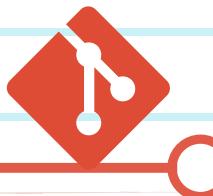
GitHub es una forja para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de ordenador. El software que opera GitHub fue escrito en **Ruby on Rails**. Desde enero de 2010, GitHub opera bajo el nombre de GitHub,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

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 trabajarla de forma remota y colaborar con otros miembros dentro y/o fuera de nuestra organización.



git

¿POR QUÉ USAR UN SISTEMA DE CONTROL DE VERSIONES COMO GIT?



TOP 5 - Softwares de controles de versiones

- Git
- CVS
- Apache Subversion (SVN)
- Mercurial
- Monotone



mercurial



Comandos Iniciales

git init #inicializa el repositorio de git

git add "nombre-archivo" #Agrega todos los cambios en todos los archivos al área de staging

git commit -m "Mensaje" #Agrega finalmente el cambio realizado al SCV y le agrega un mensaje

git status #Muestra el estado del repositorio

git show #Muestra todos los cambios históricos hechos y sus detalles (qué cambió, cuándo y quién los hizo)

git log #Muestra la historia de los cambios en los archivos

git push #Envía los archivos del repositorio a un servidor remoto

El control de versiones es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo de tal manera que sea posible recuperar versiones específicas más adelante.



Mi nombre es Freddy Vega
Soy el CTO de Platzi y en mis tiempos libres **juego tennis**, escribo libros y hago cosas raras con **Raspberry Pis**

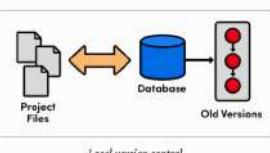
Mi nombre es Freddy Vega
Soy el CEO de Platzi y en mis tiempos libres **corro carreras**, escribo libros y hago cosas raras con **Arduino**

Git

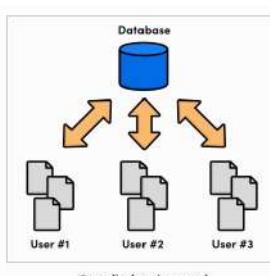
biografia.txt

Los sistemas de control de versiones han ido evolucionando a lo largo del tiempo y podemos clasificarlos en tres tipos:

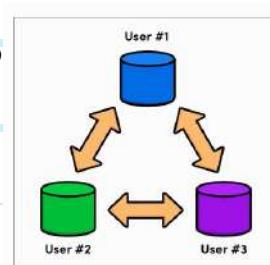
- Sistemas de Control de Versiones Locales, almacenan los cambios en una base de datos y solo se lleva en el computador de cada uno de los desarrolladores por separado.
- Centralizados, no dependen de nuestro dispositivo de computo, sino que los cambios están guardados en un servidor, todos pueden tener acceso pero puede generar conflictos cuando se trabaja en un mismo archivo.
- Distribuidos, los mas preferidos por la comunidad, ya que cada dispositivo de computo interviene como repositorio y si alguno deja de funcionar, puede ser reemplazado por otro componente y no afectara al proyecto en general.
Cada usuario tiene una copia completa del proyecto **el riesgo** por una caída del servidor, un repositorio dañado o cualquier otro tipo de perdida de datos **es mucho menor que en cualquiera de sus predecesores**.



Local version control



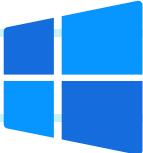
Centralized version control



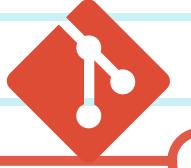
Distributed version control

Un software de control de versiones (VCS) es una valiosa herramienta con numerosos beneficios para un flujo de trabajo de equipos de software de colaboración. Permite realizar la trazabilidad sobre los cambios que se realizan en el código fuente además de recuperar versiones anteriores, lo que hace que sea más fácil de mantener el código y de trabajar con él.





INSTALANDO GIT Y GITHUB BASH EN WINDOWS

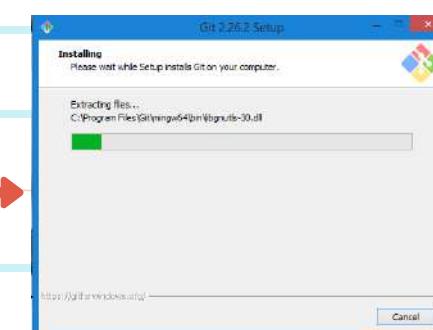
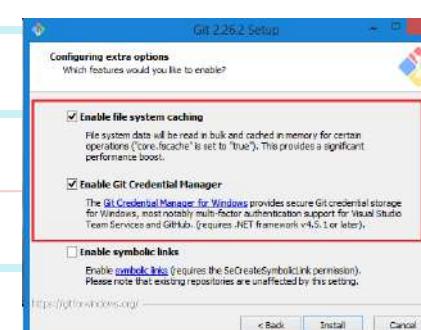
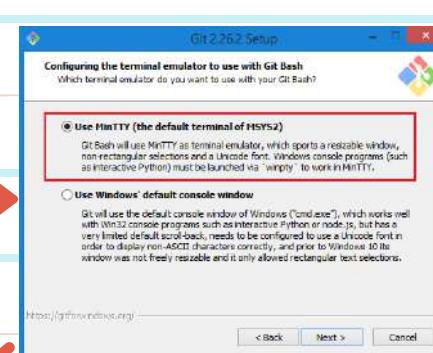
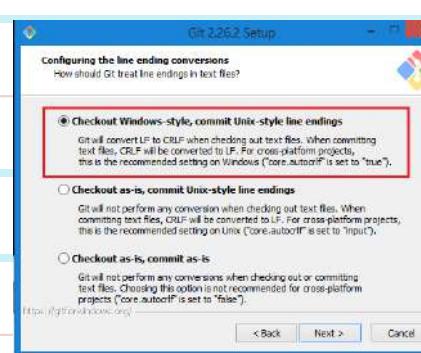
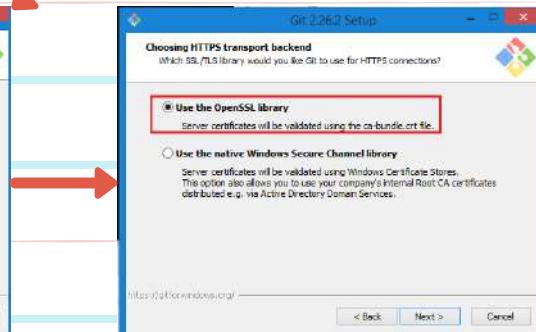
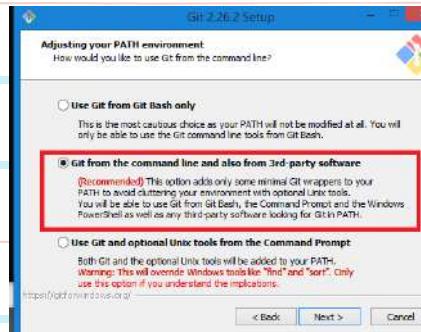
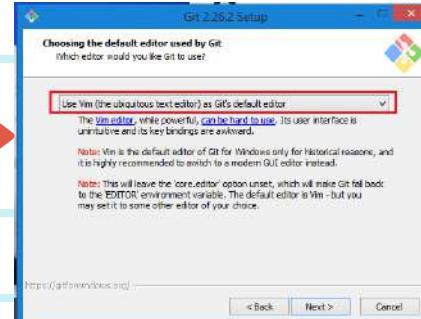
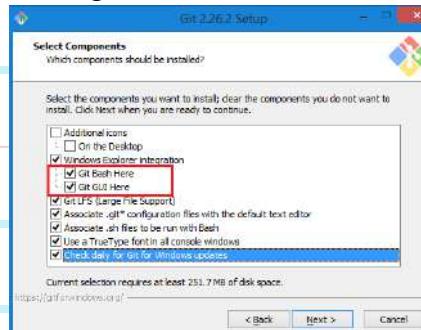


Otra forma de obtener Git fácilmente es mediante la instalación de GitHub para Windows.

Puedes descargar este instalador del sitio web de GitHub para Windows en <http://windows.github.com>

Una vez culminada la instalación, para verificar que GIT esta instalado, abrimos en  "GIT BASH" y escribimos el comando `git --version` para corroborar la versión instalada.

- Visitar <https://git-scm.com/download/win>
 - Elegir la versión necesaria, y la descarga empezará automáticamente



Aunque los programadores utilizan sistemas basados en 'UNIX' (tales como GNU/Linux, Mac OS, etc.) , Git también se puede instalar y ejecutar en Windows. Existe un proyecto llamado **msysGit** , que se puede descargar desde el sitio oficial de GIT e incluye GIT BASH (terminal adaptada para Windows) y otros, que nos van a permitir trabajar y aprovechar todas las funcionalidades de este sistema.



git

END



INSTALANDO GIT EN OSX



Otra forma de obtener Git fácilmente es mediante la instalación de GitHub para Mac.

Su interfaz gráfica de usuario tiene la opción de instalar las herramientas de línea de comandos.

Puedes descargar esa herramienta desde el sitio web de Github para Mac en <http://mac.github.com>

En Mac se utiliza el formato DMG. Los archivos en este formato

son carpetas contenedoras donde se encuentra los programas que queremos instalar en nuestro equipo, de una forma rápida y sencilla.

Una vez culminada la instalación, para verificar que GIT esta instalado, abrimos la Terminal y escribimos el comando `git --version` para corroborar la versión instalada.

La instalación de GIT en Mac es un poco más sencilla. No debemos instalar GitBash porque Mac ya trae por defecto una consola de comandos (la puedes encontrar como "Terminal"). Tampoco debemos configurar OpenSSL porque viene listo por defecto. Recordar también, que en la práctica una consola de Mac y Linux son parecidas (no iguales).

Hay varias opciones para instalar Git en macOS.

1) HOMEBREW

- Instala Homebrew desde https://brew.sh/index_es, con el siguiente comando en la terminal

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

- Después de la instalación, ejecutar el siguiente comando para instalar GIT

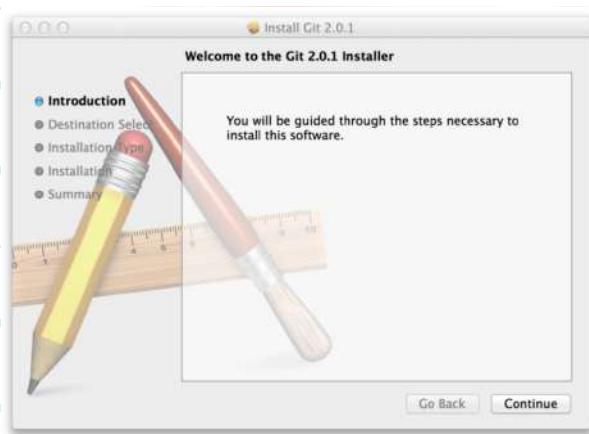
```
$ brew install git
```

2) XCODE

- Apple monta un paquete binario de Git con Xcode (<https://developer.apple.com/xcode>).
- Puedes hacer esto desde el Terminal si intentas ejecutar git por primera vez. Si no lo tienes instalado, te preguntará si deseas instalarlo.

3) BINARY INSTALLER

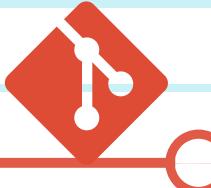
- Si deseas una versión más actualizada, puedes hacerlo a partir de un instalador binario. Un instalador de Git para OSX es mantenido en la página web de Git. Lo puedes descargar en <http://gitscm.com/download/mac>



git



INSTALANDO GIT EN LINUX



Importante

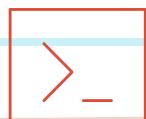
Antes de hacer la instalación, debemos hacer una actualización del sistema. En nuestro caso, los comandos para hacerlo son

sudo apt-get update
y
sudo apt-get upgrade

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

Para verificar que GIT esta instalado y actualizado, abrimos la Terminal y escribimos el comando
git --version

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



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

• DEBIAN/UBUNTU

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

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

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

• FEDORA

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

• GENTOO

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

• ARCH LINUX

```
$ pacman -S git
```

• OPENSUSE

```
$ zypper install git
```

• MAGEIA

```
$ urpmi git
```

• MAGEIA

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

• FREEBSD

```
$ pkg install git
```

• SOLARIS 9/10/11 (OPENCSW)

```
$ pkgutil -i git
```

• SOLARIS 11 EXPRESS

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

• OPENBSD

```
$ pkg_add git
```

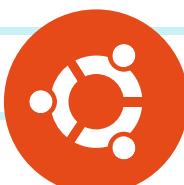
• ALPINE

```
$ apk add git
```

• SLITAZ

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

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

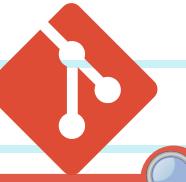


Los usuarios de Linux pueden administrar Git principalmente desde la línea de comandos. Se recomienda usar Linux debido a su alto rendimiento, a que es Open Source, a su poderosa Terminal, para gestión de Servidores, desarrollo Web, Bash, etc.



git

CICLO BÁSICO DE TRABAJO EN GIT



Comandos importantes

`git init nombre-repositorio`
#Comando para iniciar un repositorio, o sea, activar el sistema de control de versiones de Git en tu proyecto

`git add nombre-archivo`
#Agrega un archivo del Working Directory al Staging Area

`git add .` #Agrega todos los archivos del Working Directory al Staging Area

`git commit -m "mensaje del commit"` #Agrega los archivos del Staging Area al Git Repository

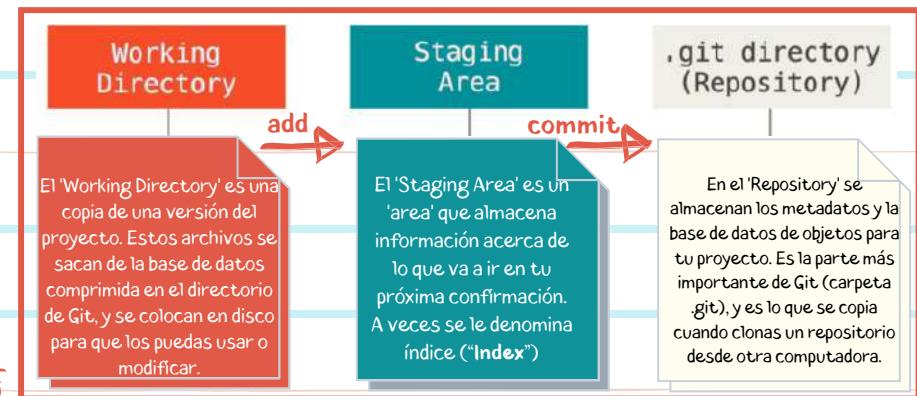
`git commit -am "mensaje del commit"` #Fusión entre git add y git commit -m, agrega los archivos y ejecuta un commit a la vez. Funciona solo para archivos que previamente fueron agregados (tracked).

`git status` #Permite ver el estado de todos nuestros archivos y carpetas

La rama "master" en Git, no es una rama especial. Es como cualquier otra rama. La única razón por la cual aparece en casi todos los repositorios es porque es la que crea por defecto el comando `git init` y la gente no se molesta en cambiarle el nombre.

Esto es lo más importante que debes recordar acerca de Git si quieres que el resto de tu proceso de aprendizaje prosiga sin problemas.

Un proyecto en Git tiene tres 'secciones' principales:



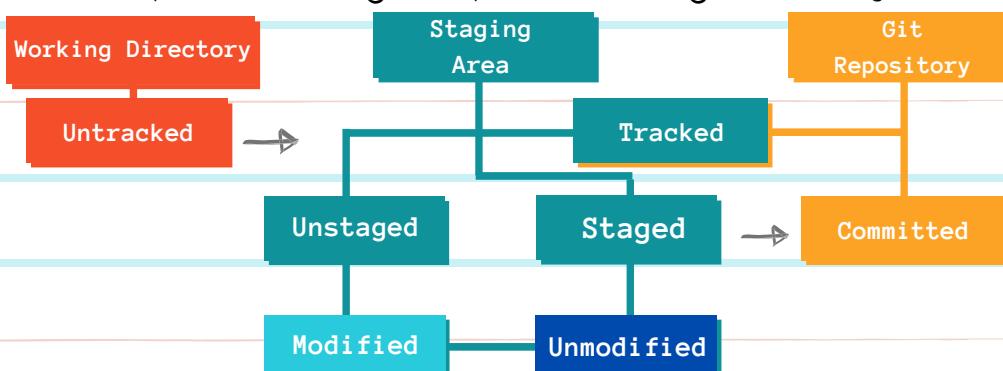
El flujo de trabajo básico en Git es algo así:

- 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.



Guardando cambios en el Repositorio

Ya tienes un repositorio Git, el siguiente paso es realizar algunos cambios y confirmarlos.

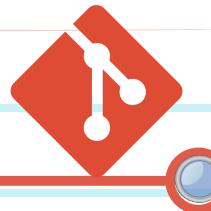


- Untracked**: Cualquier otro archivo en tu disco de trabajo que no estaba en tu última confirmación y que no está en el staging area (no add).
- Staged**: Archivos que están en staging area y se han marcado en su versión actual para que vaya en tu próximo commit.
- Unstaged**: Archivos que git tiene registro de sus cambios pero que están desactualizados.
- Modified**: Significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos.
- Committed**: Archivos que están almacenados de manera segura en tu base de datos local.

Entender el ciclo básico del trabajo en Git, es uno de los conceptos más importantes que se debe entender para tener éxito en el trabajo de nuestro proyecto. Con esto, tenemos claro que comandos debemos ejecutar y sobre todo cuando; así evitamos los conflictos en nuestros archivos y nos aseguramos de guardar nuestro trabajo de forma segura y con éxito.



¿QUÉ ES UN BRANCH (RAMA)?



Por regla general a **master** se la considera la rama principal y la raíz de la mayoría de las demás ramas. Lo más habitual es que en master se encuentre el "código definitivo", que luego va a producción, y es la rama en la que se mezclan todas las demás.

origin/master #Es una rama remota, es una copia local de la rama llamada "master", en el repositorio remoto llamado "origin"

GIT FLOW

Es una guía que nos da ciertos estándares para manejar la ramificación de nuestros proyectos, mediante un conjunto de extensiones que nos ahoran bastante trabajo a la hora de ejecutar todos estos comandos, simplificando la gestión de las ramas de nuestro repositorio.

Se puede instalar desde:



<https://github.com/nvie/gitflow/wiki/Installation>

```
$ git-flow init
Which branch should be used for bringing forth production
- master
Branch name for production releases: [master]
Branch name for "next release" development: [develop]

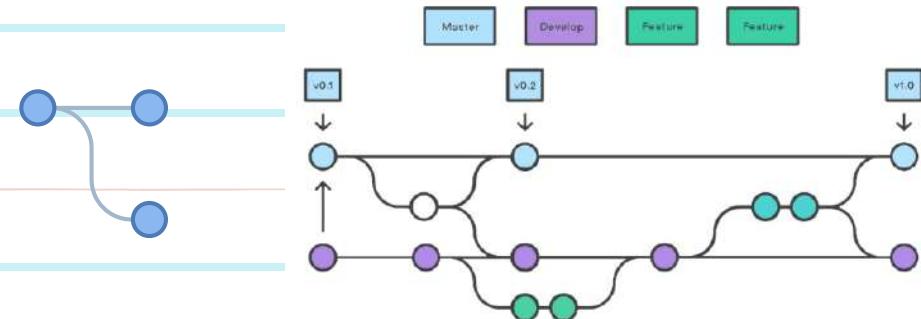
How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? [ ] v
Hooks and filters directory? [D:/Nube/OneDrive/Dокументos/blog/.git/hooks]
```

HEAD, es el commit en el que está tu repositorio posicionado en cada momento. Suele coincidir con el último commit de la rama en la que estés.

Una rama es un nombre que se da a un commit, a partir del cual se empieza a trabajar de manera independiente y con el que se van a enlazar nuevos commits (de esa misma rama).

• RAMA MASTER

Cuando creamos un repositorio (con `git init`), se genera por defecto una rama que se llama **master**. Cualquier commit que pongamos en esta rama debe estar preparado para subir a producción.

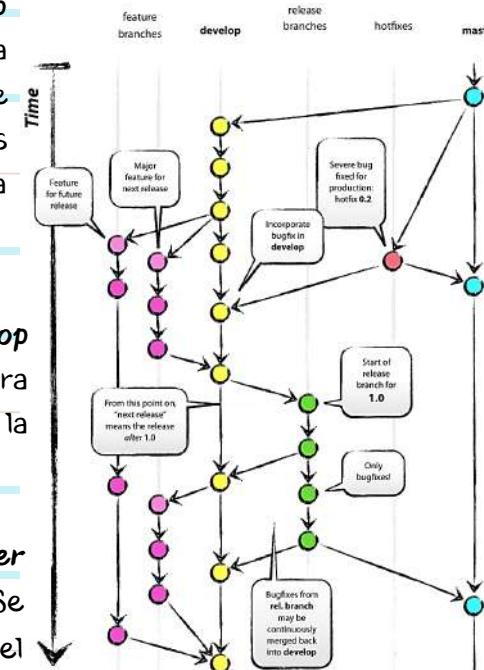


• RAMA DEVELOP

Rama en la que está el código que conformará la siguiente versión planificada del proyecto.

• RAMAS RELEASE

Ramas que se generan a partir de la rama **develop** y se incorporan a esta o a **master**. Se utilizan para preparar el siguiente código en producción, se hacen los últimos ajustes y se corrigen los últimos bugs antes de pasar el código a producción incorporándolo a la rama **master**.



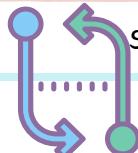
• FEATURE ó TOPIC BRANCHES

Ramas que se generan a partir de la rama **develop** y se incorporan siempre a esta. Se utilizan para desarrollar nuevas características de la aplicación.

• RAMAS HOTFIX

Ramas que se generan a partir de la rama **master** y se incorporan siempre a esta o **develop**. Se utilizan para corregir errores y **bugs** en el código en producción. Funcionan de forma parecida a las ramas **Releases**, siendo la principal diferencia que los hotfixes no se planifican.

Podemos desarrollar nuestro trabajo de manera profesional mediante el uso de las ramas. **Master** será nuestra rama principal con la versión estable de nuestro proyecto y con la rama **development** podemos verificar los cambios antes de lanzar una nueva versión. Asimismo, podemos gestionar errores con ramas **hotfix** y los cambios adicionales con ramas **feature**. Siempre debemos mantener el orden y estructura, nos podemos ayudar con el estándar de **Git Flow**.



git

CREA UN REPOSITORIO DE GIT Y HAZ TU PRIMER COMMIT



Recordar

Debemos configurar nuestros datos de usuario en el primer uso de GIT, para que registre los cambios.

```
git config --global
```

```
    user.email
```

```
"tu@email.com" #Para
```

```
configurar un correo
```

```
git config --global
```

```
    user.name "Tu Nombre"
```

```
#Para configurar nuestro nombre
```

```
git config --list #Para revisar las otras configuraciones
```

ESTRUCTURA DE LA CARPETA .GIT

```
-- COMMIT_EDITMSG  
-- FETCH_HEAD  
-- HEAD  
-- ORIG_HEAD  
-- branches  
-- config  
-- description  
-- hooks  
|   -- applypatch-msg  
|   -- commit-msg  
|   -- post-commit  
|   -- post-receive  
|   -- post-update  
|   -- pre-applypatch  
|   -- pre-commit  
|   -- pre-rebase  
|   -- prepare-commit-msg  
|   -- update  
-- index  
-- info  
|   -- exclude  
-- logs  
|   -- HEAD  
|   -- refs  
-- objects  
-- refs  
|   -- heads  
|   -- remotes  
|   -- stash  
|   -- tags
```

<http://es.gitready.com/advanced/2009/03/23/whats-inside-your-git-directory.html>

Luego de crear el directorio del proyecto, nos dirigimos a la carpeta raíz de este y usamos el siguiente comando.

• git init

Esto crea un subdirectorio nuevo llamado `.git`, el cual contiene todos los archivos necesarios del repositorio. Todavía no hay nada en tu proyecto que esté bajo seguimiento.

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/ProyectoE  
$ git init  
Initialized empty Git repository in D:/Nube/OneDrive/Documentos/Platzi/ProyectoE  
./git/
```

Luego de ejecutar este comando podemos empezar a trabajar nuestros archivos.

• git status

Este comando te mostrará los **diferentes estados de los archivos** en tu directorio de trabajo. Qué archivos están modificados y sin seguimiento y cuáles con seguimiento pero no confirmados aún. En su forma normal, también te mostrará algunos consejos básicos sobre cómo mover archivos entre estas etapas.

```
$ git status  
On branch master  
  
No commits yet  
  
nothing to commit (create/copy files and use "git add" to track)
```

```
$ git status  
On branch master  
  
No commits yet  
  
Untracked files:  
(use "git add <file>..." to include in what will be committed)  
historia.txt  
  
nothing added to commit but untracked files present (use "git add" to track)
```

Si tenemos un cambio en algún archivo, podemos agregar esos cambios a git.

• git add

Añade contenido del directorio de trabajo al **staging area** (o 'index') para el próximo commit.

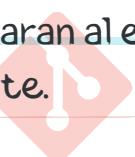
```
$ git add historia.txt  
warning: LF will be replaced by CRLF in historia.txt.  
The file will have its original line endings in your working directory
```

• git commit

El comando **git commit** captura una instantánea de los cambios preparados en ese momento del proyecto. Las instantáneas confirmadas pueden considerarse como versiones "seguras" de un proyecto: Git no las cambiará nunca a no ser que se lo pidas expresamente.

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/ProyectoE (master)  
$ git commit -m "Este es el primer commit de este archivo"  
[master (root-commit) 80f1db0] Este es el primer commit de este archivo  
1 file changed, 2 insertions(+)  
create mode 100644 historia.txt
```

Un repositorio se crea desde un directorio con **git init**. Luego de realizar una modificación a un archivo, debemos agregarlo con **git add** para luego confirmar dichos cambios con **git commit**. Recordar que siempre es buena práctica revisar el estado de nuestros archivos con **git status** y que estos, no pasaran al estado **committed** si no lo agregamos al **staging area**, previamente.



git

ANALIZAR CAMBIOS EN LOS ARCHIVOS DE

TU PROYECTO CON GIT

21/05/2020

Códigos



```
git show historia.txt
```

```
git log historia.txt
```

```
git commit historia.txt -m "Este es el primer commit"
```

```
git diff  
c497331c19178ea6432f3  
1117bbd345091382a2f  
6cd8bcf3470aeac0b5888  
362b37d1d84cea40851
```

Importante

Siempre se debe colocar un comentario en el commit si no se coloca se puede romper el commit y no se consideran los cambios.

No olvidar los códigos:

```
git init // para iniciar un repositorio en la carpeta actual
```

```
git add historia.txt // para agregar algún cambio al staging
```



• Git show

Permite ver el último commit y los cambios realizados en el archivo.

```
ayenque@ayenque:~/Documentos/Platzl/Git_Github/Proyectos$ git show historia.txt  
commit 6cd8bcf3470aeac0b58882b5701d84cea40851 (HEAD -> master)  
Author: Angelo Venque T <ayenque@gmail.com>  
Date: Thu May 21 18:16:00 2020 -0500  
  
Este sería el segundo el primer se borró/desapareció del commit  
diff --git a/historia.txt b/historia.txt  
index fcd35a3..ba29084 100644  
--- a/historia.txt  
+++ b/historia.txt  
@@ -3,4 +3,5 @@ Esta es la historia de Angelo Venque • inicio del archivo  
Angelo tiene 32 años y nació en Talara - Piura.  
Quiere ser músico!  
  
Hoy hablaremos de su historia, pero primero vamos a la clase de Git y GitHub!
```

• Git commit -m "MENSAJE CONCISO DEL CAMBIO"

Importante enviar un mensaje dentro del commit, **no se puede enviar un commit sin mensaje.**

--> ESC + SHIFT + Z Z (Fuerza y guarda el envío en editor VIM)

• Git log

Permite ver la historia de sus commits.

• Git diff

Permite comparar los cambios realizados y enviados en dos determinados commit, para esto se debe copiar la llave o **indicador** de cada commit

```
git diff [commit vers 1] [commit vers 2]
```

```
ayenque@ayenque:~/Documentos/Platzl/Git_Github/Proyectos$ git diff 64b738967ac357518d6d28aa122baa043d7de71 e72477fcff9c6d161ca584034826a031a28b7791  
diff --git a/historia.txt b/historia.txt  
index f5084fa..0b598e2 100644  
--- a/historia.txt  
+++ b/historia.txt  
@@ -1,6 +1,12 @@  
Esta es la historia de Angelo Venque  
  
-Angelo tiene 32 años y nació en Colombia.  
-Viviendo en todo el mundo.  
-Angelo tiene 32 años y nació en Talara - Piura.  
-Quiere ser músico!  
  
-En cámara parece alto, pero en realidad no lo es.  
  
-Actualmente ya no trabaja, porque perdió su trabajo a causa de la pandemia, debe ser por algo más quizás, pero en fin.  
  
Hoy hablaremos de su historia, pero primero vamos a la clase de Git y GitHub!
```

Se vieron los principales comandos para analizar los cambios en los archivos: Git Show, para ver el último commit, Git commit, para enviar los cambios al repositorio de Git, Git Log para tener una lista completa de todos los commits y Git Diff para comparar commits.



VOLVER EN EL TIEMPO EN NUESTRO

REPOSITORIO UTILIZANDO RESET Y CHECKOUT



Comandos:

```
git reset [Hash] --hard  
git reset [Hash] --soft  
git reset [Hash] --mixed
```

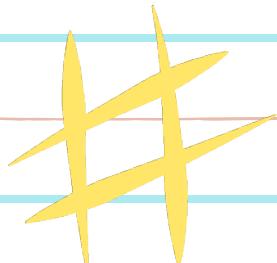
```
git reset HEAD archivo.txt  
//Quita del staging y lo pone  
en working directory.
```

```
git log --stat
```

```
git checkout [Hash] archivo.txt  
git checout master archivo.txt
```

El git checkout tambien es peligroso porque es otra forma de volver a una versión anterior y se puede dejar permanente.

El git checkout es una de las maneras en regresar a una versión o crear una nueva rama cuando se tiene que recuperar algo de una versión pasada.



Queremos volver en el tiempo, es decir a un commit 'antiguo'.



• git reset

Permite volver a una versión anterior, sin poder volver al estado anterior pero tiene dos atributos principales.

--hard

Permite restablecer al estado anterior, todo vuelve como estaba. Es el más peligroso porque borra todo y no mantiene nada.



--soft

Permite restablecer al estado anterior, pero mantiene en staging los commits "eliminados". Es decir, el directorio de trabajo vuelve a la versión del commit seleccionado.

--mixed

Permite restablecer al estado anterior, pero mantiene en Working Directory los commits "eliminados". Es decir, el directorio de trabajo se restablece, pero deja en el directorio local los cambios realizados posterior al commit seleccionado.

```
git reset [Hash del commit] --hard/soft/mixed
```

• git log --stat

Permite ver los cambios específicos que se hicieron, en cuales archivos, por cada commit.

• git checkout

Permite volver a una versión anterior, es decir ver como era el archivo en un determinado commit, en realidad no ha cambiado todavía, lo que hace es pasarlo al staging.

```
git checkout [Hash del commit] archivo.txt
```

• git checkout master

Permite ver a la versión master del archivo, es decir la ultima versión que se había enviado en el ultimo commit.

```
git checkout master archivo.txt
```

```
[ayenquet@ayenque] - [~/Documentos/Platzi/Git_Github/Proyecto1] - [122]  
-$ git checkout master historia.txt  
Actualizada 1 ruta para b11a55a  
[ayenquet@ayenque] - [~/Documentos/Platzi/Git_Github/Proyecto1] - [123]  
-$ git status  
En la rama master  
nada para hacer commit, el árbol de trabajo está limpio  
[ayenquet@ayenque] - [~/Documentos/Platzi/Git_Github/Proyecto1] - [124]  
-
```

Se vio la forma como se trabaja para recuperar versiones pasadas de nuestros archivos, pudiendo mantener (en staging o en el directorio local), los cambios realizados desde un determinado commit o eliminandolos.

FLUJO DE TRABAJO BÁSICO CON UN REPOSITORIO REMOTO



Siempre es muy buena práctica hacer **git pull** antes de intentar hacer push o empezar a trabajar.

Comandos importantes

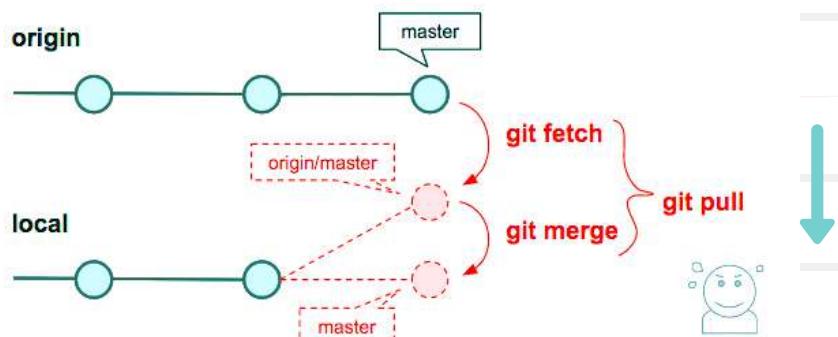
git fetch nombre-rama
#Trae los cambios del repositorio remoto, crea la rama y los deja en esta. Luego debemos fusionar con otra rama de nuestro proyecto.

git pull nombre-remoto nombre-rama
#Trae los cambios del repositorio remoto llamado "nombre-remoto" y los fusiona con la rama "nombre-rama"

git push nombre-remoto nombre-rama
#Envía los cambios de nuestro Local Repository al repositorio remoto llamado "nombre-remoto" y la rama "nombre-rama"



Los repositorios remotos son versiones de tu proyecto que están hospedadas en Internet o en cualquier otra red. Para poder colaborar con otras personas implica gestionar estos repositorios remotos enviando y trayendo datos de ellos cada vez que necesites compartir tu trabajo.



• git fetch

Lo usamos para traer actualizaciones del servidor remoto y guardarlas en nuestro Local Repository. Traemos los cambios que no tenemos pero no lo lo combina automáticamente con tu trabajo ni modifica el trabajo que llevas hecho.



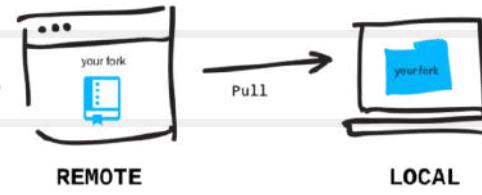
• git merge

Este comando se ejecuta para combinar los últimos cambios traídos del servidor remoto (con git fetch), y nuestro Working Directory.

• git pull

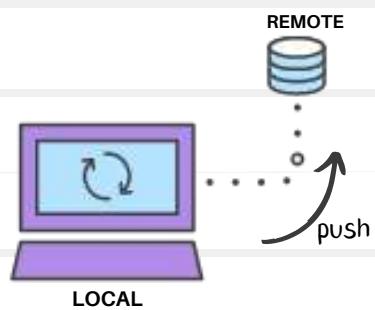
Este comando se emplea para extraer y descargar contenido desde un repositorio remoto y actualizar al instante el proyecto local para reflejar ese contenido.

Básicamente, **git fetch** y **git merge** al mismo tiempo.



• git push

Luego de hacer **git add** y **git commit** debemos ejecutar este comando para mandar los cambios de nuestro proyecto local, al servidor remoto.



Similar al ciclo básico en Git, es necesario entender el flujo de trabajo con un repositorio remoto. Con ello, podemos enviar nuestro trabajo local con el comando **git push** y traer los cambios que otros miembros del equipo realicen, con **git pull**. Esta es la manera en que todos pueden colaborar en el proyecto de una forma eficiente y con recursos independientes.

Importante: ¡Practicar todo lo aprendido!

INTRODUCCIÓN A LAS RAMAS O BRANCHES DE GIT



Puedo utilizar git commit -am "Mensaje" para hacer un add y un commit a la vez, solo para archivos previamente guardados.

Cuando uno hace una rama, en realidad esta haciendo una copia del ultimo commit en otro "lado" para que los cambios sean independientes

git status para revisar en que rama estoy actualmente

Cada vez que nos movemos de una rama a otra los archivos tambien vuelven al estado en el que se encuentren.

Cada vez que estamos en una rama no olvidar realizar add o commit a los cambios realizados en cada rama correspondiente.

RAMAS

En cada confirmación de cambios (commit), Git almacena una instantánea de tu trabajo preparado. Git crea un objeto de confirmación con los metadatos pertinentes y un **apuntador (HEAD)** al objeto árbol raíz del proyecto.



UNA RAMA GIT ES SIMPLEMENTE UN APUNTADOR MÓVIL APUNTANDO A UNA DE ESAS CONFIRMACIONES.

• GIT BRANCH "NOMBRE_RAMA"

Comando para crear una nueva rama, la rama se crea desde el lugar (rama) donde estoy.

branch



Con git show revisamos que el HEAD apunte adicionalmente a la rama actual o por defecto (master) y la rama nueva.

```
commit f561cda058a65effe519caec5272e105029c8cd (HEAD -> hotfix1, nueva-plugin, master, RD)
Author: Angelo Yenque T <ayenquet@gmail.com>
Date:  Sun May 24 18:37:14 2020 -0500
```

• GIT CHECKOUT [NOMBRE DE LA RAMA]

Comando para movernos de una rama a otra, con git status nos indica y confirma que nos movimos a la nueva rama.

```
prueba git/checkout
> git status
En la rama hotfix1
nada para hacer commit, el árbol de trabajo está limpio
```



git checkout -b "nombre_rama"

Este comando es una fusión entre "git branch" y "git checkout", y crea una rama llamada "nombre_rama" y a la vez hace un checkout de la rama "nombre_rama"

Las ramas en Git son importantes porque te permiten independizar los cambios en un proyecto de tal forma que se pueda realizar avances optimizando el tiempo y el orden, la herramienta es útil porque se pueden fusionar dichos cambios sin perder registro de las versiones anteriores.



git

FUSIÓN DE RAMAS CON GIT MERGE



OJO: Importante antes de hacer checkout a otra rama, hacer el add y commit para no perder los cambios! y el merge siempre ocurre en la rama donde estoy.

Comandos utiles

git branch -v #para ver la última confirmación de cambios en cada rama

git branch --merged #ver las ramas que han sido fusionadas con la rama activa

git branch --no-merged #mostrar todas las ramas que contienen trabajos sin fusionar

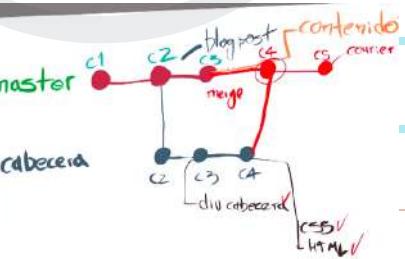
git branch -D #Permite borrar la rama, forzando el borrado incluso si se tiene trabajos sin fusionar. Se pierde el trabajo contenido en ella.

git branch -l

Comando para ver la lista de los branches del proyecto, además que indica la rama actual

```
* Cabecera
  RD
  hotfix1
  master
  nueva-plugin
  (END)
```

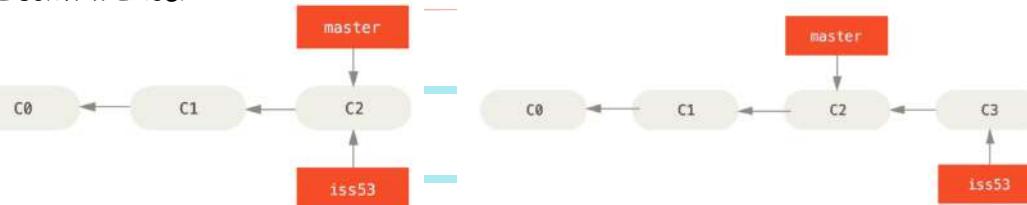
Flujo:



PROCEDIMIENTOS BÁSICOS DE RAMIFICACIÓN Y FUSIÓN

1) Para crear una nueva rama y saltar a ella, en un solo paso, puedes utilizar el comando `git checkout -b "iss53"`.

2) En la nueva rama "iss53", realizamos cambios que son independientes de la rama "master", no olvidar ejecutar los comandos add y commit (`git commit -am "mensaje"`) para confirmarlos.



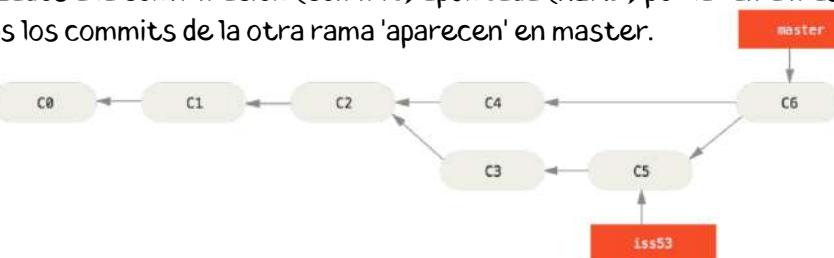
3) Se puede continuar con el trabajo en la rama "master", usando `git checkout master`. Estando en la rama master se realizan cambios y también se confirman (`git commit -am "mensaje"`)

Nota: Al hacer checkout, los archivos vuelven al estado tal como se encontraban en la rama.

4) Luego de confirmar que todo está correcto (`git status` o `git log`) se necesita fusionar ambas ramas

- **git merge "nombre rama"**

Comando que permite fusionar ramas. Lo que hace Git es traer los cambios realizados a la confirmación (commit) apuntada (HEAD) por la rama master, así todos los commits de la otra rama 'aparecen' en master.



```
$ git merge cabecera
Auto-merging css/estilos.css
Auto-merging blogpost.html
Merge made by the 'recursive' strategy.
blogpost.html |  4 ++++
css/estilos.css | 24 ++++++-----+
2 files changed, 27 insertions(+), 1 deletion(-)
```

5) En caso ya no se necesite una rama y no se va a necesitar más, es importante borrar la rama creada.

- **git merge -d "nombre rama"**

Comando que permite eliminar una rama, cuando esta ya no se va a usar, y estamos seguros que el merge fue exitoso y no tiene trabajos sin fusionar.

Para poder unificar los avances de cada rama, existe "Merge". Git fusiona los commits de cada rama y lo muestra en la rama que nos encontramos, asimismo se mantienen independientes para continuar trabajando en cada una hasta un próximo "merge".



git

SOLUCIÓN DE CONFLICTOS AL HACER UN MERGE



Puedo ver qué archivos permanecen sin fusionar en un determinado momento conflictivo de una fusión con `git status`, todo aquello que sea conflictivo y no se haya podido resolver, se marca como `unmerged`.

Comandos útiles

`git mergetool` #Arranca una herramienta visual en consola que permite resolver conflictos.

`git merge --abort`
#Comando para abortar la fusión en progreso actual, en caso no puedo resolver los conflictos en ese momento.

`git reset --merge HEAD`
#Aporte: Si hemos realizado un merge con una rama con la que no queríamos.

Si se ejecuta `add` (luego de realizar la corrección del conflicto) debemos ejecutar `commit`, para terminar de confirmar la fusión.

`git log --graph --decorate --oneline`

```
* 36e63be (HEAD -> master) Solucione el conflicto de las ramas
* 3dfe638 Solucione el conflicto de las ramas al fusionar
* b0ba5c2 Solucione el conflicto de las ramas al fusionar
| 
| + 23db066 (cabecera) Modificando la cabecera y el color de diseño azul
| | d88775a Agregue suscripción, cambie cabecera y color azul
| |
| * 03e0494 Listo para el merge
| 
| * 741a2a2 Finalizada la cabecera con diseño azul
| | 429ca05 Estructura inicial de la cabecera
| | 4f93d0a Agregó el contenido adicional y una mejor tipografía
| 
| * c0f6e44 Commit al master del blogpost en su versión más reciente
| | cb4e395 Se cambió el editor de texto
| | 3c6513b --Agregando a la rama master
| | c1c2876 cambio de maestría
| | f262bb0 cambio de vida
| | 33babc5 arrancó mi proyecto real
| | 64b7389 (tags: v0.1) Este es la primera versión subida
| 
| (END)
```

✓ En algunas ocasiones, los procesos de fusión no suelen ser fluidos. Si hay modificaciones dispares en una misma porción de un mismo archivo en las dos ramas distintas que pretendes fusionar, Git no será capaz de fusionarlas directamente.

```
Proyecto1.git/master
> git merge cabecera
Auto-fusionando css/estilos.css
CONFLICTO (contenido): Conflicto de fusión en css/estilos.css
Auto-fusionando blogpost.html
CONFLICTO (contenido): Conflicto de fusión en blogpost.html
Fusión automática falló; arregle los conflictos y luego realice un commit con el resultado.
```

✓ Git no crea automáticamente una nueva fusión confirmada (merge commit), sino que hace una pausa en el proceso, esperando a que tú resuelvas el conflicto.

```
9          HyperBlog
10         Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
11 <===== HEAD (Current Change)
12           <span id="tagline">Tu blog maestro</span>
13 =====
14           <span id="tagline">Tu blog de confianza</span>
15 >===== cabecera (Incoming Change)
16           </div>
```

Lo que está arriba del `=====` es la versión del HEAD y lo que está debajo, son los cambios de la rama con la quiero hacer el merge.

Para resolver el conflicto, se tiene que elegir manualmente el contenido de uno o de otro lado.

✓ Una vez que se resolvieron los conflictos, se debe agregar los cambios, ejemplo:

```
git commit -am "Solucioné el conflicto de las ramas"
```

✓ Se recomienda agregar al mensaje, detalles sobre cómo has resuelto la fusión, si lo consideras útil para que otros entiendan esta fusión en un futuro.

Se trata de indicar por qué has hecho lo que has hecho; a no ser que resulte obvio, claro está.

```
Education-Platzi@Laptop MINGW64 ~/proyecto1 (master|MERGING)
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:   blogpost.html
    both modified:   css/estilos.css

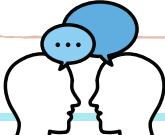
no changes added to commit (use "git add" and/or "git commit -a")
```

```
Education-Platzi@Laptop MINGW64 ~/proyecto1 (master|MERGING)
$ git commit -am "Solucione el conflicto de las ramas al fusionar"
> "
[master fcd7577] Solucione el conflicto de las ramas al fusionar

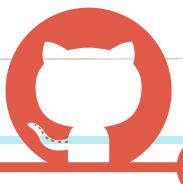
Education-Platzi@Laptop MINGW64 ~/proyecto1 (master)
$ |
```

Los conflictos en archivos son algo normal que sucede en los equipos de trabajo, sobretodo con los cambios de ultima hora. Git permite la solución de los mismos mediante herramientas propias o editores de código como VS Code, la solución se debe escoger manualmente y luego confirmar esa elección para que Git confirme la fusión.

Es importante la comunicación entre las partes que originaron los cambios en conflicto!



USO DE GITHUB



GitHub Inc.

GitHub

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.

• 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.

```
Projecto 85
> 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 refs to 'https://github.com/ayenque/hyperblog.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is caused usually by another repository
hint: updating to the same ref. You might want to integrate the changes
hint: from the remote into your local repository first.
hint: You can force the push with 'git push --force'.
hint: See 'git push --help' for details.
Projecto 105
```

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

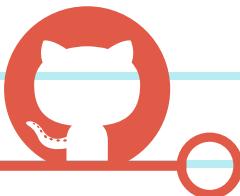
`git push origin master`

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.

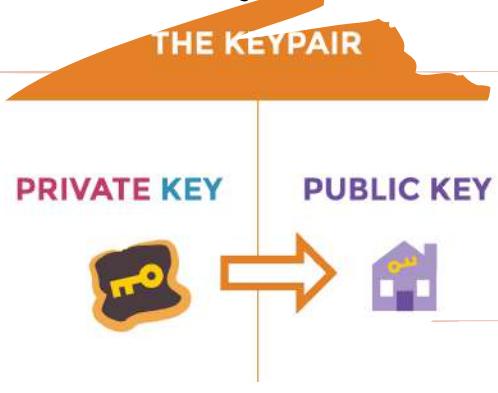
<https://www.s>

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.

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



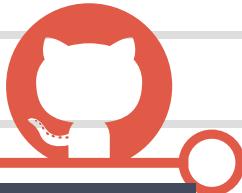
Las llaves públicas y privadas sirven para compartir información en Internet de una forma segura, incluso si el mensaje es interceptado la probabilidad de que se pueda descifrar es casi nulo, si no se tiene la llave privada. Este método se usa incluso en el mundo financiero.

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



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

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

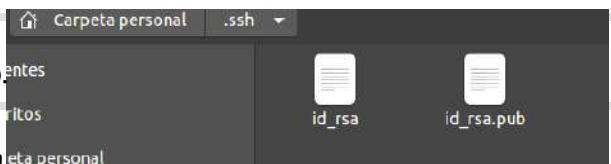
```
ssh-keygen -t rsa -b 4096 -C "ayenquet@gmail.com"
```

```
> 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:MrPCd3Y7xBe1QyX0LcnT3n2 [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
```

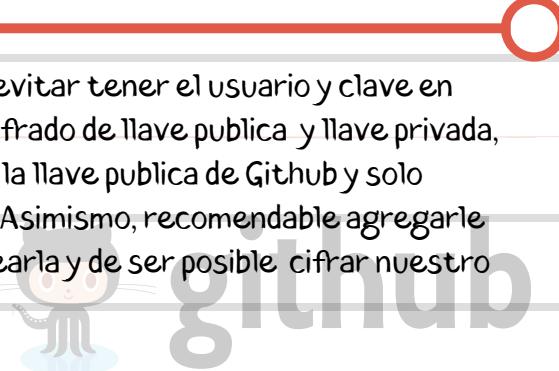
```
> ssh-add ~/.ssh/id_rsa
Enter passphrase for /home/ayenquet/.ssh/id_rsa:
Identity added: /home/ayenquet/.ssh/id_rsa (ayenquet@gmail.com)
```

PARA MAC:

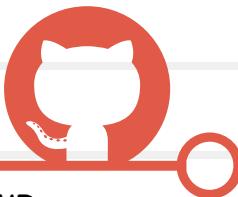
```
# Encender el "servidor" de llaves SSH de tu computadora:
eval "$(ssh-agent -s)"
# Si usas una versión de OSX superior a Mac Sierra (v10.12)
# debes crear o modificar un archivo "config" en la carpeta
# de tu usuario con el siguiente contenido (ten cuidado con
# las mayúsculas):
Host *
    AddKeysToAgent 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



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.

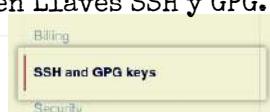
Mensaje que asegura que la IP de Github es reconocida como de confianza, no es mensaje de error, solo de advertencia.

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



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`).

• 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.

```
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 SHA256:nThbg6KXWgRZuI1E1zTW5ADqkm6Vg1E1o.
Are you sure you want to continue connecting (yes/no)? y
Please enter your passphrase: 
Warning: Permanently added 'github.com,140.82.113.3' (RSA) to the list of known hosts.
No hay información de rastreo para la rama actual.
Por favor especifica a qué ramaquieres 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=<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.
Resolving deltas: 100% (3/3), done.
git pull origin 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.

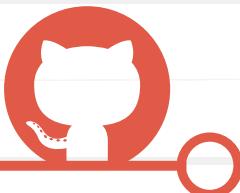
```
Proyecto git/master
> git remote -v
origin git@github.com:ayenque/hyperblog.git (fetch)
origin git@github.com:ayenque/hyperblog.git (push)
```

- 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 -m "Una versión del HyperBlog"
[master c38368c] 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.
Dese git pull origin master
> branch master -> FETCH_HEAD
Ya está actualizado.

Proyecto git/master
> git push origin master
Enumerando objetos: 5, listo.
Contando objetos: 5, listo.
Comparando deltas usando hasta 8 bits.
Comprimiendo objetos: 100% (3/3), listo.
Escribiendo objetos: 100% (3/3), 315 bytes | 355.00 KiB/s, 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
 771e3b...c3010dc master -> master
Proyecto git/master
```

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 practica.

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.

- **git show-ref --tags**

Comando para revisar la lista de tags creados y sus referencias

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

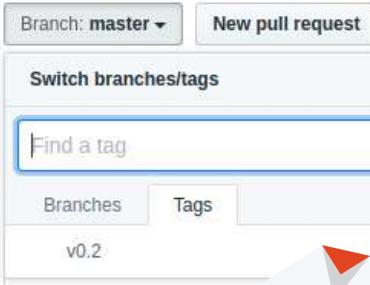
```
Proyecto1 git/master
> git tag
```

```
Proyecto1 git/master
> git show-ref --tags
64b738967ac357510d6d28aa043dd7de71 refs/tags/0.1
b1d57aaaf6ec7ef5cd4405d842f19eb53e61d8101 refs/tags/0.5
2c478a86d239dbdb5a058f38dd30ed10d43684f6 refs/tags/v0.2
```

- **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, lista.
Contando objetos: 100% (7/7), lista.
Comprimiendo delta usando hasta 8 hilos
Comprimiendo objetos: 100% (5/5), lista.
Escribiendo objetos: 100% (5/5), 645 bytes | 645.00 KiB/s, lista.
Total 5 (delta 2), reusados 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

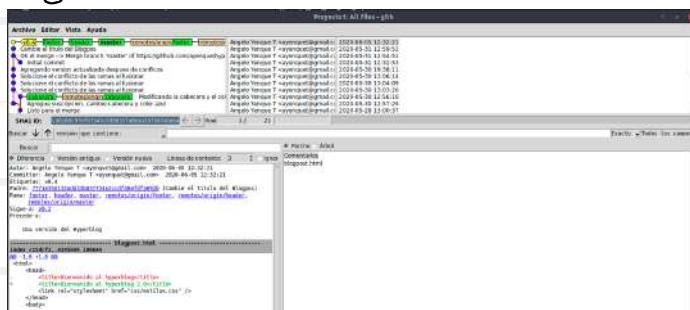
Comando para revisar las ramas creadas, pero si queremos ver más detalles podemos usar los siguientes comandos.

• 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.

```
Proyecto git/cabecera  
> git show-branch  
[cabecera] Modificando la cabecera y el color del texto  
! [master] Una versión del Hyperblog  
+ [master*] Cambio el título del Blogos  
+ [master-2*] Initial commit  
+ [master-3*] Agregando una versión actualizada después de conflictos  
+ [master-4*] Solucionando conflicto de las ramas al fusionar  
+ [master-5*] Solucionando el conflicto de las ramas al fusionar  
+ [cabecera] Modificando la cabecera y el color del texto  
  
Proyecto git/cabecera  
> git show-branch  
[cabecera] Modificando la cabecera y el color del texto  
! [master] Una versión del Hyperblog  
+ [master*] Cambio el título del Blogos  
+ [master-2*] OK el merge -> Merge branch 'master' of https://github.com/ayenque/hyperblog  
+ [master-3*] Initial commit  
+ [master-4*] Agregando una versión actualizada después de conflictos  
+ [master-5*] Solucionando el conflicto de las ramas al fusionar
```

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.



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.

Repository > 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

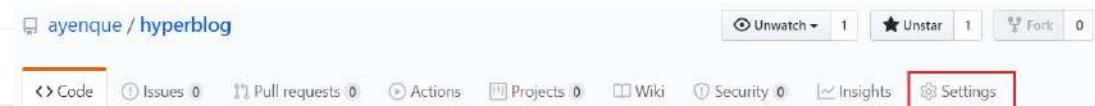
Aporte
tucorreo+algo@gmail.com
tucorreo+algo@hotmail.com

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**

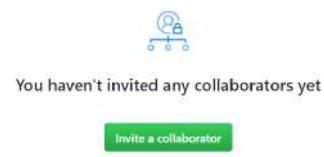
2. Debajo de tu nombre de repositorio, da clic en **Settings**.



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



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



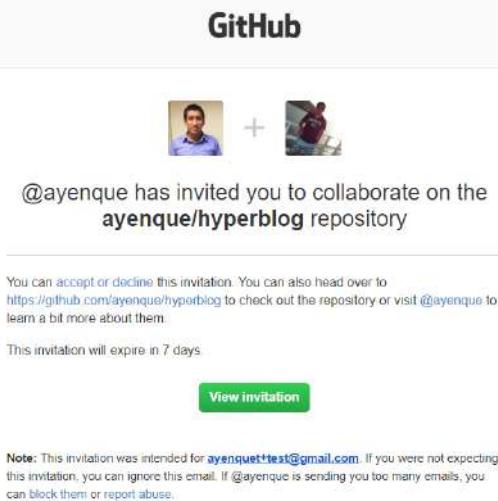
5. Comienza a teclear el nombre de la persona que deseas invitar dentro del campo de búsqueda. Posteriormente, da clic en algún nombre de la lista de coincidencias. Despues click en "**Add [user] to [repository]**"

El usuario recibirá un correo electrónico invitándolo al repositorio.

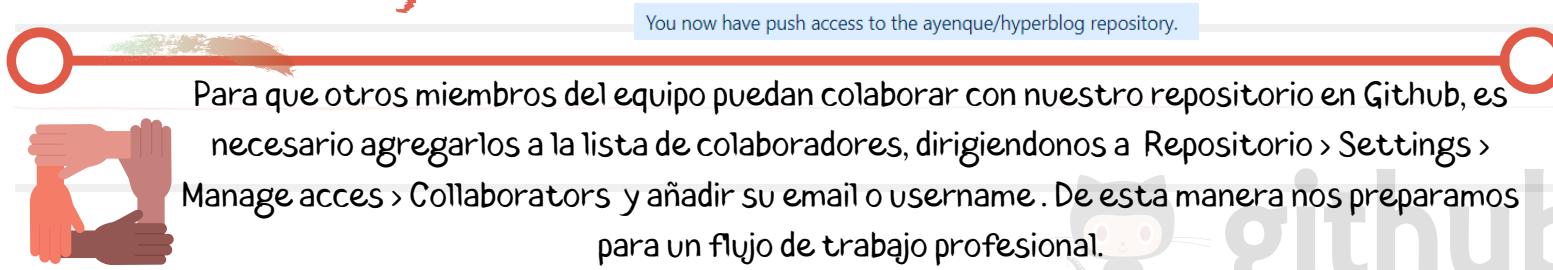
6. Click en **View invitation** y luego de ingresar a su cuenta **Accept invitation**.

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 Repository > Settings > Manage acces > Collaborators y añadir su email o username. De esta manera nos preparamos para un flujo de trabajo profesional.



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 traemos 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
```



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:

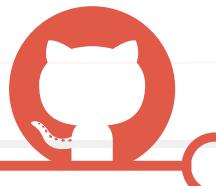
```
git merge footer
```

Finalmente envío la rama master actualizada a Github

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

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 cual es el procedimiento para tener éxito en la fusión de todos los avances y/o updates del proyecto.

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.

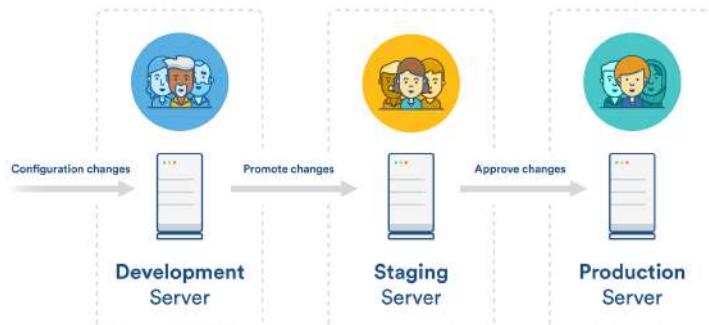
Los **pull request** tambien son importantes porque permiten a personas que no son colaboradores, trabajar y apoyar en nuestro proyecto.

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.

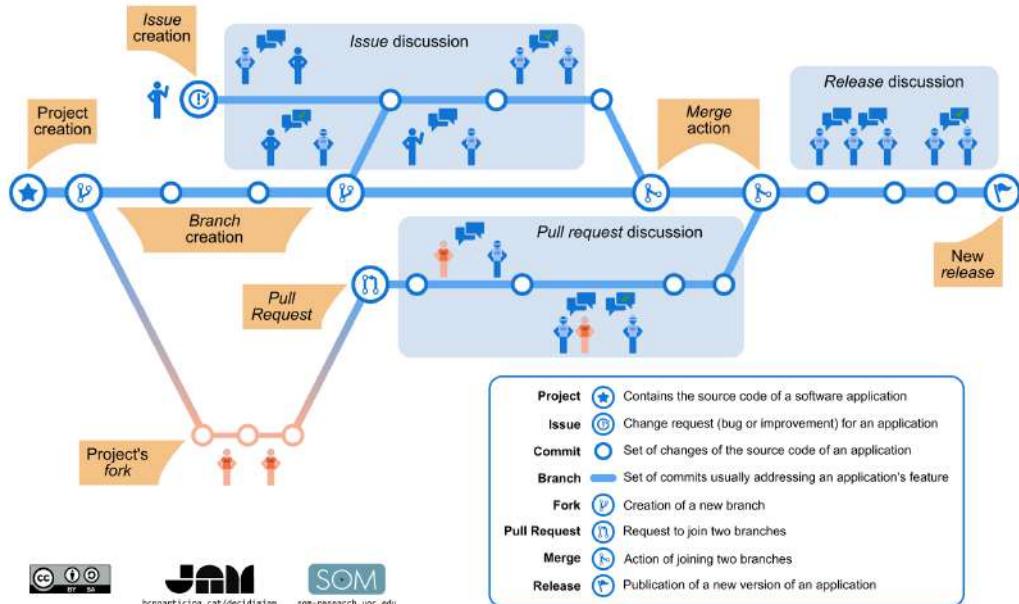
Para realizar pruebas enviamos el código a servidores que normalmente los llamamos **staging server**, luego de que se realizan las pruebas pertinentes tanto de código como de la aplicación estos pasan a el **servidor de producción**.



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?

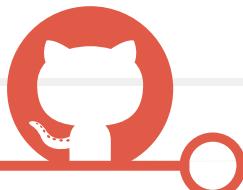


La persona que hace todo esto, normalmente son los líderes de equipo o un perfil muy especial que se llama **DevOps** (permite que los roles que antes estaban aislados se coordinen y colaboren para producir productos mejores y más confiables).

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 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

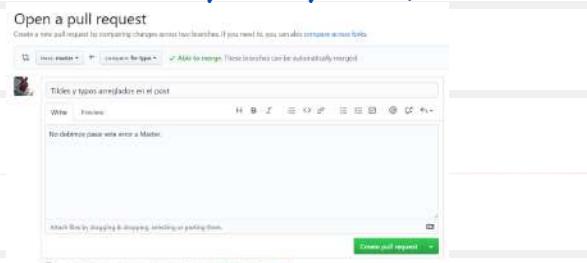
Una vez realizado el Merge, es importante traer los cambios en Git (Git Pull), en caso borramos la rama en Github, podemos borrarla en Git con `git branch -D nombre-rama`

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-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

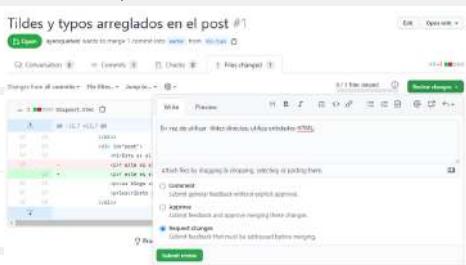
Tildes y tipos arreglados en el post

ayenque/hyperblog · Your review was requested 3 days ago

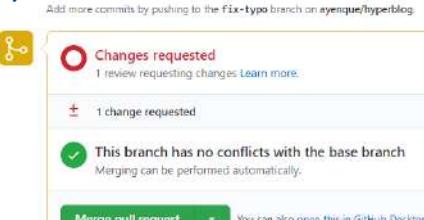
d) El administrador puede comparar los cambios haciendo click en **Files changed**, y consultar con el equipo para la revisión respectiva.



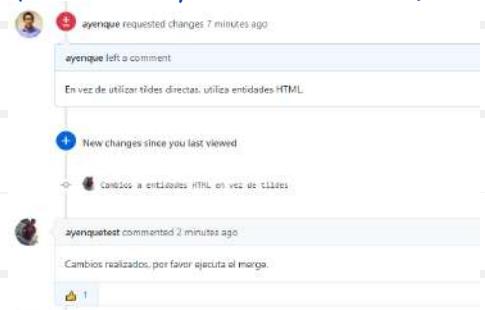
e) Dar click en **Review Changes**, en caso de requerir cambios al solicitante.



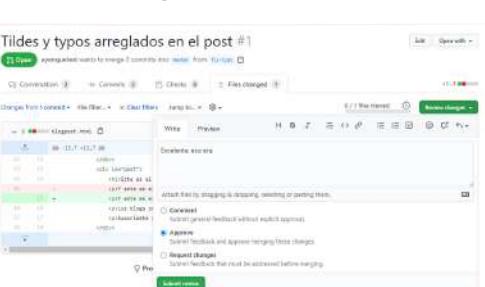
f) Al solicitante, se le notifica del cambio requerido ingresando en **Pull Requests**



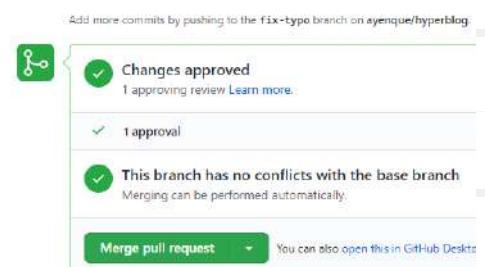
g) Una vez realizados los cambios requeridos responder el **Pull Request**



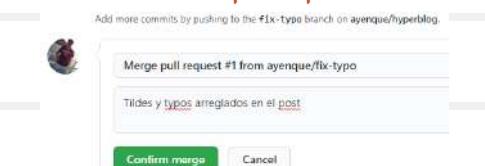
h) Podemos aprobar los cambios, en **Review Changes y Approve**



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



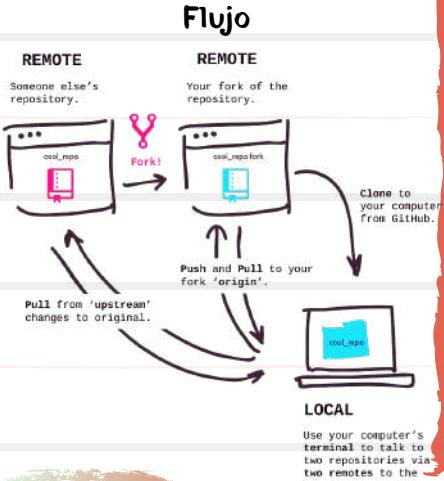
En un proyecto público podemos dar click en **Watching** y **Star** para seguir el repositorio, de esta manera apoyamos y recibimos avisos cada vez que hay cambios en dicho repositorio.

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**)

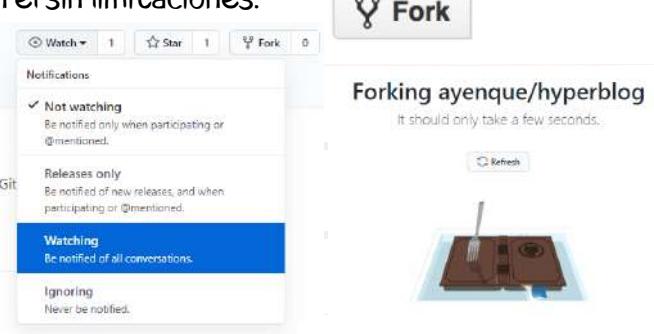
ayenquetest / hyperblog
forked from ayenque/hyperblog

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.



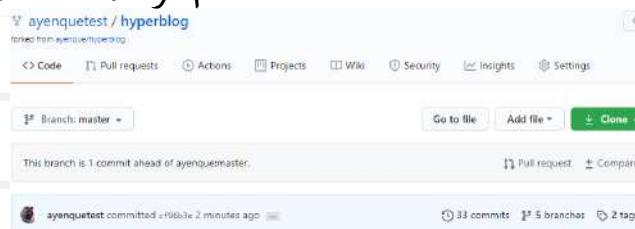
Siquieres participar en un proyecto existente, en el que no tengas permisos de escritura, puedes bifurcarlo (hacer un "fork"). Esto consiste en crear una copia completa del repositorio totalmente bajo tu control: se almacenará en tu cuenta y podrás escribir en él sin limitaciones.

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

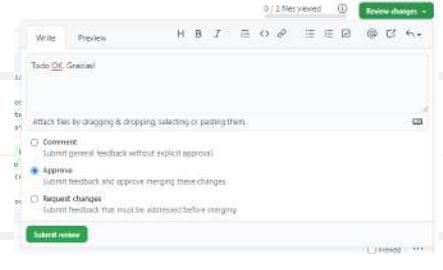
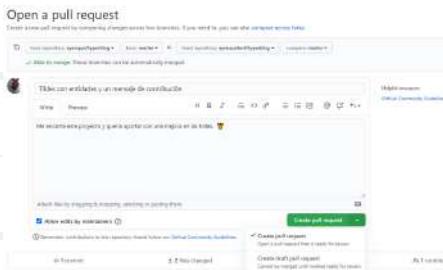


En unos segundos te redireccionarán a una página nueva de proyecto, en tu cuenta y con tu propia copia del código fuente.

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:

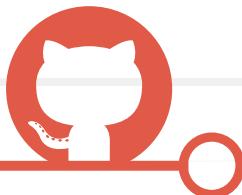
`git pull upstream master # traemos cambios del repositorio original`

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



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

Node OSX Polymer SublimeText 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>



ghost

Fast, Clean, Simple. Static.

TryGhost/Ghost

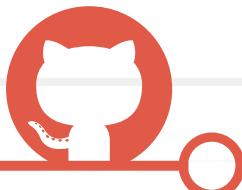
The #1 headless Node.js CMS for professional publishing - TryGhost/Ghost

[GitHub](#)

<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!



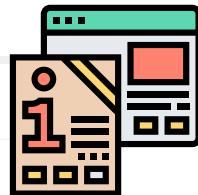
README brinda una guía rápida a los que acaban de descubrir nuestro proyecto de cómo empezar a usarlo. Es muy importante que vaya al grano, sea conciso y muy claro. Para extendernos y entrar en detalles está la documentación, a la que siempre podemos enlazar desde dentro de nuestro README.

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](#)

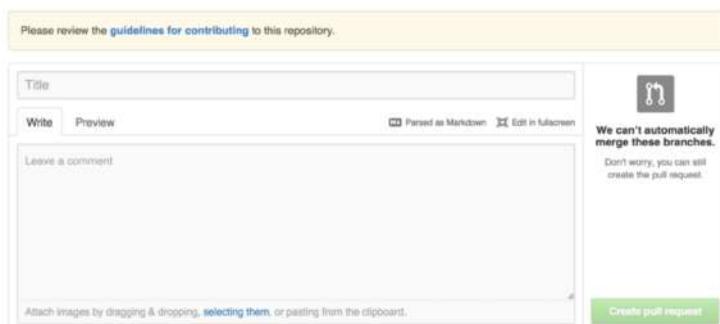
<https://github.com/laravel/laravel/blob/master/README.md>



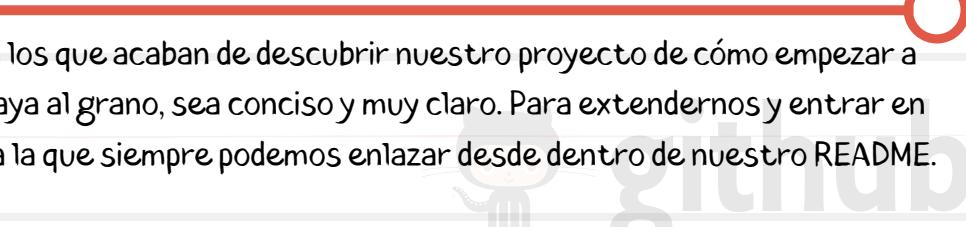
<https://github.com/TryGhost/Ghost/blob/master/README.md>

CONTRIBUTING

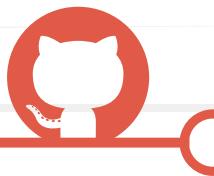
El otro archivo que GitHub reconoce es CONTRIBUTING. Si tienes un archivo con ese nombre y cualquier extensión, GitHub mostrará algo como la imagen, cuando se intente abrir un Pull Request.



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**.



<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



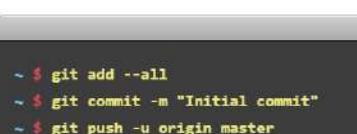
Clonar el repositorio



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



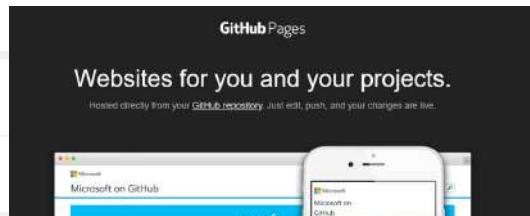
Ejecutar los comando git add, git commit, y git push para aplicar los cambios:



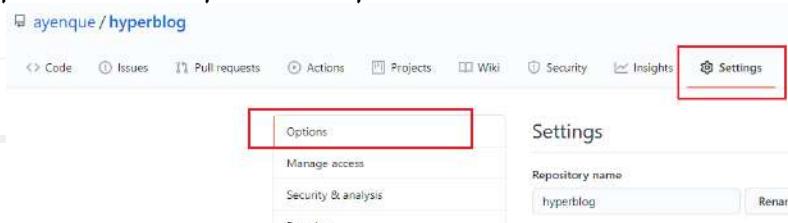
GITHUB PAGES

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

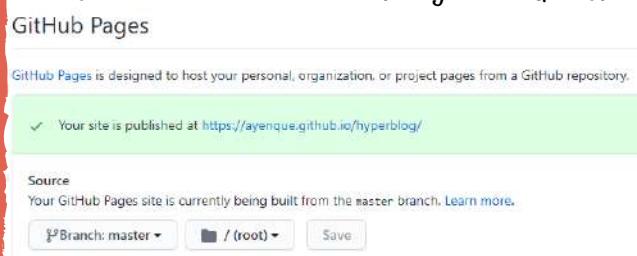
Abrir la página oficial de Github para más ayuda: <https://pages.github.com/>



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".



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



hyperblog

Hyperblog ❤️

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

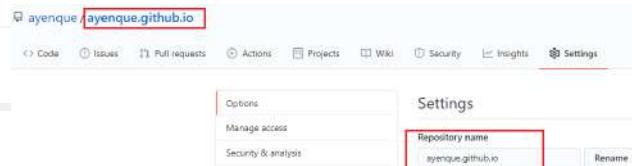
El curso de Git y Github de Platzi es lo que me ha dado más

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

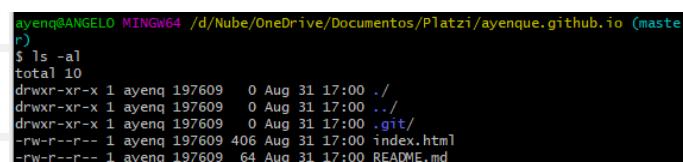
GITHUB PAGES PERSONAL

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



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

No olvidar agregar el archivo "index.html"



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

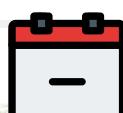
Rebase es solo para repositorio locales, porque reescribe la historia del repositorio.

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:

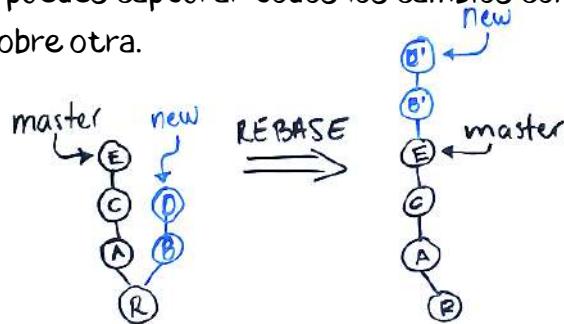
- No queda historia de los cambios originales
- No se sabe el autor real de los commits.
- Si la rama principal avanzó varios commits, puede generar varios conflictos que se tienen que arreglar de forma manual.



```
* 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 (oríz)
* 5495686 - (hace 7 días) Referenciando una imagen desde un servidor externo
```

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.

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 8a2e9d3] 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 8a2e9d3] 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.

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

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

```
Proyecto1 git/experimento 22s
> 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
Retrocediendo 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
Retrocediendo 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(+)
```

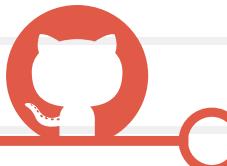
- Regresamos a la rama **Experimento** y ejecutamos **git rebase master**
- Regresamos a la rama **Master** y ejecutamos **git rebase experimento**

```
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.
```

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



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
```

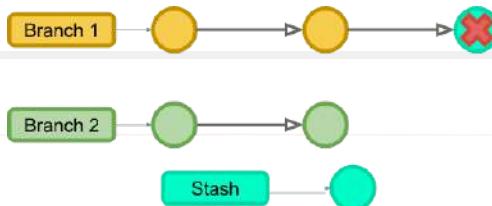
```
git stash show  
#Comando para visualizar un resumen de un stash
```

```
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.

```
Proyecto 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.

```
Proyecto 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 <archivo>..." 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

```
Proyecto 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 <archivo>..." 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} (2a09318919d16f73e08c89df63ea27f5@bc93f5@)

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**.

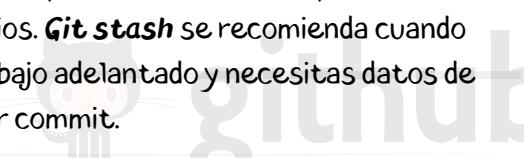
```
Proyecto git/master  
> git stash drop  
Botado refs/stash@{0} (ce8c16a81a67d846d6741d9760fe622ccffc6164)
```

• git stash clear

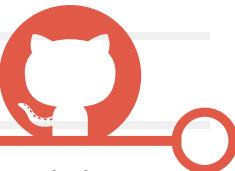
Comando para eliminar todos los stashes.

```
Proyecto 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          css/estilosv0.css  historiav0.txt  
*** 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 (mas  
ter)  
$ 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/Documen  
$ git config --global -l  
user.name=Angelo Yenque T.  
user.email=ayennouet@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/  
ter)  
$ 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/OneDriv  
ter)  
$ git clean -dn  
Would remove Error/  
Would remove README2.md  
Would remove historia2.txt
```

```
ayenq@ANGELO MINGW64 /d/Nube/OneDriv  
ter)  
$ git clean -df  
Removing Error/  
Removing README2.md  
Removing historia2.txt
```

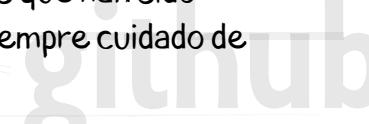
• git clean -x

`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.

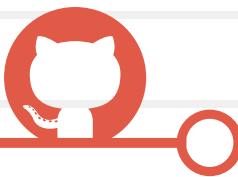
```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Do  
$ git clean -xdn  
Would remove Error/  
Would remove blogpostv0.html  
Would remove historiav2.txt  
Would remove imagenes/dragon - Copy.png
```

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/  
$ 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`

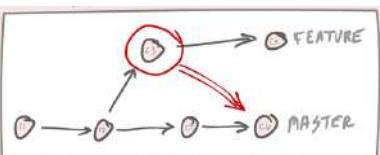
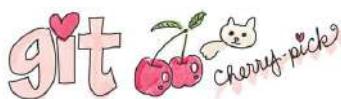
```
ayenq@ANGELO MINGW64 /  
mejorado)  
$ git log --oneline  
+d03706 [HEAD -> readme  
f854a03 Ejemplos en Wi  
a9615ca (origin/master  
6c79b73 Cambio al tag!  
64e0db Actualizando r
```

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
```

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/P  
mejorado)  
$ 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]
```

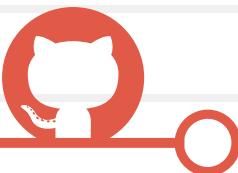
Podemos usar la opción `-x` en caso deseamos añadir una referencia al commit original

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

```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/P  
$ 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/P  
$ 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`

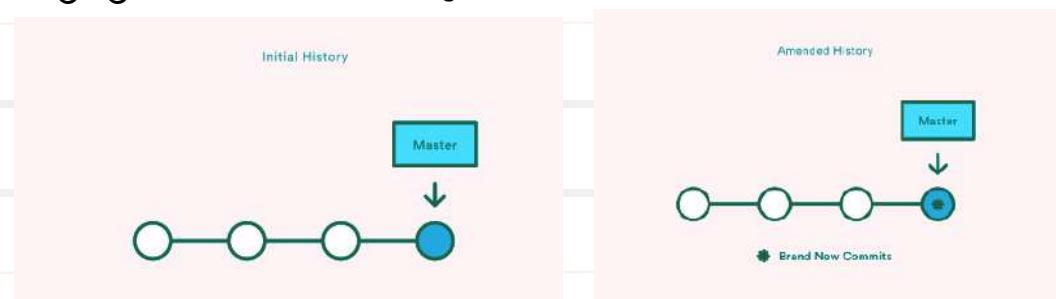
`git commit --amend --no-edit`

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 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.
```

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 log --oneline
a2d27bf (HEAD -> master) 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
8faa220 (HEAD -> master) Personalizando archivo historia.txt
 1 file changed, 1 insertion(+), 1 deletion(-)
```

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`.

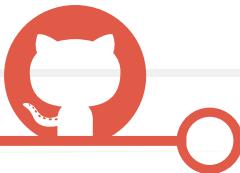
```
ayenq@ANGELO MINGW64 /d/Nube/OneDrive/Documentos/Platzi/Proyecto1/hyperblog (master)
$ git push origin master
Enter passphrase for key '/c/Users/ayenq/.ssh/id_rsa':
To github.com:ayenque/hyperblog.git
 ! [rejected]          master -> master (non-fast-forward)
error: failed to push some refs to 'github.com:ayenque/hyperblog.git'
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 REFLог: 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 REFLог

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.

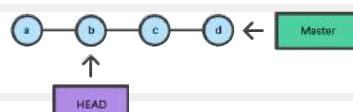
• `git reflog stash`

Comando para ver el registro de referencia de un stash.

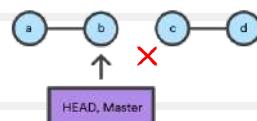
• GIT RESET

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.



- 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.

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:  
blogpost.html:
```

<p>Y este es el párrafo de inicio donde vamos a explicar las cosas increíbles
<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 -S "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
```

