

Introducción

a



git



fb.me/cshluesocc



youtube.com/CSLUESOCC



[@cshluesocc](https://twitter.com/cshluesocc)



github.com/cshluesocc



cshluesocc.org



LinuxGamerSV



Comunidad de Software y Hardware Libre

Table of Contents

Introduction	1.1
Licencia	1.2
El problema	1.3
La solución	1.4
Introducción	1.5
Instalación	1.6
Configuración inicial	1.7
Git básico	1.8
Ignorando archivos	1.8.1
Revirtiendo cambios	1.8.2
Remover y mover archivos	1.8.3
Tagging	1.8.4
Ramificaciones	1.9
Repositorios remotos	1.10
Creando un repositorio en github	1.10.1
Clonando repositorios	1.10.2
Enviando cambios al repositorio remoto	1.10.3
Obteniendo cambios del repositorio remoto	1.10.4

Comunidad de Software y Hardware Libre

DOI 10.5281/zenodo.13950



Introducción a git

El presente documento es una guía de introducción a git. Se comparte de forma publica con el objetivo de extender su alcance a toda persona que desee colaborar ya sea corrigiendo, agregando nuevo contenido o bien compartiendo el presente con sus amigos, comunidades, etc.

Desde la Comunidad de Software y Hardware Libre de la Universidad de El Salvador FMOcc, creemos que el conocimiento debe estar al alcance de todos y todas y no debe ser de ninguna forma limitado a pocas personas.

No hay aporte que no valga, todo aporte, por pequeño que sea es bienvenido en la comunidad.

Puedes contribuir con la comunidad ya sea compartiendo código, material didáctico, difundiendo el Software y hardware libre, difundiendo la comunidad, etc.

Si deseas colaborar has un fork del proyecto en <https://github.com/csluesocc/introduccionGit>, edita, agrega nuevo contenido y has un pull request a este proyecto.

Carlos Cárcamo, 2015

Queda a disposición de toda persona interesada en expandir el conocimiento con sus iguales. Copiar, editar y distribuir esta permitido, siempre y cuando se respete los términos de licencia abajo mencionados.

Quien desee contribuir a este documento, esta en la libertad de hacerlo, por lo cual este documento sera publicado en un repositorio público en github en el siguiente enlace: <https://github.com/csluesocc/introduccionGit>, desde donde podrá ser editado y posteriormente publicado. En el momento que desees colaborar, por favor has un fork del repositorio, has las respectivas observaciones y/o ediciones al documento y agrega tu nombre a la lista de colaboradores a continuación:

COLABORADORES:

- [AgusRumayor](#)



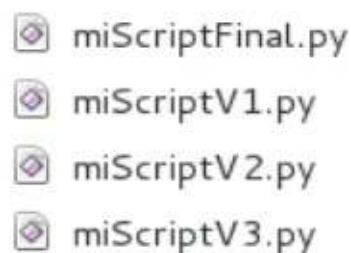
Esta obra está bajo una [Licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional](#). Puede copiar, distribuir y realizar obras derivadas del texto original siempre y cuando se haga referencia al autor y no se persiga una finalidad comercial.

EL PROBLEMA

SISTEMAS DE CONTROL DE VERSIONES

Los sistemas de control de versiones son aquellos que se encargan de registrar los cambios realizados sobre un archivo o un conjunto de archivos a lo largo del tiempo, con el objetivo de mantener un control sistemático permitiendo así en un futuro recuperar versiones específicas del o los archivos modificados.

Es muy común encontrarse con la situación donde se crea un archivo y luego se modifica perdiendo los cambios hechos antes de la ultima actualización; hay una forma de evitar el problema anterior y es haciendo una copia local, renombrarla y hacer los cambios en el nuevo archivo manteniendo la integridad del archivo original, pero, ¿que pasa si se han hecho 5 cambios bastante significativos al archivo? Habría que crear 5 copias diferentes ¿y si trabajo en equipo y cada persona modifica una parte del archivo?, esta situación se vuelve complicada y poco eficiente, la imagen a continuación muestra esta situación:



¿Y si no solo es un simple archivo sino un proyecto grande con diferentes módulos y varias personas trabajando sobre este?



El problema de tener este tipo de control, poco eficiente, radica en que puede causar serios conflictos debido a que cada usuario debe entregar sus modificaciones a los demás y viceversa y cada uno de estos debe ordenar sus archivos, teniendo cuidado de no sobrescribir algún archivo o directorio, y ¿que pasa si por error mi disco duro colapsa y no tuve tiempo de entregar mis últimas modificaciones?, tengo que volver a hacer los cambios! o si la persona que lleva control del sistema pierde la información... que triste situación, cuantas copias deberé crear, cuanto tiempo invertiré copiando, pegando, renombrando, etc.

LA SOLUCIÓN

LOS SISTEMAS DE CONTROL DE VERSIONES AL RESCATE

La penosa situación anterior llevo a la necesidad de crear un sistema capaz de realizar las tareas anteriores de forma ordenada, concisa y sobre todo eficiente, pudiendo centralizar el trabajo en un lugar específico, un servidor, a este tipo de sistemas se les llama sistemas de control de versiones centralizados (como apache subversión) con esto se tiene un lugar donde mantener las versiones, pero aun existe un problema, ¿Y si el servidor colapsa? Se pierde todo!. Tampoco suena muy eficiente, entonces que necesitamos? La respuesta es sencilla, necesitamos un sistema de control de versiones distribuidos. En los sistemas de control de versiones distribuidos (ejemplos: Git, Mercurial, entre otros), los clientes no sólo descargan la última versión de los archivos, la replican completamente en sus computadoras. Así, si un servidor colapsa, cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo (es lo mismo que tener una copia de seguridad completa de los datos).

Ahora en día con servicios como los de [github](#), [bitbucket](#), entre otros, mantener un proyecto es más fácil y nos ahorramos la necesidad de tener un servidor dedicado para mantener el proyecto al alcance de todos, además los usuarios pueden acceder a los repositorios desde cualquier lugar con una conexión a internet.

Lo anterior fue una breve reseña de lo que son los sistemas de control de versiones y del porque deberíamos usarlos.

INTRODUCCIÓN

A continuación nos centraremos en utilizar y poner en práctica uno de los sistemas de control de versiones (SVC por sus siglas en inglés) más utilizados en el mundo, hablamos de **git**, creado por **Linus Torvalds** el mismo creador del Kernel Linux y por la comunidad que trabaja en el mismo kernel.

En el año 2005 nació git como una necesidad de los desarrolladores del kernel Linux de tener su propio sistema de control de versiones, después de varios años usando BitKeeper (un software propietario) el cual dejó de ser una opción viable para la comunidad de desarrolladores.

Git es distribuido bajo licencia GNU GPL v2 y esta disponible para múltiples sistemas operativos entre ellos, y sin duda alguna, GNU/Linux.

El presente documento está hecho a manera de tutorial con el objetivo que sea fácil y entretenido utilizar git.

Este tutorial no pretende ser un documento extenso sobre git, más bien es una breve introducción al mismo y sus comandos más usados, aunque con el tiempo pueda que este documento crezca, todo depende de quienes colaboren.

Para más información sobre git visitar la documentación oficial en <http://git-scm.com/doc>.

INSTALACIÓN

Para poder seguir este tutorial, lo único que necesitarás será una distro GNU/Linux, una terminal para ejecutar un par de comandos y sobre todo muchas ganas de aprender e investigar.

INSTALANDO GIT EN TU DISTRO FAVORITA

La forma más sencilla de instalar git en nuestro ordenador es utilizando el gestor de paquetes de nuestra distro favorita e instalar git desde sus repositorios, git seguramente está en los repositorios de las distros más conocidas por lo que vamos a proceder de la siguiente manera:

Fedora, CentOS y derivados:

```
# yum install git-core
```

o bien con:

```
$ sudo yum install git-core
```

nota: si “git-core” no existe en los repositorios utilizar “git” en su lugar.

Debian y derivados:

```
# apt-get install git
```

o

```
$ sudo apt-get install git
```

Arch linux y derivados:

```
# pacman -S git
```

o también con:

```
$ sudo pacman -S git
```

Si tu distro no esta entre las mencionadas, visita la documentación de tu distro, seguramente encontraras los pasos necesarios para instalar git en tu sistema favorito.

La opción alternativa a usar el gestor de paquetes es compilar el código fuente nosotros mismos. Si quieres compilar el código fuente puedes buscar en el siguiente enlace los pasos a seguir: <http://git-scm.com/book/en/Getting-Started-Installing-Git>.

CONFIGURACIÓN INICIAL

Lo primero que debemos hacer, luego de instalar git, es configurar nuestro nombre y email con el que git nos identificará cada vez que trabajemos sobre nuestros proyectos.

Git permite configurar varias opciones y solo es necesario hacerlo una vez, por el momento nos centraremos en lo básico, como siempre, recomendamos indagar un poco en la documentación oficial para más detalles.

El comando que usaremos para configurar nuestro entorno git sera: **git config**, que nos permiten definir variables de configuración.

Comencemos por decir a git quienes somos y cual es nuestro email, con esto git y demás usuarios sabrán quien ha hecho cambios (commits).

```
$ git config --global user.name "Mr. Floyd"
$ git config --global user.email mr.floyd@ues.edu.sv
```

con **--global** estas diciendo a git que use esta información en todo lo que hagas en el sistema.

Puedes ver tu configuración actual ejecutando:

```
$ git config --list
user.name= Mr. Floyd
user.email=mr.floyd@cshluesocc.org
```

puedes además preguntar a git por parámetros específicos definidos en tu configuración:

```
$ git config user.email
mr.floyd@cshluesocc.org
```

Si necesitas una lista completa de los comandos usados en git puedes ejecutar en tu terminal:

```
$ git help
```

o bien consultar por un comando específico con **git help «comando»**:

```
$ git help config
```

o como usualmente lo hacemos en GNU/Linux usando '**man**'

```
$ man git-config
```

GIT BÁSICO

Ya hemos configurado nuestro entorno git, ahora es hora de comenzar a trabajar con el.

Vamos a comenzar creando un directorio en el lugar que queramos y le diremos a git que tome el control de todo lo que en el hagamos:

```
$ mkdir practica-git && cd practica-git
$ git init
Initialized empty Git repository in ~/practica-git/.git/
```

git init creará un subdirectorio llamado **.git** donde se encuentran los archivos necesarios para llevar el control de tu directorio actual.

```
$ ls -a
$ ls .git/
branches  config  description  HEAD  hooks  info  objects  refs
```

Comencemos con un comando que usaremos mucho a lo largo de este tutorial:

```
$ git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

git status nos da información sobre nuestro repositorio actual, nos dice si algo cambio, si hay conflicto, etc.

Creemos un archivo sencillo dentro de nuestro directorio actual y veamos que nos dice git status:

```
$ touch script.py
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#    script.py
nothing added to commit but untracked files present (use "git add" to track)
```

Nos damos cuenta que git ha detectado que hemos hecho cambios en nuestro directorio, en este caso hemos agregado un archivo nuevo y git nos pide que usemos “**git add**” para poder llevar control de este archivo, procedamos entonces:

```
$ git add script.py
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:   script.py
```

Hemos incluido nuestro archivo, pero falta un paso y es confirmar (commit) los cambios:

```
$ git commit -m "script inicial"
[master (root-commit) 3ccd2e0] script inicial
0 files changed
create mode 100644 script.py
$ git status
# On branch master
nothing to commit (working directory clean)
```

git status nos dice que tenemos un directorio de trabajo limpio, esto quiere decir ya tenemos un repositorio git con un archivo en seguimiento y una confirmación inicial. Git además nos dice en que rama estamos trabajando, en nuestro caso “master” que es la creada por default (más adelante crearemos y usaremos ramas).

Como hemos visto el proceso de agregar un archivo para que git lleve su control consta de dos pasos, agregar (add) y confirmar (commit). El comando **git add «archivo»** puede resultar tedioso si tenemos varios archivos y queremos agregarlos todos, pues tendríamos

que agregar archivo por archivo, pero git nos ofrece otro comando útil: **git add** . (el punto después del add indica a git que vamos a agregar al repositorio todos los archivos dentro de nuestro directorio, luego bastaría con hacer un commit para confirmar los cambios).

Git nos permite ver los commits que se han hecho en nuestro repositorio, para ello usaremos el comando **log** así:

```
$ git log
commit 3ccd2e06ed65ce95f7af3df9ca28c0dfb5af5fed
Author: Mr. Floyd <mr.floyd@ues.edu.sv>
Date:   Fri Jul 18 23:39:18 2014 -0600

    script inicial
```

Podemos notar que el commit tiene una serie de números y letras, eso es el identificador único (ID) que git asigna a nuestro commit, git utiliza el hash sha1 para identificar cada commit como único dentro de nuestro repositorio, por lo que podemos hacer algo como:

```
$ git show 3ccd2e06ed65ce95f7af3df9ca28c0dfb5af5fed
commit 3ccd2e06ed65ce95f7af3df9ca28c0dfb5af5fed
Author: Mr. Floyd <mr.floyd@cshluesocc.org>
Date:   Fri Jul 18 23:39:18 2014 -0600

    script inicial

diff --git a/script.py b/script.py
new file mode 100644
index 0000000..e69de29
```

con esto, decimos a git que nos muestre con detalle el commit con ID 3ccd2e06ed65ce95f7af3df9ca28c0dfb5af5fed. Escribir todo el ID del commit resulta un poco tedioso, pero no es necesario escribir todos los caracteres, podemos obtener el mismo resultado anterior haciendo:

```
$ git show 3ccd2e
```

git show nos permite utilizar los primeros 5, 6 o 7 caracteres del ID del commit en lugar de los 40, con esto evitamos escribir tanto. También podemos usar el comando **show** sin especificar un id y git nos dará detalle del commit más reciente.

```
$ git show
```

Ahora vamos a modificar el script que hemos creado y ver como reacciona git ante los cambios que hagamos, para el ejemplo agregaremos un par de lineas:

Nota: con **cat** mostramos las lineas que hemos agregado a nuestro script.py, puedes agregar las lineas que desees, al fin y al cabo solo nos interesa lo que nos muestre git.

```
$ cat script
#!/usr/bin/env python

print "hola mundo, somos la CSHL"
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#    modified:   script.py
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Git nos informa que el archivo que creamos ha sido modificado y pide que confirmemos sus modificaciones, nos sugiere dos comandos, “**git add**” o “**git commit -a**”, el primero ya lo hemos utilizado y sabemos que consta de dos pasos, agregar (add) y confirmar (commit), el segundo comando que nos sugiere puede efectuar los dos pasos anteriores de una sola vez, por conveniencia usaremos el segundo pero agregaremos una opción más, así:

```
$ git commit -am "Hola mundo desde la CSHL"
[master 79ba623] Hola mundo desde la CSHL
 1 file changed, 3 insertions(+)
 mode change 100644 => 100755 script.py
```

Veamos ahora que información podemos obtener sobre los últimos cambios hechos:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```



```
$ git log --graph
* commit 79ba623acf7741e21cc1b6ccb3435301405ec0d3
| Author: Mr. Floyd <mr.floyd@cshluesocc.org>
| Date:   Sat Jul 19 00:36:48 2014 -0600
|
|     Hola mundo desde la CSHL
|
* commit 3ccd2e06ed65ce95f7af3df9ca28c0dfb5af5fed
  Author: Mr. Floyd <mr.floyd@cshluesocc.org>
  Date:   Fri Jul 18 23:39:18 2014 -0600

    script inicial
```

Vemos como status nos muestra que no hay más cambios en nuestro archivos, el comando **log --graph** nos muestra una pequeña gráfica en a modo consola de los commits que hemos hecho.

IGNORANDO ARCHIVOS

Podemos también decirle a git que ignore ciertos archivos y/o directorios, estos no serán rastreados y no serán parte del repositorio aunque estos estén en nuestro directorio actual, esto es necesario muchas veces porque a veces los editores que usamos para trabajar crean archivos temporales en el directorio actual, archivos como `~script.py` que realmente no queremos que sean parte de nuestro proyecto, para eso git tiene un archivo especial llamado **.gitignore** que debemos crearlo y hacerle un commit en nuestra repo.

El archivo **.gitignore** es un simple archivo de texto que indica que tipo de archivos queremos que git ignore, creemos nuestro **.gitignore**:

```
$ touch .gitignore
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   .gitignore
nothing added to commit but untracked files present (use "git add" to track)
```

git nos dice que hay un nuevo archivo, en el vamos a agregar lo siguiente:

```
$ cat .gitignore
# ignorar los archivos que coincidan con los siguientes patrones

~*
*.so
*.log
```

Agreguemos el archivo anterior a nuestro repositorio

```
$ git add .gitignore
$ git commit -am "ignorar archivos"
[master 8597e52] ignorar archivos
 1 file changed, 6 insertions(+)
 create mode 100644 .gitignore
$ git status
# On branch master
nothing to commit (working directory clean)
```

Ahora podemos probar agregando un archivo que cumpla con los patrones que se especificaron en **.gitignore**, probemos si funciona:

```
$ touch ~temp
$ git status
# On branch master
nothing to commit (working directory clean)
```

efectivamente git ha ignorado ese archivo, puedes agregar los patrones que consideres necesarios para que git ignore los archivos que coincidan con lo que has definido.

En .gitignore podemos usar expresiones regulares, por ejemplo, hemos usado ~* que indica a git que ignore todos los archivos que comiencen con el simbolo ~, así podemos crear patrones para hacer que git se comporte de manera más inteligente. Notar que también hemos usado la almohadilla # para escribir un comentario, en .gitignore todo el texto precedido por la almohadilla sera ignorado y no afectara el su funcionamiento.

Podemos ignorar directorios completos, por ejemplo:

```
#ignorar los directorios siguientes
bower/
libs/
```

Lo anterior le dice a git que ignore el contenido de los directorios bower y libs, podemos además decirle a git que ignore todo el directorio excepto ciertos archivos, por ejemplo:

```
#ignorar los directorios siguientes
bower/
libs/
#no ignorar el siguiente archivo
!libs/miScript.py
```

Le hemos dicho a git que ignore todos los archivos en el directorio libs/ excepto por el archivo miScript.py, simplemente hemos usado el simbolo ! para idicar que ese archivo no debe ser ignorado, esto es realmente util en proyectos grandes donde hay archivos que no deben ser ignorados.

Una lista extensa de ejemplos de .gitignore para diferentes tipos de proyectos puede ser encontrada en: <https://github.com/github/gitignore>. Más información de gitignore: <http://git-scm.com/docs/gitignore>.

REVIRTIENDO CAMBIOS

Para continuar con el tutorial vamos a crear un par de archivos más en nuestro repositorio actual.

```
$ touch script2.py README
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   README
#   script2.py
nothing added to commit but untracked files present (use "git add" to track)
```

Efectivamente git sabe que hemos hecho cambios en nuestro repositorio y espera a que agreguemos los nuevos archivos y que confirmemos su seguimiento, procedamos entonces:

```
$ git add .
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   new file:   script2.py
```

git nos solicita que confirmemos cambios, también nos da la opción de revertir la acción anterior con reset HEAD, veamos que sucede entonces:

```
$ git reset HEAD
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   README
#   script2.py
nothing added to commit but untracked files present (use "git add" to track)
```

Hemos revertido la acción anterior y volvemos a tener dos archivos nuevos sin agregar en nuestro repositorio, los agregaremos de nuevo para continuar con el tutorial.

```
$ git add .
$ git commit -am "segundo script y README"
[master fc59723] segundo script y README
0 files changed
create mode 100644 README
create mode 100644 script2.py
```

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

```
$ git log --pretty=short
commit fc597238a5147ea8ece42ca6817f932a366b6069
Author: Mr. Floyd <mr.floyd@ues.edu.sv>

    segundo script y README

commit 79ba623acf7741e21cc1b6ccb3435301405ec0d3
Author: Mr. Floyd <mr.floyd@ues.edu.sv>

    Hola mundo estilo CSHL

commit 3ccd2e06ed65ce95f7af3df9ca28c0dfb5af5fed
Author: Mr. Floyd <mr.floyd@ues.edu.sv>

    script inicial
```

El comando **log** tiene varias opciones útiles (más información en la documentación oficial) que nos dan detalle de lo que hemos estado haciendo en nuestro repositorio, el comando anterior muestra el detalle de manera bastante amigable y sencilla al usuario.

Vamos a editar un archivo más para mostrar otras opciones útiles de git:

```
$ cat README
Comunidad de software y hardware libre UES FM0cc
Introducción a git

$ git diff
diff --git a/README b/README
index e69de29..d603bef 100644
--- a/README
+++ b/README
@@ -0,0 +1,3 @@
+Comunidad de software y hardware libre UES FM0cc
+
+Introducción a git
```

El comando **git diff** compara lo que hay actualmente en nuestro directorio con lo que se tenía previamente confirmado, indica los cambios hechos que aun no han sido confirmados.

```
$ git commit -am "README"
```

Acabamos de confirmar los cambios en README. ¿Que sucede si nos equivocamos en algo al editar el archivo o si la descripción del commit no nos gusta del todo y queremos revertir la confirmación? Pues sencillo hagamos un reset al ultimo commit que hemos hecho:

```
$ git log -1 --pretty=short
commit 592577537bf0587cc4bf80b57b74357081787690
Author: Mr. Floyd <mr.floyd@ues.edu.sv>

README
```

Veamos cual fue el último commit hecho, ahora pasemos a revertir el commit

```
$ git reset --soft HEAD^
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    modified:   README
```

```
$ cat README
Comunidad de software y hardware libre UES FM0cc
Introducción a git
```

Hemos utilizado `git reset --soft HEAD^` que revierte el ultimo commit realizado manteniendo los cambios que se hicieron en el archivo o archivos modificados, al usar `git status` vemos que efectivamente regresamos al estado anterior del commit, ahora podemos, si lo queremos, editar de nuevo el archivo y hacer un nuevo commit.

Vamos a hacer commit de nuevo para continuar:

```
$ git commit -am "README actualizado"
[master e12b294] README actualizado
1 file changed, 3 insertions(+)
```

Si por casualidad queremos eliminar permanentemente un commit o varios, podemos usar el comando `reset` con la opción `--hard` y el ID del commit: sintaxis: `$ git reset --hard «commit sha1»` Es importante mencionar que este comando es peligroso, ya que con esto estaríamos borrando los commit que se han hecho posteriormente al commit que hemos especificado, no habrá forma de revertir los cambios ¿o quizás sí?

«tendríamos que investigar un poco, de paso aprendemos algo nuevo y lo compartimos acá...»

Si por ejemplo queremos eliminar todos los archivos que no estan siendo seguidos por git en el repositorio actual git nos da un comando sencillo y util para esta tarea:

```
$ git clean -f
```

Lo anterior indica a git que borre todo lo que no ha sido previamente añadido al control de versión, para más informacion sobre este comando ir a la documentación oficial.

REMOVER Y MOVER ARCHIVOS

Si queremos eliminar archivos de nuestro repositorio, git ofrece otro comando útil llamado **git rm** muy similar al comando **rm** unix.

Hablemos un poco sobre lo que hace este comando **git rm**, este elimina el o los archivos de nuestro repositorio y pide una confirmación para mantener el registro, el problema con este comando es que elimina el archivo del repositorio y del directorio en el que estamos trabajando, es como hacer **rm «archivo»** en una terminal GNU/Linux (pero si formo parte del histórico de git, es posible recuperarlo), git es bondadoso y ofrece la opción de eliminar el archivo del repositorio pero dejando el archivo intacto en nuestro directorio. Si hemos hecho commit a un archivo por accidente, podemos usar **git rm --cached «archivo»** git lo dejara tal y como estaba antes de hacer el commit, veamos esto con un ejemplo sencillo:

Vamos a crear un archivo y lo incluiremos “accidentalmente” a nuestro repositorio.

```
$ echo "Oops!" > oops
$ git rm oops
fatal: pathspec 'oops' did not match any files
```

La operación no funciona porque el archivo aun no es parte de nuestro repositorio.

```
$ git add oops
$ git commit -am "commit accidental"
[master 8127794] commit accidental
 1 file changed, 1 insertion(+)
 create mode 100644 oops

$ git log -1 --pretty=short
commit 812779449b6fb39450d1426899a39945b340fb19
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    commit accidental
```

Hemos agregado el archivo “accidentalmente” a nuestro repositorio. Como no lo queríamos agregar, pero si queremos mantenerlo en nuestro directorio procedamos a quitarlo del repositorio:

```
$ git rm --cached oops
rm 'oops'
```



```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   deleted:    oops
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   oops

$ ls
oops  README  script2.py  script.py
```

```
$ git commit -am "oops elimiado del repositorio"
[master 22592c0] oops elimiado del repositorio
 1 file changed, 1 deletion(-)
 delete mode 100644 oops

$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   oops
```

Git ha hecho bien su trabajo, ha borrado del repositorio el archivo pero lo mantiene intacto en nuestro directorio. Agregaremos de nuevo el archivo y esta vez lo borraremos con git rm.

```
$ git add oops
$ git commit -am "commit accidental"
[master 43af71f] commit accidental
 1 file changed, 1 insertion(+)
 create mode 100644 oops
$ git rm oops
rm 'oops'

$ ls
README  script2.py  script.py

$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   deleted:    oops
#
```

```
$ git commit -am "oops borrado correctamente"
[master 0f6d43a] oops borrado correctamente
1 file changed, 1 deletion(-)
delete mode 100644 oops
```

Detengámonos un poco y veamos los últimos 5 commits que hemos hecho:

```
$ git log -5 --pretty=short
commit 0f6d43a9c0687fa70ca95700aae45d75b782bb43
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    oops borrado correctamente

commit 43af71fcd7a71e468ebfa57d059f51bd5a9cacee
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    commit accidental

commit 22592c0f14990764a21437d792c07fadca50d24a
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    oops elimiado del repositorio

commit 812779449b6fb39450d1426899a39945b340fb19
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    commit accidental

commit e12b294a508bee8602943644c3e12a4c2f81583b
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    README actualizado
```

Conozcamos ahora otro útil comando **git mv**, es el análogo a mv en unix para mover y renombrar archivos y directorios. Según la documentación oficial: “Git no hace un seguimiento explícito del movimiento de archivos. Si renombas un archivo, en Git no se almacena ningún metadato que indique que lo has renombrado...”.

«quedan dudas sobre eso, si sabes algo más no dudes en compartirlo acá...»

Renombremos uno de nuestros archivos:

```
$ git mv script.py holaMundo.py
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    renamed:    script.py -> holaMundo.py

$ git commit -am "script.py renombrado a holaMundo.py"
[master f3e2083] script.py renombrado a holaMundo.py
1 file changed, 0 insertions(+), 0 deletions(-)
rename script.py => holaMundo.py (100%)
```

TAGGING

Cuando tenemos un historial de cambios lo suficientemente grande:

```
$ git log
commit fce0e7c556544fe200d78e563f7678cf5de7959f
Author: Carlos Cárcamo <carloscarcamo.m@gmail.com>
Date:   Mon Feb 23 23:17:32 2015 -0600

    Update revirtiendo_cambios.md

commit af00094f53070d509c6b92d95c2c0641e0d9abfd
Author: Carlos Cárcamo <carloscarcamo.m@gmail.com>
Date:   Mon Feb 23 23:14:59 2015 -0600

    Update ignorando_archivos.md

commit ebd2e359cf8977ffda7da37f10416b0c9c180029
Author: Carlos Cárcamo <carloscarcamo.m@gmail.com>
Date:   Mon Feb 23 23:11:55 2015 -0600

    Update git_basico.md

commit 1ad7986a2cffd13b7959d35735a7f9fcc16097fd
Author: Carlos Cárcamo <carloscarcamo.m@gmail.com>
Date:   Mon Feb 23 22:55:43 2015 -0600

    Update configuracion_inicial.md
:
```

Es posible empezar a tener problemas a buscar dentro de él, realizar cambios, o regresar a algún punto específico para realizar una ramificación.

Utilizamos las etiquetas (tag) para especificar eventos en nuestro historial que sean importantes, hacer versiones o delimitar lanzamientos.

Para mostrar las diferentes etiquetas que se han otorgado en nuestro repositorio ejecutamos:

```
$ git tag
v1.0.0
v2.0.0
```

Podemos crear puntos en cualquier cambio realizado, si queremos agregar uno en nuestro estado actual:

```
$ git tag -a v3.0.0 -m "Versión 3"
```

O si queremos agregarlo a un punto anterior, utilizamos el id del commit (completo o simplificado):

```
$ git tag -a v2.1.0 -m "Segundo release de la versión 2" af00094
```

Para buscar las subversiones o lanzamientos de alguna versión en específico, podemos listar:

```
$ git tag -l "v2.*"  
v2.0.0  
v2.1.0
```

O ver sus detalles (líneas y archivos modificados):

```
$ git show v2.1.0  
tag v2.1.0  
Tagger: agustin.rumayor <agustin.rumayor@gmail.com>  
Date: Tue Sep 20 10:46:12 2016 -0500  
  
Segundo release de la versión 2  
  
commit af00094f53070d509c6b92d95c2c0641e0d9abfd  
Author: Carlos Cárcamo <carloscarcamo.m@gmail.com>  
Date: Mon Feb 23 23:14:59 2015 -0600  
  
Update ignorando_archivos.md  
  
diff --git a/ignorando_archivos.md b/ignorando_archivos.md  
index c2464f4..7ef17fc 100644  
--- a/ignorando_archivos.md  
+++ b/ignorando_archivos.md
```

Una herramienta que se vuelve esencial cuando nuestro repositorio se vuelve complejo es grep. El cual nos ayuda a buscar textos en nuestros archivos modificados en las diferentes versiones que hayamos publicado. Ya sea alguna línea que hayamos borrado hace tiempo o buscar con expresiones regulares para hacer algún cambio en alguna versión:

```
$ git grep "comando" v2.0.0
```

v2.0.0:Pushing .md:Veamos ahora la salida del comando `git status`, __“Your branch is ahead of 'origin/master' by 1 commit.”__ nos indica que tenemos cambios en nuestro repositorio local que no han sido enviados al repositorio remoto. Podemos comprobar si tenemos diferencias entre nuestra repo local y la repo remota haciendo un diff así:

v2.0.0:clonando_repositorios.md:Comencemos entonces con la practica, copiemos el vinculo “clone url” y abramos una terminal y ejecutemos el comando clone:
:

RAMIFICACIONES

La ramificación es uno de los mecanismos más interesantes y usados en los sistemas de control de versiones modernos, al hablar de ramificaciones, hablamos que hemos tomado la rama principal de desarrollo (master) y a partir de ella hemos continuado trabajando sin seguir la rama principal de desarrollo, es decir en una rama diferente.

Un ejemplo de esto podría ser el desarrollo del sistema operativo universal GNU/debian, el equipo de debian mantiene siempre tres versiones de su sistema, el estable, en pruebas y de desarrollo (actualmente estas ramas son wheezy, jessie y sid) . La **rama estable** es donde los paquetes han sido probados y están listos para salir al mundo con un mínimo o nulos bugs, en la **rama en pruebas** sin embargo es donde se mantienen los paquetes más actualizados pero que aun no ha superado la fase de pruebas para salir al mundo e integrarse a la rama estable, mientras que la **rama de desarrollo** están los paquetes, módulos del sistema, etc. que aun no están listos para producción y que están cambiando constantemente y que pueden terminar agregándose o ser eliminados en cualquier momento.

Procedamos entonces a crear nuestra primera ramificación, actualmente ya estamos trabajando bajo una rama y es la rama master que por defecto git nos crea y que es donde hemos estado realizando los cambios, así que crearemos una rama nueva, cabe mencionar que al crear una rama a partir de otra, la nueva rama contiene, inicialmente, los mismos archivos y commits que la rama base, hasta que modifiquemos o agreguemos nuevos archivos y directorios.

```
$ git branch testing
$ git status
# On branch master
nothing to commit (working directory clean)
```

Al parecer no hemos hecho nada, pero hemos creado una rama nueva llamada testing, para ver que esto es cierto ejecutemos:

```
$ git branch
* master
testing
```

El "*" nos dice en que rama estamos actualmente, cambiemos de rama entonces con git checkout:

```
$ git checkout testing
Switched to branch 'testing'
$ git status
# On branch testing
nothing to commit (working directory clean)
```

Vamos a agregar un par de líneas de código a un archivo

```
$ cat script2.py
#!/usr/bin/env python
for a in [1, 2]:
    for b in ["a", "b"]:
        print a, b
```

```
$ git status
# On branch testing
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   script2.py
#
no changes added to commit (use "git add" and/or "git commit -a")
```

```
$ git add .
$ git commit -am "combinaciones en python"
[testing d3dbf28] combinaciones en python
 1 file changed, 5 insertions(+)
 mode change 100644 => 100755 script2.py
```

Añadamos un nuevo archivo para luego unir las ramas testing y master en una sola:

```
$ touch script3.py
$ git add .
$ git commit -am "3er script"
[testing 0294943] 3er script
 0 files changed
 create mode 100644 script3.py
$ git status
# On branch testing
nothing to commit (working directory clean)
```

hagamos un simple log de los 3 últimos commits hechos en la rama testing:


```
$ git log -3 --pretty=short
commit 0294943eb590210846c2875a07b5a24b08360c52
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    3er script

commit d3dbf28039a7fede95eafea4c0f5378c144c34d6
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    combinaciones en python

commit f3e208303b789e5d944f11a6565f1514714f4132
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    script.py renombrado a holaMundo.py
```

Regresemos al la rama master y hagamos también un log:

```
$ git checkout master
Switched to branch 'master'
$ git log -3 --pretty=short
commit f3e208303b789e5d944f11a6565f1514714f4132
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    script.py renombrado a holaMundo.py

commit 0f6d43a9c0687fa70ca95700aae45d75b782bb43
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    oops borrado correctamente

commit 43af71fcd7a71e468ebfa57d059f51bd5a9cacee
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    commit accidental
```

en nuestra rama master no están los últimos commits que hicimos en la rama testing, tenemos que unir las ramas para que los cambios en testing se apliquen a la rama master, para eso usaremos el comando git merge así:

```
$ git merge testing
Updating f3e2083..0294943
Fast-forward
 script2.py |    5 +++++
 1 file changed, 5 insertions(+)
 mode change 100644 => 100755 script2.py
 create mode 100644 script3.py
```

```
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls
holaMundo.py  README  script2.py  script3.py
$ cat script2.py
#!/usr/bin/env python

for a in [1, 2]:
    for b in ["a", "b"]:
        print a, b
```

Vemos que tenemos los archivos actualizados según como los teníamos en la rama testing, hagamos un pequeño log para ver que ha pasado:

```
$ git log -4 --pretty=short
commit 0294943eb590210846c2875a07b5a24b08360c52
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    3er script

commit d3dbf28039a7fede95eafea4c0f5378c144c34d6
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    combinaciones en python

commit f3e208303b789e5d944f11a6565f1514714f4132
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    script.py renombrado a holaMundo.py

commit 0f6d43a9c0687fa70ca95700aae45d75b782bb43
Author: Mr. Floyd <mr.floyd@cshluesocc.org>

    oops borrado correctamente
```

Tenemos todo actualizado.

Un par de observaciones: si editamos/agregamos/borramos algo en una rama y no hacemos los respectivos adds, commits y nos cambiamos a una rama distinta, esta acción pueda que nos genere problemas en la rama a la que nos movimos pues esos cambios también estarán en esta última a pesar que no hayamos trabajado en ella. Por lo anterior se recomienda hacer los respectivos commit en la rama actual (dejando limpio el repositorio) antes de moverse a una rama distinta.

«Si crees que falta algo que aclarar o añadir en esta parte, por favor ayúdanos a mejorarla...»

Como ya hemos unidos nuestros cambios de la rama testing en la rama master, hagamos de cuenta que ya no necesitamos la rama testing y queremos borrarla. Para borrar una rama usamos **git branch** con la opción **-d** como en el siguiente ejemplo:

```
$ git branch -d testing
Deleted branch testing (was 060ac2d).
$ git branch
* master
```

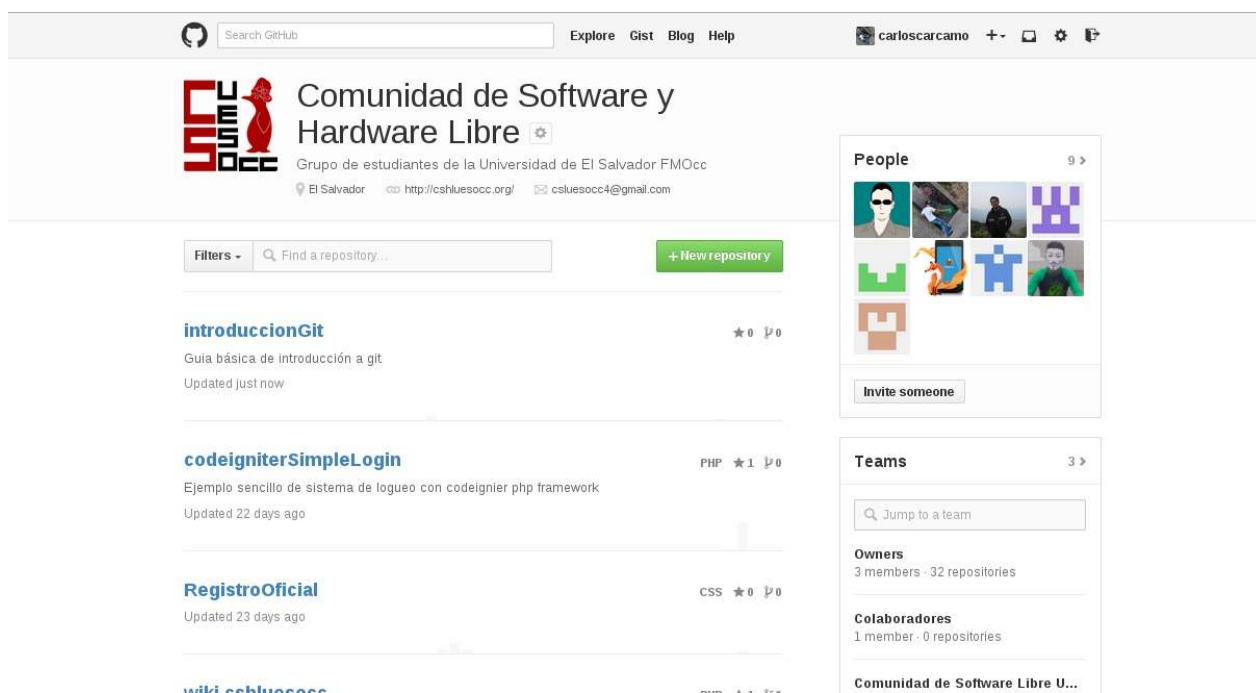
A groso modo esto es la base de las ramificaciones en git, puedes visitar la documentación oficial para más info y algunas buenas practicas. También este enlace puede ser útil <http://nvie.com/posts/a-successful-git-branching-model/> en el encontraras un par de consejos para crear un modelo de ramas exitoso para tu trabajo en git en equipo o bien proyectos personales.

«Si crees conveniente y deseas ampliar esta sección no dudes en hacerlo, toda ayuda es bienvenida :)»

REPOSITORIOS REMOTOS

Como se menciona al principio de este tutorial, existen varios sitios en internet que ofrecen servicios de alojamiento de código y que permiten utilizar git para llevar control de nuestros proyectos. En este apartado usaremos uno de los sitios más famosos en el mundo para alojar código, hablamos de github.com , github nos ofrece alojamiento gratuito de código entre muchas otras interesantes opciones, también ofrece servicio privado para empresas o individuales que quieran mantener sus repositorios en privado, nosotros, como debe ser, tenemos una cuenta en github donde publicamos código open source que puede ser clonado, pueden hacer un fork contribuir directamente, etc.

Veamos como luce entonces la interfaz de github (interfaz para organizaciones en nuestro caso):



Nota: para continuar con este tutorial deberás crear una cuenta en github (es gratis!!!) con la que crearemos un repositorio publico y trabajaremos con el.

CREANDO UN REPOSITORIO EN GITHUB

Vamos a comenzar por crear un repositorio en github con el cual trabajaremos a lo largo de este tutorial, a continuación te mostramos los pasos necesarios:

En el menú superior derecho luego de tu nombre de usuario encontraras una serie de opciones como muestra la imagen siguiente:



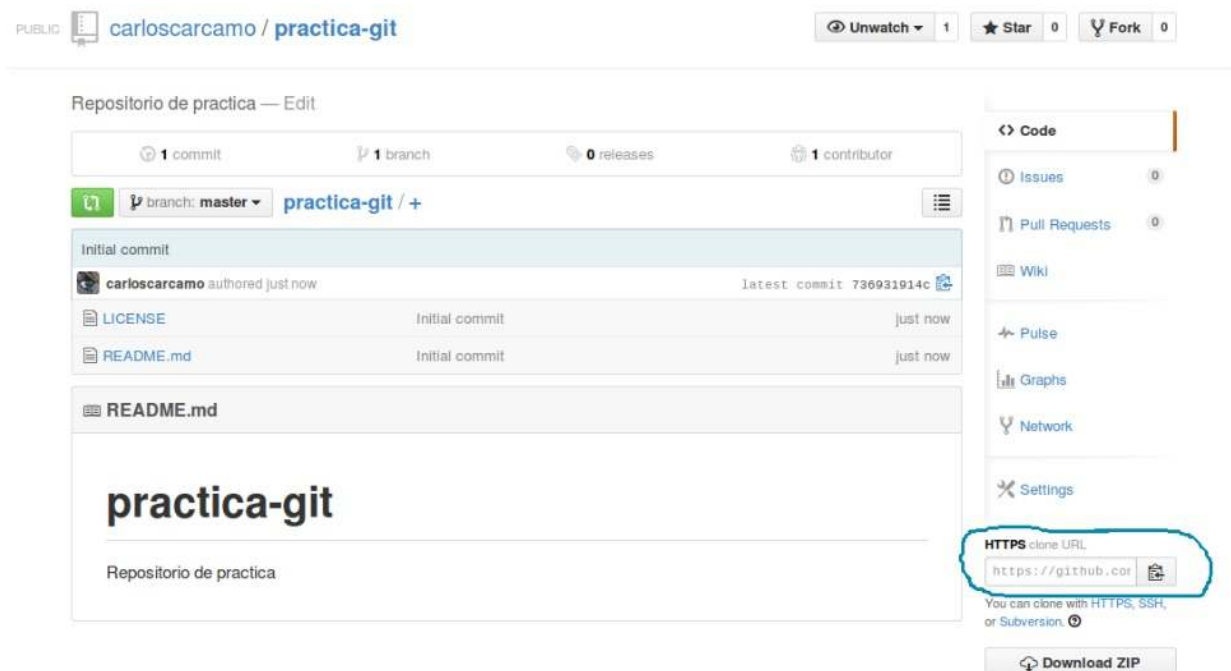
Damos click a la opción crear nuevo repositorio y nos aparece una pantalla similar a la siguiente:

A screenshot of the 'Create new repository' form in GitHub. The form has two main sections: 'Owner' and 'Repository name'. The 'Owner' is 'carloscarcamo' and the 'Repository name' is 'practica-git'. Below this, there's a 'Description (optional)' field with the text 'Repositorio de practica'. The 'Public' checkbox is selected, and the 'Private' checkbox is unselected. The 'Initialize this repository with a README' checkbox is also selected. At the bottom, there are two dropdown menus: 'Add .gitignore: None' and 'Add a license: GPL v2'. A green 'Create repository' button is at the bottom.

Procedamos entonces a crear nuestro primer repositorio en github, llenamos el formulario con los datos que queramos y damos click al botón “create repository”, nos mostrara una pantalla con una serie de opciones para el manejo de nuestros repositorios, es buena idea navegar un poco esta interfaz y ver que opciones nos ofrece github.

CLONANDO REPOSITORIOS

La imagen a continuación nos muestra la información de nuestro recién creado repositorio, pongamos atención al área marcada con azul **"HTTPS clone URL"**, esa área nos da el vínculo directo a nuestro repositorio con el cual podemos clonarlo a nuestro ordenador y poder entonces trabajar sobre el, enviando cambios (push) y obteniendo las actualizaciones más recientes del repositorio (pull).



Comencemos entonces con la practica, copiemos el vínculo "clone url" y abramos una terminal y ejecutemos el comando clone:

```
$ git clone https://github.com/carloscarcamo/practica-git.git
Cloning into 'practica-git'...
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
$ cd practica-git
$ ls
LICENSE  README.md
```

Hemos clonado exitosamente nuestro repositorio a nuestro ordenador, luego entramos al directorio y listamos los archivos que tenemos dentro, efectivamente podemos observar que tenemos dos archivos LICENSE y README que son los que se añadieron al momento de crear nuestro repositorio en github.

Podemos ver a que url apunta nuestro repositorio remoto usando git remote -v, así:

```
$ git remote -v  
origin    https://github.com/carloscarcamo/practica-git.git (fetch)  
origin    https://github.com/carloscarcamo/practica-git.git (push)
```

Git nos informa de la url a la que nuestro repositorio hará los respectivos [push](#) y [pull](#).

ENVIANDO CAMBIOS AL REPOSITORIO REMOTO

Ahora estamos listos para agregar y modificar nuevos archivos dentro de este repositorio y enviar los cambios a github, procedamos entonces agregando un par de archivos y directorios:

```
$ mkdir css js
$ touch index.html
```

Hemos añadido dos directorios y un archivo nuevos, veamos que ha detectado git dentro de nuestro repositorio:

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   index.html
nothing added to commit but untracked files present (use "git add" to track)
```

A pesar de que hemos agregado dos directorios nuevos, git no detecta esos cambios, eso es debido a que git no lleva control de los directorios sino de archivos, al momento en que agreguemos un archivo en cualquiera de esos directorios podremos ver que git nos informará de ello, creemos un archivo entonces dentro de uno de esos directorios:

```
$ touch js/app.js
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   index.html
#   js/
nothing added to commit but untracked files present (use "git add" to track)
```

Todo bien, ahora agreguemos los archivos y hagamos un commit:

```
$ git add .
$ git commit -am "Inicio de proyecto"
[master 19118ae] inicio de proyecto
0 files changed
create mode 100644 index.html
create mode 100644 js/app.js
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
nothing to commit (working directory clean)
```

Veamos ahora la salida del comando `git status`, **“Your branch is ahead of 'origin/master' by 1 commit.”** nos indica que tenemos cambios en nuestro repositorio local que no han sido enviados al repositorio remoto. Podemos comprobar si tenemos diferencias entre nuestra repo local y la repo remota haciendo un diff así:

```
$ git diff origin/master master
diff --git a/index.html b/index.html
new file mode 100644
index 0000000..e69de29
diff --git a/js/app.js b/js/app.js
new file mode 100644
index 0000000..e69de29
```

Enviemos nuestros cambios a github:

```
$ git push origin master
Username for 'https://github.com':
Password for 'https://carloscarcamo@github.com':
To https://github.com/carloscarcamo/practica-git.git
7369319..19118ae master -> master
```

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Git nos pide nuestro usuario y contraseña para poder enviar los cambios, luego podemos ver que `git status` nos muestra que tenemos nuestro repositorio local y remoto actualizados. Podemos ver también en github que los cambios han sido efectuados correctamente:

Repositorio de practica — Edit

2 commits 1 branch 0 releases 1 contributor

branch: master practica-git / +

Inicio de proyecto

carloscarcamo authored 20 minutes ago latest commit 19118ae1ab

js	Inicio de proyecto	20 minutes ago
LICENSE	Initial commit	35 minutes ago
README.md	Initial commit	35 minutes ago
index.html	Inicio de proyecto	20 minutes ago

README.md

practica-git

Repositorio de practica

Code

Issues 0

Pull Requests 0

Wiki

Pulse

Graphs

Network

Settings

HTTPS clone URL

<https://github.com>

You can clone with HTTPS, SSH, or Subversion

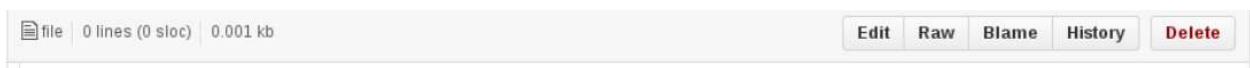
Download ZIP

hemos hecho nuestro primer commit a un repositorio remoto.

OBTENIENDO CAMBIOS DEL REPOSITORIO REMOTO

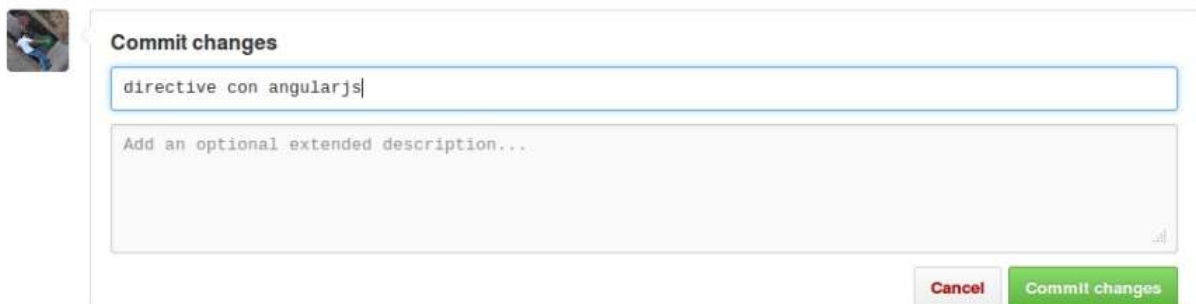
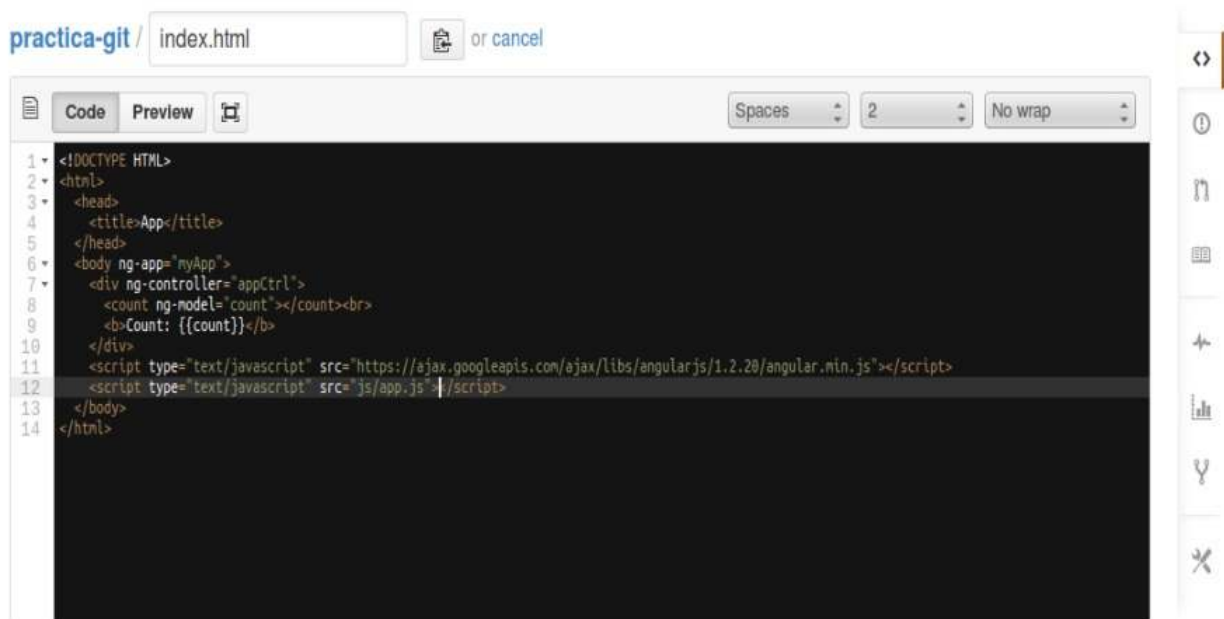
Que pasa si el repositorio ha cambiado como puedo obtener los cambios más recientes?, eso es fácil, simplemente hacemos un pull al repositorio de origen. Para este ejemplo vamos a editar los archivos en internet desde github.

Seleccionemos uno de los archivos para editar, las imágenes a continuación muestran el proceso de edición de un archivo desde github:

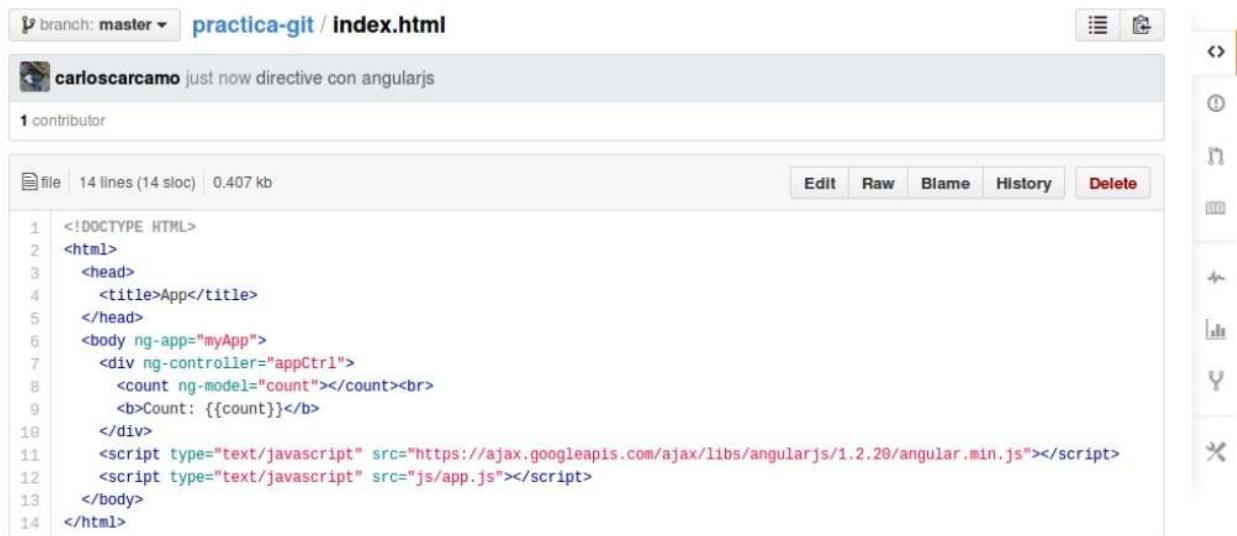


Una vez seleccionado el archivo que queremos editar, veremos una barra de botones como el de la imagen anterior.

Hacemos click al boton “edit” el cual nos mostrara una pantalla donde podemos editar nuestros archivos.

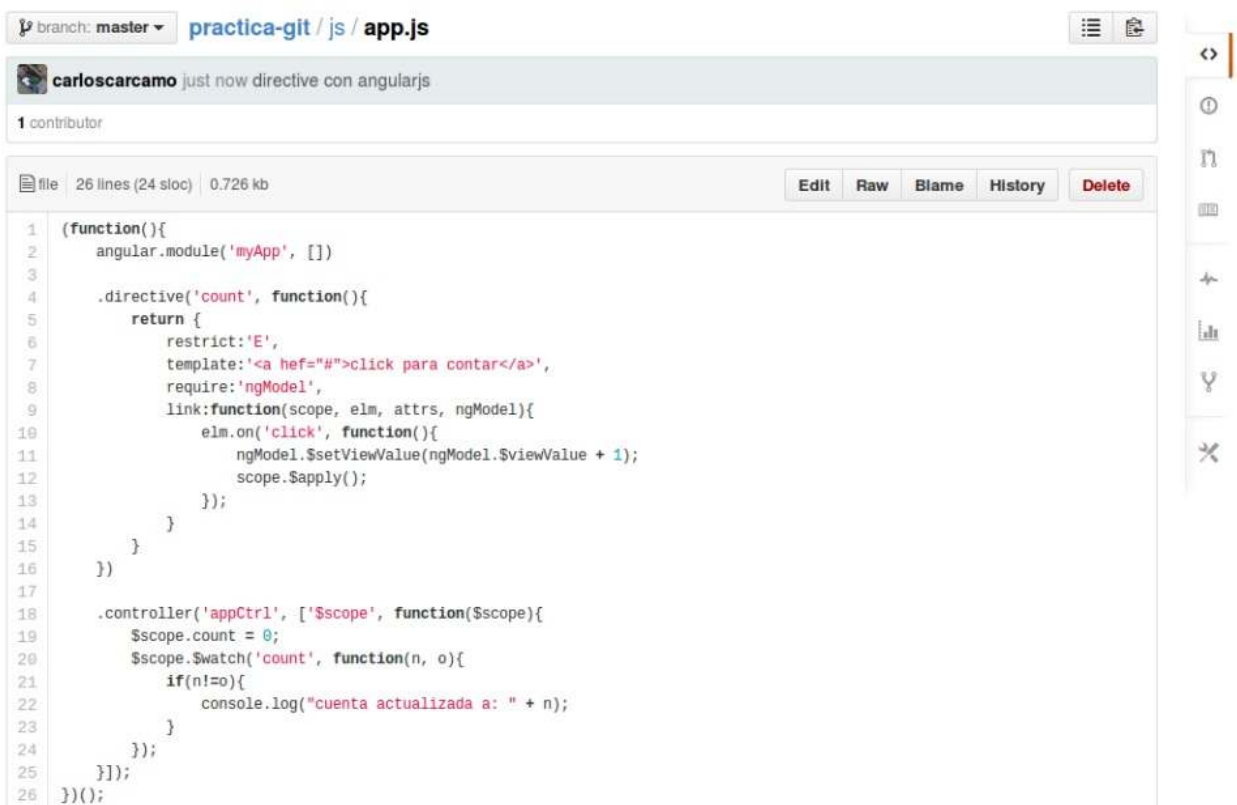


Github nos ofrece una interfaz muy sencilla para editar y hacer commit en linea. El resultado final lo podemos ver en la siguiente imagen:



```
1 <!DOCTYPE HTML>
2 <html>
3   <head>
4     <title>App</title>
5   </head>
6   <body ng-app="myApp">
7     <div ng-controller="appCtrl">
8       <count ng-model="count"></count><br>
9       <b>Count: {{count}}</b>
10    </div>
11    <script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.20/angular.min.js"></script>
12    <script type="text/javascript" src="js/app.js"></script>
13  </body>
14 </html>
```

Haremos lo mismo para editar el script en "js/app.js", el resultado sera:



```
1 (function(){
2   angular.module('myApp', [])
3
4   .directive('count', function(){
5     return {
6       restrict:'E',
7       template:'<a href="#">click para contar</a>',
8       require:'ngModel',
9       link:function(scope, elm, attrs, ngModel){
10         elm.on('click', function(){
11           ngModel.$setViewValue(ngModel.$viewValue + 1);
12           scope.$apply();
13         });
14       }
15     };
16   })
17
18   .controller('appCtrl', ['$scope', function($scope){
19     $scope.count = 0;
20     $scope.$watch('count', function(n, o){
21       if(n!=o){
22         console.log("cuenta actualizada a: " + n);
23       }
24     });
25   }]);
26 })();
```

Podemos ver en github los commit que hemos hecho hasta ahora:



Para saber si necesitamos hacer un pull al repositorio remoto, vamos a ejecutar el comando:

```
$ git remote update
Fetching origin
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 7 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (7/7), done.
From https://github.com/carloscarcamo/practica-git
   19118ae..2a6236c  master    -> origin/master
$ git status
# On branch master
# Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
#
nothing to commit (working directory clean)
```

Con `git remote update` actualizamos todas las referencias de los objetos, ramas y demás que tengamos de nuestro repositorio remoto en nuestro repositorio local, luego ejecutamos **git status** y este nos dice que nuestra rama actual esta desactualizada respecto a la rama remota. Tambien podriamos haber usado otro comando llamado **fetch** para obtener el mismo resultado anterior:

```
$ git fetch origin
```

`git remote update` y `git fetch origin` nos daran los mismos resultados, pero algunas cosas pueden variar internamente, para más información visitar la documentación oficial de [git remote](#) y [git fetch](#)

Vamos a actualizar nuestra repo local con los cambios hechos en github, esto lo haremos con un **pull** de la siguiente manera:

```
$ git pull origin
From https://github.com/carloscarcamo/practica-git
 * branch                master      -> FETCH_HEAD
Updating 19118ae..2a6236c
Fast-forward
 index.html |   14 ++++++
 js/app.js  |   26 ++++++
 2 files changed, 40 insertions(+)
$ git status
# On branch master
nothing to commit (working directory clean)
```

Hemos actualizado nuestro repositorio local con los cambios hechos en github, utilizamos `git pull «remote»` en donde especificamos la rama remota.

Ejecuta los comandos siguientes y verifica su salida, veras que en efecto tienes los archivos actualizados.

```
$ cat index.html
$ cat js/app.js
```

El proceso de trabajo con repositorios remotos es siempre similar al que hemos hecho en este tutorial usando github.

¿y esto es todo? Pues no, este tutorial es solamente una pequeña introducción, ahora queda a criterio del lector indagar más en la documentación, practicar, y trabajar con git en sus futuros proyectos.

«Recuerda que si deseas extender este tutorial, no dudes en hacerlo, solo asegúrate de compartirlo con los demás.»