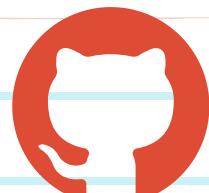


¿QUÉ ES GIT?



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

Sistema operativo: Unix-like, Windows, Linux
Programado en: C, Bourne Shell, Perl
Modelo de desarrollo: Software libre
Escrito en: C, Perl, Tcl, Python

Importante

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

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

GIT

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



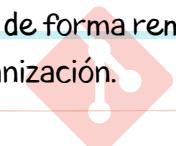
github

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

Características de Github

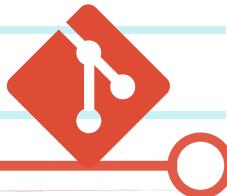
- GitHub permite que alojemos proyectos en repositorios de forma gratuita y pública, pero tiene una forma de pago para privados.
- Puedes compartir tus proyectos de una forma mucho más fácil.
- Te permite colaborar para mejorar los proyectos de otros y a otros mejorar o aportar a los tuyos.
- Ayuda a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

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



git

¿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



monotone

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**

Git



Mi nombre es Freddy Vega

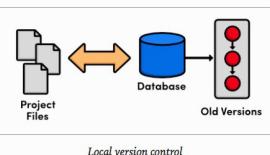
Soy el CEO de Platzi y en mis tiempos libres **corro carreras**, escribo libros y hago cosas raras con **Arduino**

biografia.txt

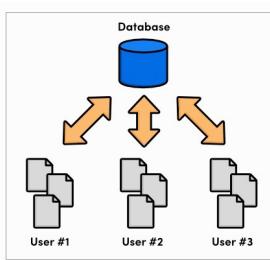
- T > E
- juego tennis > corro carreras
- Raspberry Pis > Arduino

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

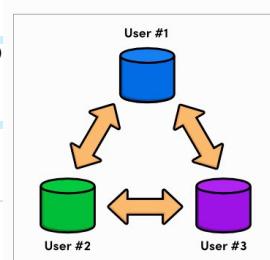
- Sistemas de Control de Versiones Locales, almacenan los cambios en una base de datos y solo se lleva en el computador de cada uno de los desarrolladores por separado.
- Centralizados, no dependen de nuestro dispositivo de computo, sino que los cambios están guardados en un servidor, todos pueden tener 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



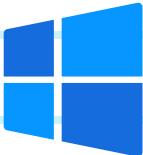
Centralized version control



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 GITBASH 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>

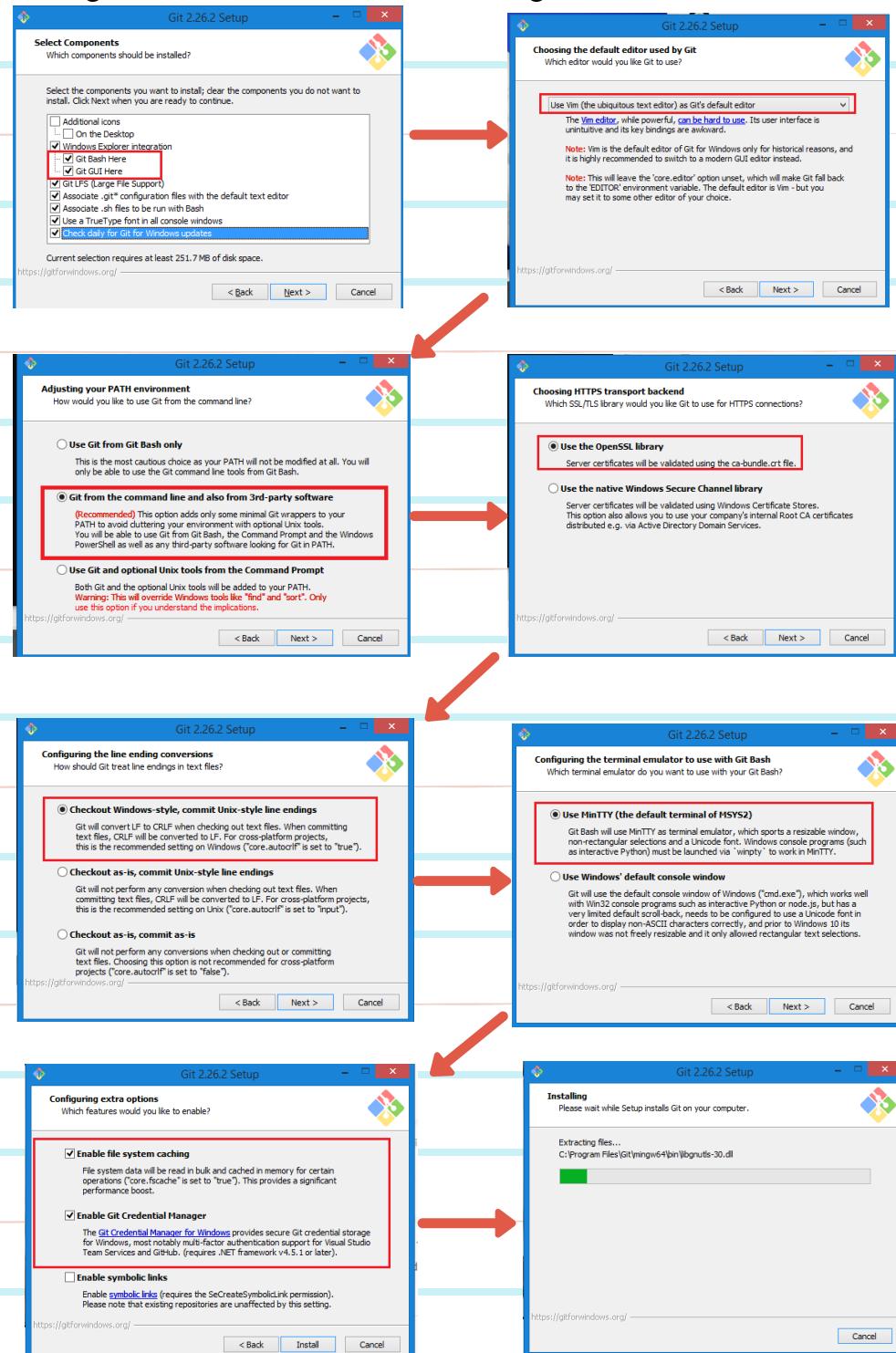
Una vez culminada la instalación, para verificar que GIT esta instalado, abrimos en **"GIT BASH"** y escribimos el comando **git --version** para corroborar la versión instalada.

```
git@PauloBalan: MINGW64 ~
usage: git [<version>] [<--help>] [<--spaths>] [<--name=><value>]
        [<--exec-path=><path>] [<--html-path>] [<--man-path>] [<--info-path>
        [<p> | <--paginate> | <--no-replace-objects>] [<--bare>]
        [<--git-dir=><path>] [<--work-tree=><path>] [<--namespace=><name>]
        [<command> | <git>]

these are common Git commands used in various situations:
init          - Create a new Git repository or reinitialize an existing one
add           - Add file contents to the index
commit       - Record changes to the repository
diff          - Show differences between the current commit and working tree, etc.
grep          - Print lines matching a pattern
log           - Show commit logs
show          - Show details of objects
status        - Show the working tree status
branch       - List, create, or delete branches
commit       - Record changes to the repository
merge        - Merge multiple histories together
rebase       - Reapply commits on top of another base tip
reset        - Reset current HEAD to the specified state
stash        - Save and restore work in progress
tag          - Create, list, delete or verify a tag object signed with GPG
pull         - Download objects and refs from another repository
push         - Fetch from and integrate with another repository or a local branch
fetch        - Update remote refs along with associated objects
git help -c  -> git help workflow
git help -c  -> git help <command> or <git help <concept>>
git help -c  -> read about a specific subcommand or concept.
git help git  -> for an overview of the system.

git@PauloBalan: MINGW64 ~
git --version
git version 2.26.2.windows.1
git@PauloBalan: MINGW64 ~
```

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

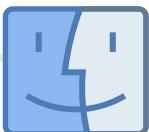


END

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



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

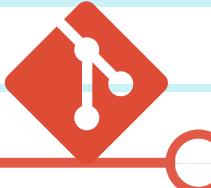
- Si deseas una versión más actualizada, puedes hacerlo a partir de un instalador binario. Un instalador de Git para OSX es mantenido en la página web de Git. Lo puedes descargar en <http://gitscm.com/download/mac>



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

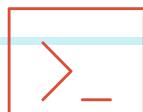
sudo apt-get update
y
sudo apt-get upgrade

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>)

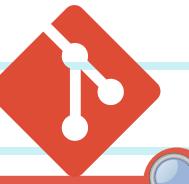


Los usuarios de Linux pueden administrar Git principalmente desde la línea de comandos. Se recomienda usar Linux debido a su alto rendimiento, a que es Open Source, a su poderosa Terminal, para gestión de Servidores, desarrollo Web, Bash, etc.



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

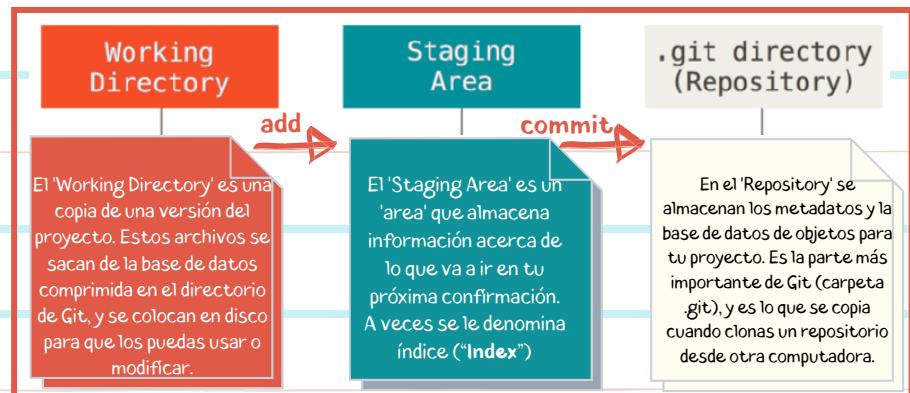
git commit -am "mensaje del commit" #Fusión entre git add y git commit -m, agrega los archivos y ejecuta un commit a la vez. Funciona solo para archivos que previamente fueron agregados (tracked).

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:



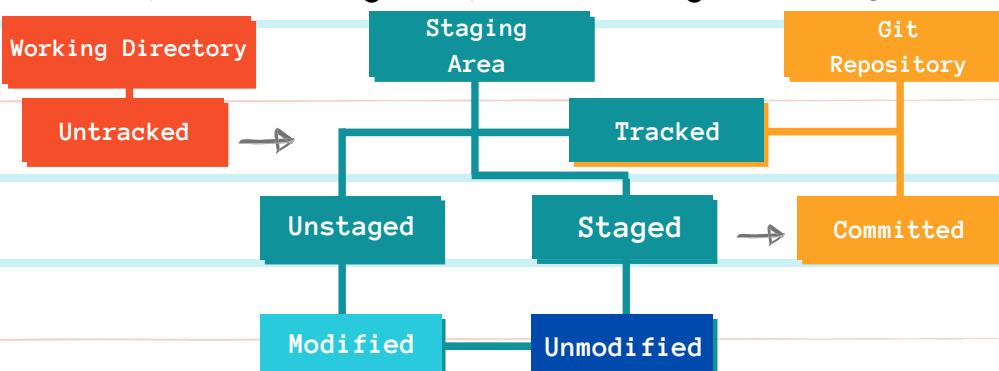
El flujo de trabajo básico en Git es algo así:

- 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 sobretodo 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]

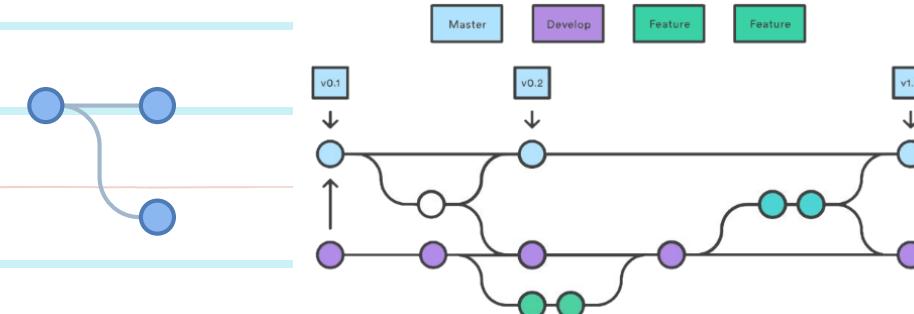
How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? [] v
Hooks and filters directory? [D:/Nube/OneDrive/Dокументos/blog/.git/hooks]
```

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.

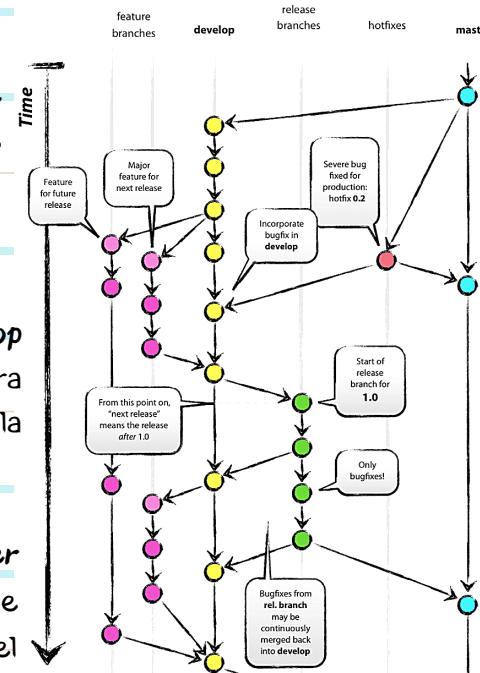


• RAMA DEVELOP

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.

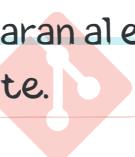
```
$ git add historia.txt  
warning: LF will be replaced by CRLF in historia.txt.  
The file will have its original line endings in your working directory
```

• 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

Codigos



```
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 codigos:

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

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



• Git show

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

```
ayenquet@ayenque:~/Documentos/Platzl/Git_Github/Proyecto1$ git show historia.txt  
commit c497331c19178ea6432f3117bbd345091382a2f (HEAD -> master)  
Author: Angelo Yenque T <ayenquet@gmail.com>  
Date: Thu May 21 18:16:00 2020 -0500  
  
Este sería el segundo el primer se borro #mensaje del commit  
diff --git a/historia.txt b/historia.txt #diferencias entre commit  
index fc015a2..0ad0904 100644  
--- a/historia.txt # versiones esta 1  
+++ b/historia.txt # esta 2  
@@ -3..4 +3..5 @@ Esta es la historia de Angelo Yenque # inicio del archivo  
Angelo tiene 32 años y nació en Talara - Piura.  
Quiere ser músico!!  
  
Hoy hablaremos de su historia .. se quite linea
```

```
ayenquet@ayenque:~/Documentos/Platzl/Git_Github/Proyecto1$ git show historia.txt  
Archivo Editar Ver Buscar Terminal Ayuda  
commit c497331c19178ea6432f3117bbd345091382a2f (HEAD -> master)  
Author: Angelo Yenque T <ayenquet@gmail.com>  
Date: Thu May 21 18:29:00 2020 -0500  
  
Agregamos cambio en la estatura  
La persona es Angelo  
  
diff -a historia.txt b/historia.txt  
index 6440094..56ef5bf 100644  
--- a/historia.txt  
+++ b/historia.txt  
@@ -3..4 +3..5 @@ Esta es la historia de Angelo Yenque  
Angelo tiene 32 años y nació en Talara - Piura.  
Quiere ser músico!!  
  
+En cámara parece alto, pero en realidad no lo es.  
+ 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 + SCHFT + ZZ (Fuerza y guarda el envio 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]
```

```
ayenquet@ayenque:~/Documentos/Platzl/Git_Github/Proyecto1$ git diff 64b738967ac357510d6d28aa122baa043dd7de71 e72477fcff9c6d161ca584034826a031a28b7791  
diff --git a/historia.txt b/historia.txt  
index f5084f9..0b598e2 100644  
--- a/historia.txt  
+++ b/historia.txt  
@@ -1..6 +1..12 @@  
Esta es la historia de Angelo Yenque  
  
Angelo tiene 33 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!  
  
+Hoy hablaremos de su historia.
```



Se vieron los principales comandos para analizar los cambios en los archivos: Git Show, para ver el ultimo 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

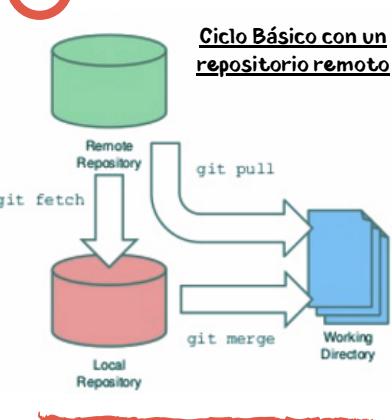
Permite ver a la versión master del archivo, es decir la ultima versión que se había enviado en el ultimo commit.

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

```
[ayenquet@ayenque] - [~/Documentos/Platzi/Git_Github/Proyecto1] - [122]  
[~]$ git checkout master historia.txt  
Actualizada 1 ruta para b1a55a  
[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.

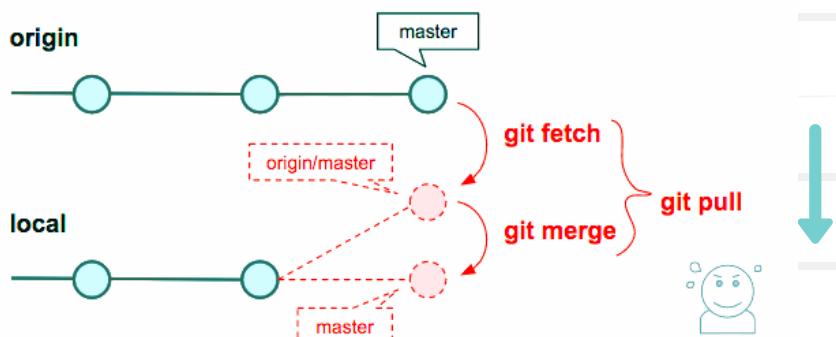
git pull nombre-remoto nombre-rama
#Trae los cambios del repositorio remoto llamado "nombre-remoto" y los fusiona con la rama "nombre-rama"

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

GitHub Bitbucket

GitLab

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.



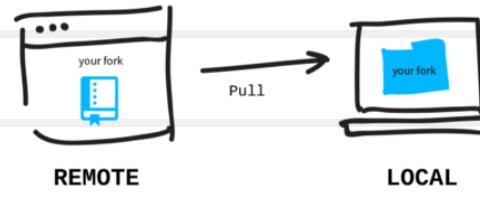
• git merge

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

• git pull

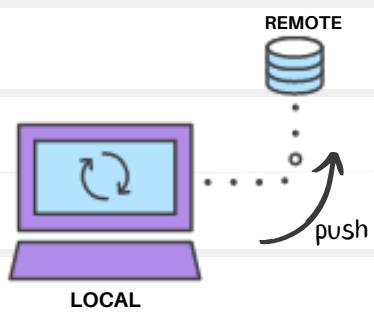
Este comando se emplea para extraer y descargar contenido desde un repositorio remoto y actualizar al instante el proyecto local para reflejar ese contenido.

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/hotfix1
> 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

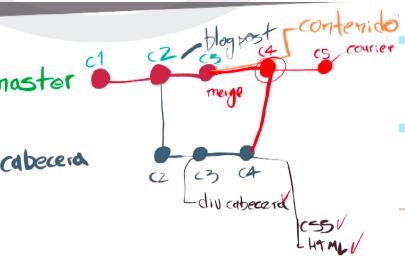
git branch -D #Permite borrar la rama, forzando el borrado incluso si se tiene trabajos sin fusionar. Se pierde el trabajo contenido en ella.

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:



```
$ 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`

```
* 36e3be (HEAD -> master) Solucione el conflicto de las ramas
* 3dfe630 Solucione el conflicto de las ramas al fusionar
* bdbba5c2 Solucione el conflicto de las ramas al fusionar
| *
| * 23d9066 (cabecera) Modificando la cabecera y el color de la cabecera
| * d08775a Agregue suscripción, cambie cabecera y color azul
| *
| * 03e0494 Listo para el merge
| *
| * 741a2a2 Finalizada la cabecera con diseño azul
| * 429caa5 Estructura inicial de la cabecera
| * 4f93da0 Agregado el contenido adicional y una mejor tipografía
| *
| * caf6e44 Commit al master del blogpost en su versión más reciente
| * c84e395 Se cambió el editor de texto
| * 3c6513b - Agregando a la rama master
| * c1c2876 cambio de maestría
| * f262b6f cambio de vida
| * 33babca arrancó mi proyecto real
| * 64b7389 (tag: 0.1) 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.

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

Lo que está arriba del ===== es la versión del HEAD y lo que está debajo, son los cambios de la rama con la que 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

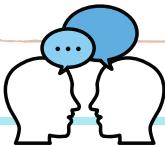
no changes added to commit (use "git add" and/or "git commit -a")
```

```
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!



git