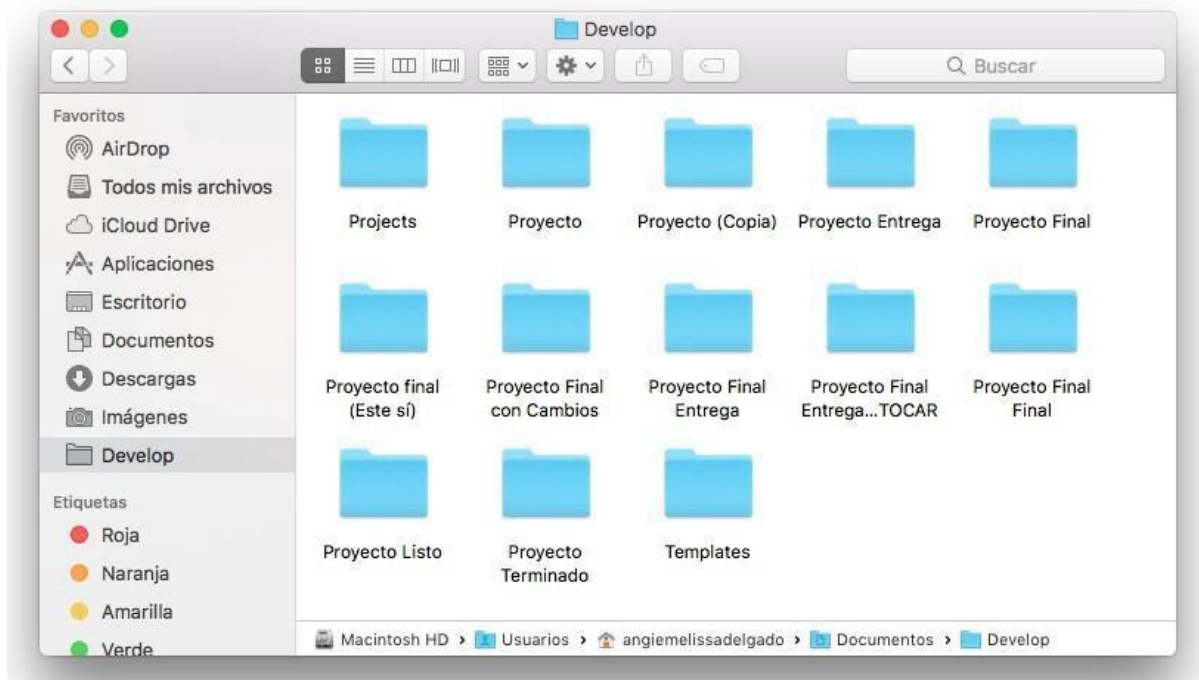


Sistemas de Control de Versiones

Hace apenas pocos años atrás, un desarrollador de software tenía que ingeniárselas para poder organizar sus proyectos de software. En otras palabras existía un gran problema a la hora de controlar los cambios que un proyecto iba teniendo a lo largo de su ciclo de vida. Tal era el dilema que acabamos nombrando archivos y/o carpetas con nombres muy poco frecuentes y poco ilustrativos. Vamos a graficar este escenario caótico con dos estados básicamente: un antes y un después de la existencia de estos sistemas de control de versiones. Veamos a través de una imagen la etapa oscura(antes) de este escenario:



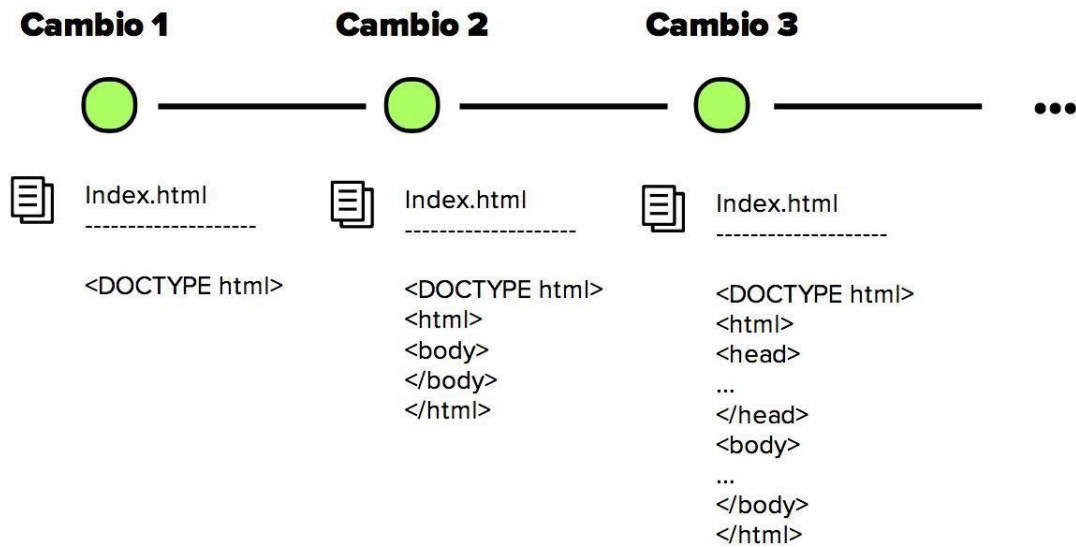
Es muy probable que en algún momento de nuestras vidas nuestra carpeta de documentos luciera como la de la imagen y nos haya tocado recurrir a tener muchas copias de nuestros proyectos, copias que requerían de toda nuestra creatividad para nombrarlas con etiquetas super útiles para poder reconocer cuál era nuestro ansiado “proyecto final”.

Tiempo después por algún accidente del destino conocemos los sistemas de control de versiones (si aún no ha ocurrido, entonces este es el accidente) y no podemos negarlo, nuestras vidas cambian y entramos a una era donde todo es mucho más bonito: el después.

Entonces ¿Qué es control de versiones?

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

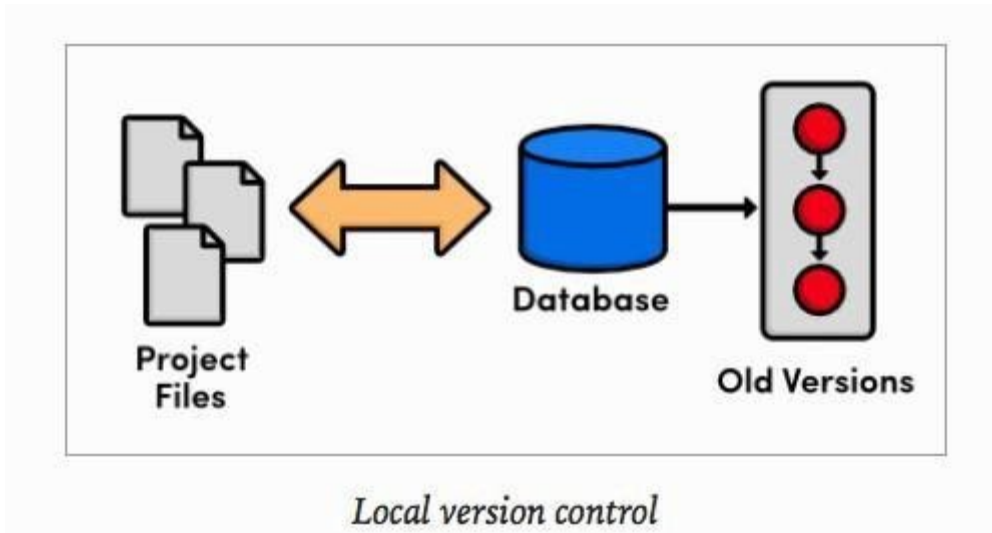
específicas más adelante.



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, Centralizados y Distribuidos.

Sistemas de Control de Versiones Locales

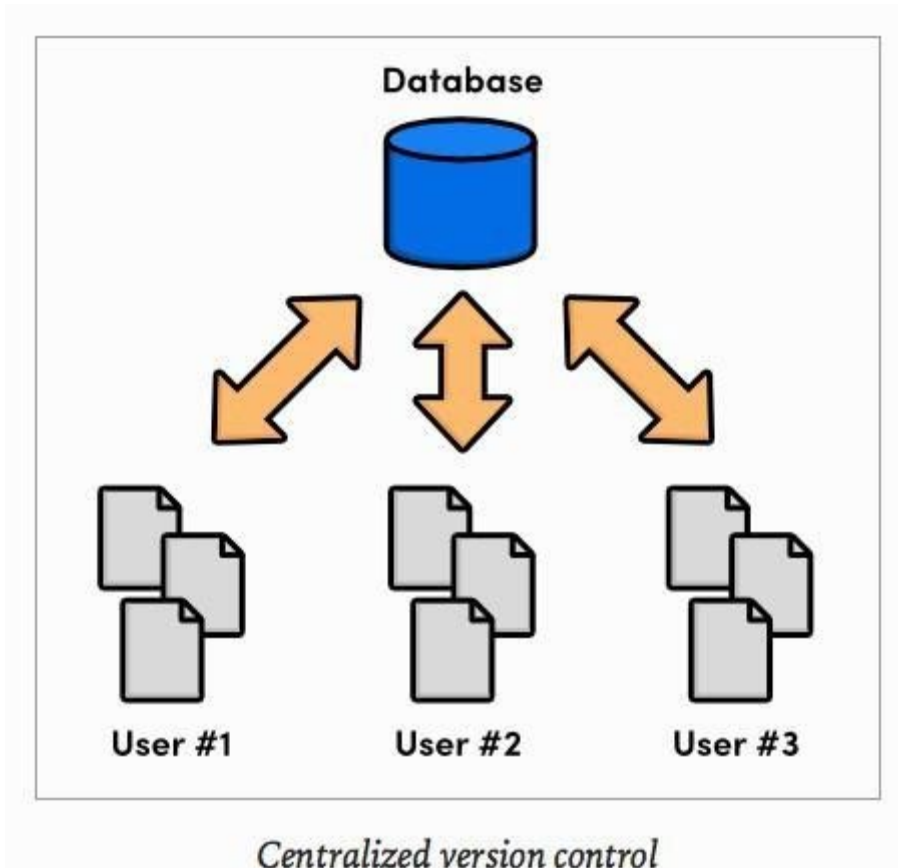
Los sistemas de control de versiones locales en vez de mantener las versiones como archivos independientes, los almacenaban en una base de datos. Cuando era necesario revisar una versión anterior del proyecto se usaba el sistema de control de versiones en vez de acceder directamente al archivo, de esta manera en cualquier momento solo se tenía una copia del proyecto, eliminando la posibilidad de confundir o eliminar versiones.



En este punto el control de versiones se llevaba a cabo en el computador de cada uno de los desarrolladores y no existía una manera eficiente de compartir el código entre ellos.

Sistemas de Control de Versiones Centralizados

Para facilitar la colaboración de múltiples desarrolladores en un solo proyecto los sistemas de control de versiones evolucionaron: en vez de almacenar los cambios y versiones en el disco duro de los desarrolladores, estos se almacenaban en un servidor.



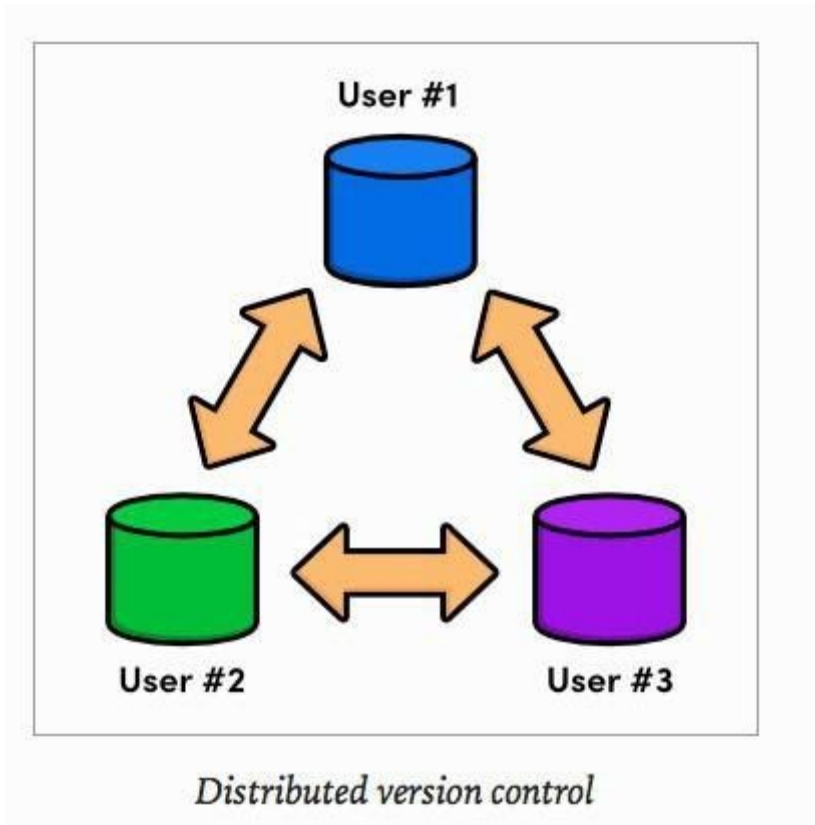
Sin embargo, aunque el avance frente a los sistemas de control de versiones locales fue enorme, los sistemas centralizados trajeron consigo nuevos retos: ¿Cómo trabajaban múltiples usuarios en un mismo archivo al mismo tiempo?

Los sistemas de control de versiones centralizados abordaron este problema impidiendo que los usuarios invalidaran el trabajo de los demás. Si dos personas editaban el mismo archivo y se presentaba un conflicto alguien debía solucionar este problema de manera manual y el desarrollo no podía continuar hasta que todos los conflictos fueran resueltos y puestos a disposición del resto del equipo.

Esta solución funcionó en proyectos que tenían relativamente pocas actualizaciones y por ende pocos conflictos pero resultó muy engorroso para proyectos con docenas de contribuyentes activos que realizaban actualizaciones a diario.

Sistemas de Control de Versiones Distribuidos

La siguiente generación de sistemas de control de versiones se alejó de la idea de un solo repositorio centralizado y optó por darle a cada desarrollador una copia local de todo el proyecto, de esta manera se construyó una red distribuida de repositorios, en la que cada desarrollador podía trabajar de manera aislada pero teniendo un mecanismo de resolución de conflictos mucho más elegante que un su versión anterior.



Al no existir un repositorio central, cada desarrollador puede trabajar a su propio ritmo, almacenar los cambios a nivel local y mezclar los conflictos que se presenten sólo cuando se requiera. Como cada usuario tiene una copia completa del proyecto el riesgo por una caída del servidor, un repositorio dañado o cualquier otro tipo de pérdida de datos es mucho menor que en cualquiera de sus predecesores.

¿Por dónde empezar?

Existen muchos Sistemas de Control de Versiones siendo algunos de los más conocidos Git, CVS, Subversion y Mercurial. En este curso abordaremos Git como control de versiones a usar. Y esto no supone un capricho, sino más bien que actualmente el 90% de las empresas de tecnología adoptan Git como su control de versiones por defecto por su gran versatilidad y confianza.

Fundamentos de Git

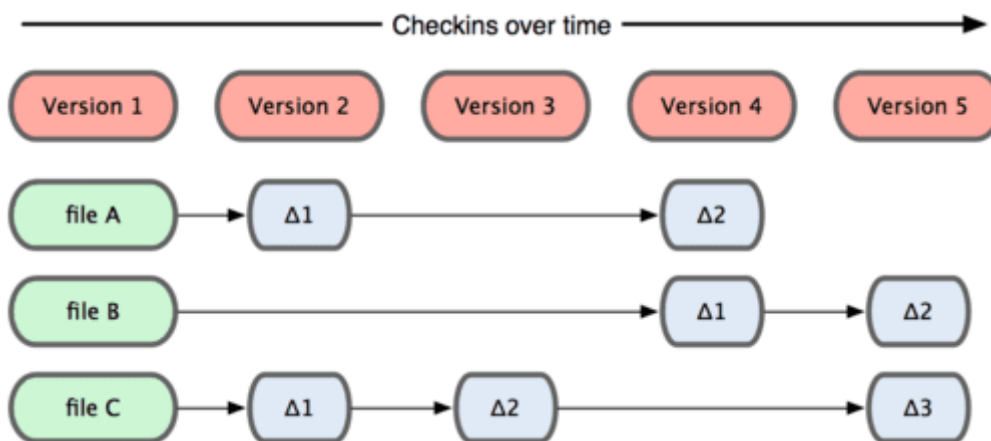
Entonces, ¿qué es Git en pocas palabras? Es una herramienta de software que realiza una función del control de versiones de código de forma distribuida.

¿Qué características tiene?

Es rápido
 Es muy potente
 No depende de un repositorio central
 Es software libre
 Fue diseñada por Linus Torvalds
 Con ella podemos mantener un historial completo de versiones
 Podemos movernos, como si tuviéramos un puntero en el tiempo, por todas las revisiones de código y desplazarnos una manera muy ágil.
 Tiene un sistema de trabajo con ramas que lo hace especialmente potente
 En cuanto a la funcionalidad de las ramas, las mismas están destinadas a provocar proyectos divergentes de un proyecto principal, para hacer experimentos o para probar nuevas funcionalidades.
 Las ramas pueden tener una línea de progreso diferente de la rama principal donde está el core de nuestro desarrollo.

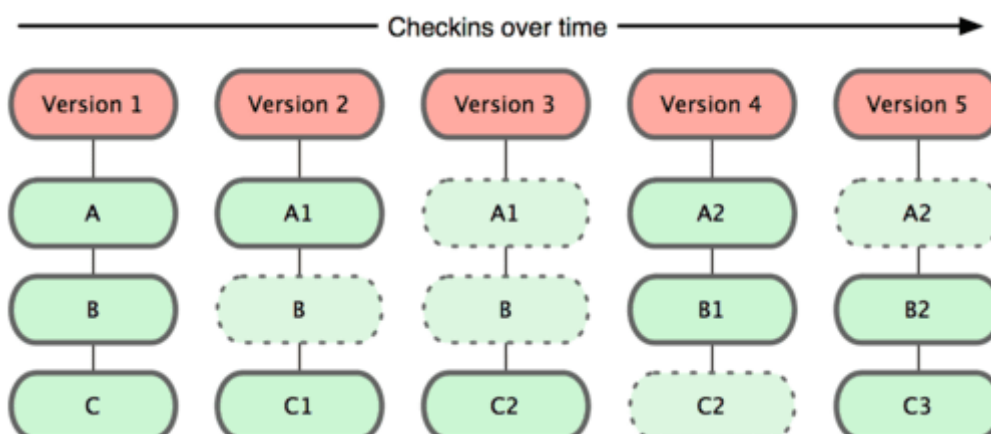
Instantáneas, no diferencias

La principal diferencia entre Git y cualquier otro VCS (Subversion y compañía incluidos) es cómo Git modela sus datos. Conceptualmente, la mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) modelan la información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo, como se muestra en la siguiente imagen.



Otros sistemas tienden a almacenar los datos como cambios de cada archivo respecto a una versión base.

Git no modela ni almacena sus datos de este modo, sino que, modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado. Git modela sus datos más como en la siguiente figura.



Git almacena la información como instantáneas del proyecto a lo largo del tiempo.

Esta es una distinción importante entre Git y prácticamente todos los demás VCSs. Hace que Git reconsidere casi todos los aspectos del control de versiones que muchos de los demás sistemas copiaron de la generación anterior. Esto hace que Git se parezca más a un mini sistema de archivos con algunas herramientas tremendamente potentes construidas sobre él, que a un VCS.

Casi cualquier operación es local

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para operar. Por lo general no se necesita información de ningún otro ordenador de tu red. Si estás acostumbrado a un CVCS donde la mayoría de las operaciones tienen esa sobrecarga del retardo de la red, este aspecto de Git te va a hacer pensar que los dioses de la velocidad han bendecido Git con poderes sobrenaturales. Cómo tienes toda la historia del proyecto ahí mismo, en tu disco local, la mayoría de las operaciones parecen prácticamente inmediatas.

Por ejemplo, para navegar por la historia del proyecto, Git no necesita salir al servidor para obtener la historia y presentarla, simplemente la lee directamente de tu base de datos local. Esto significa que ves la historia del proyecto casi al instante. Si quieres ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga, u obtener una versión antigua desde la red y hacerlo de manera local.

Esto también significa que hay muy poco que no puedas hacer si estás desconectado o sin VPN. Si te subes a un avión o a un tren y quieres trabajar un poco, puedes confirmar tus cambios felizmente hasta que consigas una conexión de red para subirlos. Si te vas a casa y no consigues que tu cliente VPN funcione correctamente, puedes seguir trabajando. En muchos otros sistemas, esto es casi imposible. En Perforce, por ejemplo, no puedes hacer mucho cuando no estás conectado al servidor; y en Subversion y CVS, puedes editar archivos, pero no puedes confirmar los cambios a tu base de datos (porque tu base de datos no tiene conexión). Esto puede no parecer gran cosa, pero te sorprendería la diferencia que puede suponer.

Integridad

Todo en Git es verificado mediante una suma de comprobación (checksum en inglés) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. Esta funcionalidad está integrada en Git al más bajo nivel y es parte integral de su filosofía. No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git lo detecte.

El mecanismo que usa Git para generar esta suma de comprobación se conoce como hash SHA-1. Se trata de una cadena de 40 caracteres hexadecimales (0-9 y a-f), y se calcula en base a los contenidos del archivo o estructura de directorios. Un hash SHA-1 tiene esta pinta:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Verás estos valores hash por todos lados en Git, ya que los usa con mucha frecuencia. De hecho, Git guarda todo no por nombre de archivo, sino por el valor hash de sus contenidos.

Git generalmente sólo añade información

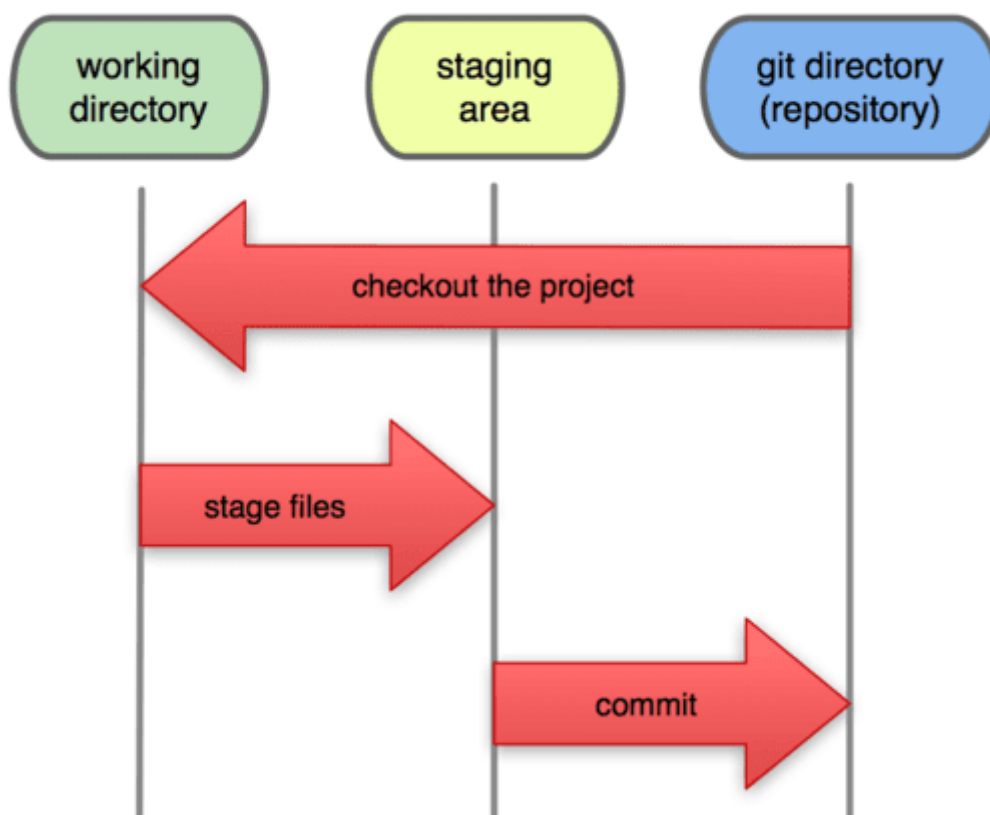
Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información. Como en cualquier VCS, puedes perder o estropear cambios que no has confirmado todavía; pero después de confirmar una instantánea en Git, es muy difícil de perder, especialmente si envías (push) tu base de datos a otro repositorio con regularidad.

Esto hace que usar Git sea un placer, porque sabemos que podemos experimentar sin peligro de fastidiar gravemente las cosas.

Los tres estados

Esto es lo más importante a recordar acerca de Git si quieres que el resto de tu proceso de aprendizaje continúe sin problemas. Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (**committed**), modificado (**modified**), y preparado (**staged**). Confirmado significa que los datos están almacenados de manera segura en tu base de datos local. Modificado significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos. Preparado significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación. Esto nos lleva a las tres secciones principales de un proyecto de Git: el directorio de Git (**Git directory**), el directorio de trabajo (**working directory**), y el área de preparación (**staging area**).

Local Operations



Directorio de trabajo, área de preparación y directorio de Git.

El directorio de Git es donde Git almacena los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otro ordenador. El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar. El área de preparación es un sencillo archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice, pero se está convirtiendo en estándar el referirse a ella como el área de preparación. El flujo de trabajo básico en Git es algo así:

- Modificas una serie de archivos en tu directorio de trabajo.

- Preparas los archivos, añadiéndolos a tu área de preparación.

- Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esas instantáneas de manera permanente en tu directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (**committed**). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (**staged**). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (**modified**).

Instalación de Git

Linux

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

o

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

Mac

<http://sourceforge.net/projects/git-osx-installer/>

Opción 1

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

Opción 2

```
$ brew install git
```

Windows

<https://git-scm.com/download/win>

Una vez instalado en cualquiera de los sistemas operativos, ya podrás empezar a usar git y para hacerlo en linux o mac , deberás abrir una terminal de comandos y escribir el comando `git --version` . En windows deberás abrir el intérprete de comandos CMD o la utilidad git bash y escribes nuevamente el comando `git --version`. En cualquier caso deberías visualizar lo siguiente:

```
$ git --version
git version 2.20.1
```

Configurando Tu identidad

Lo primero que deberías hacer cuando instalas Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios (commits) en Git usan esta información, y es introducida de manera inmutable en los commits que envías:

```
$ git config --global user.name "Johnny Guzman"
$ git config --global user.email jcguzman@gmail.com
```

Sólo necesitas hacer esto una vez si especificas la opción `--global`, ya que Git siempre usará esta información para todo lo que hagas en ese sistema. Si quieres sobrescribir esta información con otro nombre o dirección de correo para proyectos específicos, puedes ejecutar el comando sin la opción `--global` cuando estés en ese proyecto.

Tu editor por defecto

Ahora que tu identidad está configurada, puedes elegir el editor de texto por defecto que se utilizará cuando Git necesite que introduzcas un mensaje. Si no indicas nada, Git usa el editor por defecto de tu sistema, que generalmente es Vi o Vim. Si quieres usar otro editor de texto, como Nano, puedes hacer lo siguiente:

```
$ git config --global core.editor nano
```

Obtener un repositorio Git

Obteniendo un repositorio Git

Puedes obtener un proyecto Git de dos maneras. La primera toma un proyecto o directorio existente y lo importa en Git. La segunda clona un repositorio Git existente desde otro servidor.

Inicializando un repositorio en un directorio existente

Si estás empezando el seguimiento en Git de un proyecto existente, necesitas ir al directorio del proyecto y escribir:

```
$ git init
```

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

Si deseas empezar a controlar versiones de archivos existentes (a diferencia de un directorio vacío), probablemente deberías comenzar el seguimiento de esos archivos y hacer una confirmación inicial. Puedes conseguirlo con unos pocos comandos `git add` para especificar qué archivos quieres controlar, seguidos de un `commit` para confirmar los cambios:

```
$ git add index.html
$ git add README
$ git commit -m 'versión inicial del proyecto'
```

En este momento, tienes un repositorio Git con archivos bajo seguimiento, y una confirmación inicial.

Clonando un repositorio existente

Si deseas obtener una copia de un repositorio Git existente, por ejemplo un proyecto de la empresa donde vas a trabajar o un proyecto donde has decidido contribuir, el comando que necesitas es `git clone`. Cada versión de cada archivo de la historia del proyecto es descargado cuando ejecutas `git clone`. De hecho, si el disco de tu servidor se corrompe, puedes usar cualquiera de los clones en cualquiera de los clientes para devolver al servidor al estado en el que estaba cuando fue clonado.

Puedes clonar un repositorio con `git clone [url]`. Por ejemplo, si quieres clonar la librería `Lodash`, harías algo así:

```
$ git clone git@github.com:lodash/lodash.git
```

Esto crea un directorio llamado "lodash", inicializa un directorio `.git` en su interior, descarga toda la información de ese repositorio, y saca una copia de trabajo de la última versión. Si te metes en el nuevo directorio `lodash`, verás que están los archivos del proyecto, listos para ser utilizados. Si quieres clonar el repositorio a un directorio con otro nombre que no sea `lodash`, puedes especificarlo con la siguiente opción de línea de comandos:

```
$ git clone git@github.com:lodash/lodash.git mylodash
```

Ese comando hace lo mismo que el anterior, pero el directorio de destino se llamará `mylodash`.

Git te permite usar distintos protocolos de transferencia. El ejemplo anterior usa el protocolo `git://`, pero también te puedes encontrar con `http(s)://` o `usuario@servidor:ruta.git`, que utiliza el protocolo de

transferencia SSH.

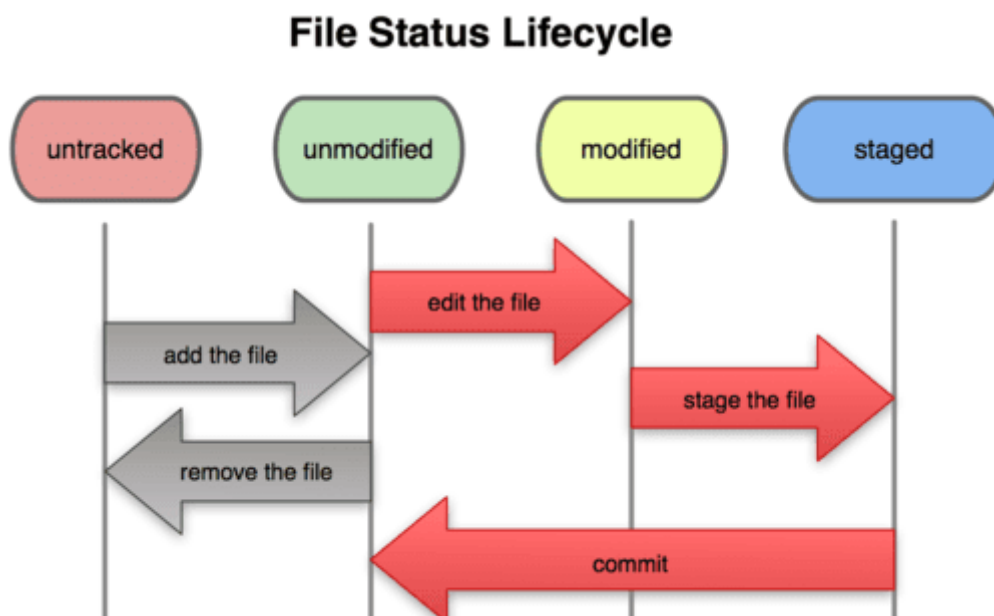
Guardar cambios en el repositorio

Tienes un repositorio Git completo, y una copia de trabajo de los archivos de ese proyecto. Necesitas hacer algunos cambios, y confirmar instantáneas de esos cambios a tu repositorio cada vez que el proyecto alcance un estado que desees grabar.

Recuerda que cada archivo de tu directorio de trabajo puede estar en uno de estos dos estados: bajo seguimiento (tracked), o sin seguimiento (untracked). Los archivos bajo seguimiento son aquellos que existían en la última instantánea; pueden estar sin modificaciones, modificados, o preparados. Los archivos sin seguimiento son todos los demás —cualquier archivo de tu directorio que no estuviese en tu última instantánea ni está en tu área de preparación. La primera vez que clonas un repositorio, todos tus archivos estarán bajo seguimiento y sin modificaciones, ya que los acabas de copiar y no has modificado nada.

A medida que editas archivos, Git los ve como modificados, porque los has cambiado desde tu última confirmación. Preparas estos archivos modificados y luego confirmas todos los cambios que hayas preparado, y el ciclo se repite. Este proceso queda ilustrado en la siguiente figura.

El ciclo de vida del estado de tus archivos.



Comprobando el estado de tus archivos

Tu principal herramienta para determinar qué archivos están en qué estado es el comando `git status`. Si ejecutas este comando justo después de clonar un repositorio, deberías ver algo así:

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

Esto significa que tienes un directorio de trabajo limpio, en otras palabras, no tienes archivos bajo seguimiento y modificados. Git tampoco ve ningún archivo que no esté bajo seguimiento, o estaría listado ahí. Por último, el comando te dice en qué rama estás. Por ahora, esa rama siempre es "**master**", que es la predeterminada. No te preocupes de eso por ahora, mas adelante te explicaremos qué significa esto de las ramas. Digamos que añades un nuevo archivo a tu proyecto, un sencillo archivo README. Si el archivo no existía y ejecutas **git status**, verás tus archivos sin seguimiento así:

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

Puedes ver que tu nuevo archivo README aparece bajo la cabecera "Archivos sin seguimiento" ("Untracked files") de la salida del comando. Sin seguimiento significa básicamente que Git ve un archivo que no estaba en la instantánea anterior; Git no empezará a incluirlo en las confirmaciones de tus instantáneas hasta que se lo indiques explícitamente. Lo hace para que no incluyas accidentalmente archivos binarios generados u otros archivos que no tenías intención de incluir.

Seguimiento de nuevos archivos

Para empezar el seguimiento de un nuevo archivo se usa el comando **git add**. Iniciaremos el seguimiento del archivo README ejecutando esto:

```
$ git add README
```

Si vuelves a ejecutar el comando **git status**, verás que tu README está ahora bajo seguimiento y preparado:

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

Puedes ver que está preparado porque aparece bajo la cabecera "Cambios a confirmar" ("Changes to be committed"). Si confirmas ahora, la versión del archivo en el momento de ejecutar **git add** será la que se incluya en la instantánea. Recordarás que cuando antes ejecutaste **git init**, seguidamente ejecutaste **git add (archivos)**. Esto era para iniciar el seguimiento de los archivos de tu directorio. El comando git add recibe la ruta de un archivo o de un directorio; si es un directorio, añade todos los archivos que contenga de manera recursiva.

Preparando archivos modificados

Vamos a modificar un archivo que estuviese bajo seguimiento. Si modificas el archivo `index.html` que estaba bajo seguimiento, y ejecutas el comando `git status` de nuevo, verás algo así:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   index.html
#
```

El archivo `index.html` aparece bajo la cabecera “Modificados pero no actualizados” (“Changes not staged for commit”). Esto significa que un archivo bajo seguimiento ha sido modificado en el directorio de trabajo, pero no ha sido preparado todavía. Para prepararlo, ejecuta el comando `git add` (es un comando multiuso ya que puedes utilizarlo para empezar el seguimiento de archivos nuevos, para preparar archivos, y para otras cosas como marcar como resueltos archivos con conflictos de unión). Ejecutamos `git add` para preparar el archivo `index.html`, y volvemos a ejecutar `git status`:

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

Ambos archivos están ahora preparados y se incluirán en tu próxima confirmación. Asume que en este momento recuerdas que tenías que hacer una pequeña modificación en `index.html` antes de confirmarlo. Lo vuelves abrir, haces ese pequeño cambio, y ya estás listo para confirmar. Sin embargo, si vuelves a ejecutar `git status` verás lo siguiente:

```
$ nano index.html
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   index.html
#
```

Ahora index.html aparece listado como preparado y como no preparado. ¿Cómo es esto posible? Resulta que Git prepara un archivo tal y como era en el momento de ejecutar el comando `git add`. Si haces `git commit` ahora, la versión de index.html que se incluirá en la confirmación será la que fuese cuando ejecutaste el comando `git add`, no la versión que estás viendo ahora en tu directorio de trabajo. Si modificas un archivo después de haber ejecutado `git add`, tendrás que volver a ejecutar `git add` para preparar la última versión del archivo:

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

Ignorar archivos

A menudo tendrás un tipo de archivos que no quieras que Git añada automáticamente o te muestre como no versionado. Suelen ser archivos generados automáticamente, como archivos de log, o archivos generados por tu compilador, instaladores, etc. Para estos casos puedes crear un archivo llamado `.gitignore`, en el que listas los patrones de nombres que desees que sean ignorados. A continuación un archivo `.gitignore` de ejemplo:

```
$ nano .gitignore
*.apk
*~
* /dist
```

La primera línea le dice a Git que ignore cualquier archivo cuyo nombre termine en `.apk` que son archivos binarios resultado de la compilación de código. La segunda línea le dice a Git que ignore todos los archivos que terminan en tilde (`~`), usada por muchos editores de texto, como Emacs, para marcar archivos temporales. La tercera línea dice que se ignore todo lo que se encuentre dentro del directorio `/dist` que generalmente contiene archivos resultado de una compilación de código. También puedes incluir directorios de log, temporales, documentación generada automáticamente, etc. Configurar un archivo `.gitignore` antes de empezar a trabajar suele ser una buena idea, para así no confirmar archivos que no quieres en tu repositorio Git.

Las reglas para los patrones que pueden ser incluidos en el archivo `.gitignore` son:

- Las líneas en blanco, o que comienzan por `#`, son ignoradas.

- Puedes usar patrones glob estándar.

- Puedes indicar un directorio añadiendo una barra hacia delante (`/`) al final.

- Puedes negar un patrón añadiendo una exclamación (`!`) al principio.

Los patrones glob son expresiones regulares simplificadas que pueden ser usadas por las shells. Un asterisco (`*`) reconoce cero o más caracteres; `[abc]` reconoce cualquier carácter de los especificados entre

corchetes (en este caso, a, b o c); una interrogación (?) reconoce un único carácter; y caracteres entre corchetes separados por un guión ([0-9]) reconoce cualquier carácter entre ellos (en este caso, de 0 a 9).

He aquí otro ejemplo de archivo .gitignore:

```
# a comment - this is ignored
# no .a files
*.a
# but do track lib.a, even though you're ignoring .a files above
!lib.a
# only ignore the root TODO file, not subdir/TODO
/TODO
# ignore all files in the build/ directory
build/
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
# ignore all .txt files in the doc/ directory
doc/**/*.txt
```

El patrón */ está disponible en Git desde la versión 1.8.2.

Ver cambios preparados y no preparados

Si el comando **git status** es demasiado impreciso para ti y quieres saber exactamente lo que ha cambiado, no sólo qué archivos fueron modificados, entonces puedes usar el comando **git diff**. Veremos **git diff** en más detalle después; pero probablemente lo usarás para responder estas dos preguntas:

¿Qué has cambiado pero aún no has preparado?, y

¿Qué has preparado y estás a punto de confirmar?

Aunque **git status** responde esas preguntas de manera general, **git diff** te muestra exactamente las líneas añadidas y eliminadas.

Supongamos que quieres editar y preparar el archivo README otra vez, y luego editar el archivo index.html sin prepararlo. Si ejecutas el comando **git status**, de nuevo verás algo así:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   index.html
#
```

Para ver lo que has modificado pero aún no has preparado, escribe **git diff**:

```
$ git diff
diff --git a/index.html b/index.html
index 1bf4c61..e81e094 100644
--- a/index.html
+++ b/index.html
```

```
@@ -3,6 +3,7 @@
    <title>Mi primera p<C3><A1>gina</title>
  </head>
  <body>
+    <h1>T<C3><AD>tulo de la p<C3><A1>gina</h1>
    <p>Ejemplo de un p<C3><A1>rrafo</p>
  </body>
</html>
```

git diff compara lo que hay en tu directorio de trabajo con lo que hay en tu área de preparación. El resultado te indica los cambios que has hecho y que todavía no has preparado.

Si quieres ver los cambios que has preparado y que irán en tu próxima confirmación, puedes usar **git diff --staged**. Este comando compara tus cambios preparados con tu última confirmación:

```
$ git diff --staged
diff --git a/index.html b/index.html
index 1bf4c61..4472f29 100644
--- a/index.html
+++ b/index.html
@@ -3,6 +3,8 @@
    <title>Mi primera p<C3><A1>gina</title>
  </head>
  <body>
+    <h1>T<C3><AD>tulo de la p<C3><A1>gina</h1>
+    <h2>Subt<C3><AD>tulo de la p<C3><A1>gina</h2>
    <p>Ejemplo de un p<C3><A1>rrafo</p>
  </body>
</html>
```

Es importante indicar que **git diff** por sí solo no muestra todos los cambios hechos desde tu última confirmación, sólo muestra los cambios que todavía no están preparados. Esto puede resultar desconcertante, porque si has preparado todos tus cambios, **git diff** no mostrará nada.

Confirmar cambios

Ahora que el área de preparación está como tú quieres, puedes confirmar los cambios. Recuerda que cualquier cosa que todavía esté sin preparar, cualquier archivo que hayas creado o modificado, y sobre el que no hayas ejecutado **git add** desde su última edición no se incluirá en esta confirmación. Se mantendrán como modificados en tu disco.

En este caso, la última vez que ejecutaste **git status** viste que estaba todo preparado, por lo que estás listo para confirmar tus cambios. La forma más fácil de confirmar es escribiendo **git commit**:

```
$ git commit
```

Al hacerlo, se ejecutará tu editor de texto por defecto que configuraste. El editor mostrará el siguiente texto (este ejemplo usa nano):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
```



```
# Changes to be committed:
#       modified:   index.html
#
```

Puedes ver que el mensaje de confirmación predeterminado contiene la salida del comando **git status** comentada, y una línea vacía arriba del todo. Puedes eliminar estos comentarios y escribir tu mensaje de confirmación, o puedes dejarlos para ayudarte a recordar las modificaciones que estás confirmando. (Para un recordatorio todavía más explícito de lo que has modificado, puedes pasar la opción **-v** a git commit. Esto provoca que se agreguen también las diferencias de tus cambios, para que veas exactamente lo que hiciste.) Cuando sales del editor, Git crea tu confirmación con el mensaje que hayas especificado (omitiendo los comentarios y las diferencias).

Como alternativa, puedes escribir tu mensaje de confirmación desde la propia línea de comandos mediante la opción **-m**:

```
$ git commit -m "git commit -m "se ha añadido las etiquetas html para indicar un
título y subtítulo"
[master 28fdc35] se ha añadido las etiquetas html para indicar un título y
subtítulo
1 file changed, 2 insertions(+)
```

¡Listo acabas de crear tu primera confirmación (commit como normalmente lo escucharas)! Puedes ver que el comando **commit** ha dado cierta información sobre la confirmación: a qué rama has confirmado (master), cuál es su suma de comprobación SHA-1 de la confirmación (**28fdc35**), cuántos archivos se modificaron, y estadísticas acerca de cuántas líneas se han añadido y cuántas se han eliminado.

Recuerda que la confirmación registra la instantánea de tu área de preparación. Cualquier cosa que no preparases sigue estando modificada; puedes hacer otra confirmación para añadirla a la historia del proyecto. Cada vez que confirmas, estás registrando una instantánea de tu proyecto, a la que puedes volver o con la que puedes comparar más adelante.

Evitar el área de preparación

Aunque puede ser extremadamente útil para elaborar confirmaciones exactamente a tu gusto, el área de preparación es en ocasiones demasiado compleja para las necesidades de tu flujo de trabajo. Si quieres saltarte el área de preparación, Git proporciona un atajo. Pasar la opción **-a** al comando **git commit** hace que Git prepare todo archivo que estuviese en seguimiento antes de la confirmación, permitiéndote obviar toda la parte de git add:

```
$ git status
# On branch master
#
# Changes not staged for commit:
#
#       modified:   index.html
#
$ git commit -a -m "se ha añadido un 2do párrafo"
[[master f04cf6a] se ha añadido un 2do párrafo
1 file changed, 1 insertion(+)
```

Fíjate que no has tenido que ejecutar **git add** sobre el archivo `index.html` antes de hacer la confirmación.

Eliminar archivos

Para eliminar un archivo de Git, debes eliminarlo de tus archivos bajo seguimiento (más concretamente, debes eliminarlo de tu área de preparación), y después confirmar. El comando **git rm** se encarga de eso, y también elimina el archivo de tu directorio de trabajo, para que no lo veas entre los archivos sin seguimiento.

Si simplemente eliminas el archivo de tu directorio de trabajo, aparecerá bajo la cabecera “Modificados pero no actualizados” (“Changes not staged for commit”) (es decir, *sin preparar*) de la salida del comando **git status**:

```
$ rm test.html
$ git status
# On branch master
#
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#
#       deleted:    test.html
#
```

Si entonces ejecutas el comando **git rm**, preparas la eliminación del archivo en cuestión:

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

La próxima vez que confirmes, el archivo desaparecerá y dejará de estar bajo seguimiento. Si ya habías modificado el archivo y lo tenías en el área de preparación, deberás forzar su eliminación con la opción **-f**. Ésta es una medida de seguridad para evitar la eliminación accidental de información que no ha sido registrada en una instantánea, y que por tanto no podría ser recuperada.

Otra cosa que puede que quieras hacer es mantener el archivo en tu directorio de trabajo, pero eliminarlo de tu área de preparación. Dicho de otro modo, puede que quieras mantener el archivo en tu disco duro, pero interrumpir su seguimiento por parte de Git. Esto resulta particularmente útil cuando olvidaste añadir algo a tu archivo `.gitignore` y lo añadiste accidentalmente, como por ejemplo un archivo de log enorme. Para hacer esto, usa la opción **--cached**:

```
$ git rm --cached readme.txt
```

El comando **git rm** acepta archivos, directorios, y patrones glob. Es decir, que podrías hacer algo así:

```
$ git rm log/*.log
```

Fíjate en la barra hacia atrás (`\`) antes del `*`. Es necesaria debido a que Git hace su propia expansión de rutas, además de la expansión que hace tu shell. En la consola del sistema de Windows, esta barra debe de ser omitida. Este comando elimina todos los archivos con la extensión `.log` en el directorio `log/`. También puedes hacer algo así:

```
$ git rm \*~
```

Este comando elimina todos los archivos que terminan en `~`.

Historial de confirmaciones

Después de haber hecho varias confirmaciones, o si has clonado un repositorio que ya tenía un histórico de confirmaciones, probablemente quieras mirar atrás para ver qué modificaciones se han llevado a cabo. La herramienta más básica y potente para hacer esto es el comando `git log`.

Vamos a ver un ejemplo de como usar `git log` clonando el siguiente proyecto y para ello ejecuta:

```
git clone git@gitlab.com:rollingcodeschool/fullstack-modulo1-git.git
```

Cuando ejecutes `git log` sobre este proyecto, deberías ver una salida similar a esta:

```
$ git log
commit 73bca181cf568c816568bba2d3a56a17ed22188c (HEAD -> master)
Merge: 0c9832a 65ec7bc
Author: Johnny Guzman <guzmanjhonny@gmail.com>
Date: Mon Apr 1 15:33:44 2019 -0300

    Merge branch 'master' of gitlab.com:rollingcodeschool/fullstack-modulo1-git

commit 0c9832aeec50ae7ea27010883ebc3baf12e9f3c1
Merge: b899e88 8ce25b6
Author: Johnny Guzman <guzmanjhonny@gmail.com>
Date: Mon Apr 1 15:29:53 2019 -0300

    se soluciono conflictos en merge

commit 65ec7bc71fcb56a3a8654079a8380b5aa2dc4781 (origin/master)
Author: francof2842 <francof2842@gmail.com>
Date: Mon Apr 1 12:18:14 2019 -0300

    Agrego clase de HTML - CSS - Box Model

commit b899e88bf24cf2959e6eb4df088c323c9d3a6312
Author: Johnny Guzman <guzmanjhonny@gmail.com>
Date: Mon Apr 1 11:42:01 2019 -0300

    se añade nuevo parrafo dentro de un contenedor
```

Por defecto, si no pasas ningún argumento, `git log` lista las confirmaciones hechas sobre ese repositorio en orden cronológico inverso. Es decir, las confirmaciones más recientes se muestran al principio. Como puedes ver, este comando lista cada confirmación con su suma de comprobación SHA-1, el nombre y dirección de correo del autor, la fecha y el mensaje de confirmación.

El comando **git log** proporciona gran cantidad de opciones para mostrarte exactamente lo que buscas. Aquí veremos algunas de las más usadas. Para ver todas las opciones puedes escribir **git log --help**.

Una de las opciones más útiles es **-p**, que muestra las diferencias introducidas en cada confirmación. También puedes usar la opción **-2**, que hace que se muestren únicamente las dos últimas entradas del histórico:

```
$ git log -p -2
commit 73bca181cf568c816568bba2d3a56a17ed22188c (HEAD -> master)
Merge: 0c9832a 65ec7bc
Author: Johnny Guzman <guzmanjhonny@gmail.com>
Date: Mon Apr 1 15:33:44 2019 -0300

    Merge branch 'master' of gitlab.com:rollingcodeschool/fullstack-modulo1-git

commit 0c9832aeec50ae7ea27010883ebc3baf12e9f3c1
Merge: b899e88 8ce25b6
Author: Johnny Guzman <guzmanjhonny@gmail.com>
Date: Mon Apr 1 15:29:53 2019 -0300

    se soluciono conflictos en merge
```

Esta opción muestra la misma información, pero añadiendo tras cada entrada las diferencias que le corresponden. Esto resulta muy útil para revisiones de código, o para visualizar rápidamente lo que ha pasado en las confirmaciones enviadas por un colaborador.

Deshacer cambios

En cualquier momento puedes querer deshacer algo. En esta sección veremos algunas herramientas básicas para deshacer cambios. Ten cuidado, porque no siempre puedes volver atrás después de algunas de estas operaciones. Ésta es una de las pocas áreas de Git que pueden provocar que pierdas datos si haces las cosas incorrectamente.

Modificando tu última confirmación

Uno de los casos más comunes en el que quieres deshacer cambios es cuando confirmas demasiado pronto y te olvidas de añadir algún archivo, o te confundes al introducir el mensaje de confirmación. Si quieres volver a hacer la confirmación, puedes ejecutar un **commit** con la opción **--amend**:

```
$ git commit --amend
```

Este comando utiliza lo que haya en tu área de preparación para la confirmación. Si no has hecho ningún cambio desde la última confirmación (por ejemplo, si ejecutas este comando justo después de tu confirmación anterior), esta instantánea será exactamente igual, y lo único que cambiarás será el mensaje de confirmación.

Se lanzará el editor de texto para que introduzcas tu mensaje, pero ya contendrá el mensaje de la confirmación anterior. Puedes editar el mensaje, igual que siempre, pero se sobrescribirá tu confirmación anterior.

Por ejemplo, si confirmas y luego te das cuenta de que se te olvidó preparar los cambios en uno de los archivos que querías añadir, puedes hacer algo así:

```
$ git commit -m 'initial commit'
$ git add forgotten file
$ git commit --amend
```

Estos tres comandos acabarán convirtiéndose en una única confirmación y la segunda confirmación reemplazará los resultados de la primera.

Deshaciendo la preparación de un archivo

Las dos secciones siguientes muestran cómo trabajar con las modificaciones del área de preparación y del directorio de trabajo. Lo bueno es que el comando que usas para determinar el estado de ambas áreas te recuerda cómo deshacer sus modificaciones. Por ejemplo, digamos que has modificado dos archivos, y quieres confirmarlos como cambios separados, pero tecleas accidentalmente `git add *` y preparas ambos. ¿Cómo puedes sacar uno de ellos del área de preparación? El comando `git status` te lo recuerda:

```
$ git add .
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#       modified:   index.html
#
```

Justo debajo de la cabecera “Cambios a confirmar” (“Changes to be committed”), dice que uses `git reset HEAD <archivo>...` para sacar un archivo del área de preparación. Vamos a aplicar ese consejo sobre `index.html`:

```
$ git reset HEAD index.html
index.html: locally modified
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
# 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:   index.html
#
```

El archivo `index.html` ahora está modificado, no preparado.

Deshaciendo la modificación de un archivo

¿Qué pasa si te das cuenta de que no quieres mantener las modificaciones que has hecho sobre el archivo `index.html`? ¿Cómo puedes deshacerlas fácilmente y revertir el archivo al mismo estado en el que estaba

cuando hiciste tu última confirmación (o cuando clonaste el repositorio, o como quiera que metieras el archivo en tu directorio de trabajo)? Afortunadamente, **git status** también te dice como hacer esto. En la salida del último ejemplo, la cosa estaba así:

```
# 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:   index.html
#
```

Te dice de forma bastante explícita cómo descartar las modificaciones que hayas hecho. Vamos a hacer lo que dice:

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

Puedes ver que se han revertido los cambios. También deberías ser consciente del peligro de este comando: cualquier modificación hecha sobre este archivo ha desaparecido acabas de sobrescribirlo con otro archivo. Nunca uses este comando a no ser que estés absolutamente seguro de que no quieres los cambios previamente aplicados sobre el archivo. Si lo único que necesitas es olvidarte de él momentáneamente, veremos los conceptos de apilamiento (stashing) y ramificación (branching) en la siguiente sección en general son formas más adecuadas de trabajar.

Recuerda, cualquier cosa que esté confirmada en Git casi siempre puede ser recuperada. Incluso confirmaciones sobre ramas que han sido eliminadas, o confirmaciones sobrescritas con la opción **--amend**, pueden recuperarse. Sin embargo, cualquier cosa que pierdas y que no estuviese confirmada, probablemente no vuelvas a verla nunca más.

Repositorios remotos

Para poder colaborar en cualquier proyecto Git, necesitas saber cómo gestionar tus repositorios remotos. Los repositorios remotos son versiones de tu proyecto que se encuentran alojados en Internet o en algún punto de la red. Puedes tener varios, cada uno de los cuales puede ser de sólo lectura, o de lectura/escritura, según los permisos que tengas. Colaborar con otros implica gestionar estos repositorios remotos, y mandar (push) y recibir (pull) datos de ellos cuando necesites compartir cosas. Existen en la actualidad algunos sistemas en línea que te permiten almacenar una copia de tu proyecto de git y de esa

forma puedes trabajar en forma colaborativa entre varias personas. Ejemplos de ellos son Gitbub, Gitlab, Bitbucket, etc. Hablaremos de ellos más adelante.

Gestionar repositorios remotos implica conocer cómo añadir repositorios nuevos, eliminar aquellos que ya no son válidos, gestionar ramas remotas e indicar si están bajo seguimiento o no, y más cosas. En esta sección abordaremos todos estos conceptos.

Ver tus repositorios remotos

Para ver qué repositorios remotos tienes configurados, puedes ejecutar el comando **git remote**. Mostrará una lista con los nombres de los remotos que hayas especificado. Si has clonado tu repositorio, deberías ver por lo menos "origin" que es el nombre predeterminado que le da Git al servidor del que clonaste:

```
$ git clone git@gitlab.com:rollingcodeschool/fullstack-modulo1-git.git
Cloning into 'fullstack-modulo1-git'...
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 17 (delta 3), reused 0 (delta 0)
Receiving objects: 100% (17/17), done.
Resolving deltas: 100% (3/3), done.

$ cd fullstack-modulo1-git/
$ git remote
origin
```

También puedes añadir la opción **-v**, que muestra la URL asociada a cada repositorio remoto:

```
$ git remote -v
origin      git@gitlab.com:rollingcodeschool/fullstack-modulo1-git.git (fetch)
origin      git@gitlab.com:rollingcodeschool/fullstack-modulo1-git.git (push)
```

Si tienes más de un remoto, este comando los lista todos.

Esto significa que podemos recibir contribuciones de cualquiera de estos usuarios de manera bastante fácil. Pero vemos que sólo el remoto origen tiene una URL SSH, por lo que es el único al que podemos enviar.

Agregar repositorios remotos

Ya he mencionado y he dado ejemplos de repositorios remotos en secciones anteriores, pero a continuación veremos cómo añadirlos explícitamente. Para añadir un nuevo repositorio Git remoto, asignándole un nombre con el que referenciar fácilmente, ejecuta **git remote add [nombre] [url]**:

```
$ git remote
origin
$ git remote add jhonny git@gitlab.com:guzmanjhonny/fullstack-modulo1-git.git

$ git remote -v
jhonny      git@gitlab.com:guzmanjhonny/fullstack-modulo1-git.git (fetch)
jhonny      git@gitlab.com:guzmanjhonny/fullstack-modulo1-git.git (push)
origin      git@gitlab.com:rollingcodeschool/fullstack-modulo1-git.git (fetch)
origin      git@gitlab.com:rollingcodeschool/fullstack-modulo1-git.git (push)
```

Ahora puedes usar la cadena "jhonny" en la línea de comandos, en lugar de toda la URL. Por ejemplo, si quieres recuperar toda la información de Jhonny que todavía no tienes en tu repositorio, puedes ejecutar **git fetch johnny**:

```
$ git fetch johnny
From gitlab.com:guzmanjhonny/fullstack-modulol-git
* [new branch]      master    -> johnny/master
```

La rama maestra de johnny es accesible localmente como **johnny/master** y puedes unirla a una de tus ramas, o copiarla localmente para inspeccionarla.

Recibir datos de tus repositorios remotos

Como acabas de ver, para recuperar datos de tus repositorios remotos puedes ejecutar:

```
$ git fetch [remote-name]
```

Este comando recupera todos los datos del proyecto remoto que no tengas todavía. Después de hacer esto, deberías tener referencias a todas las ramas del repositorio remoto, que puedes unir o inspeccionar en cualquier momento.

Si clonas un repositorio, el comando añade automáticamente ese repositorio remoto con el nombre de "origin". Por tanto, **git fetch origin** recupera toda la información enviada a ese servidor desde que lo clonaste (o desde la última vez que ejecutaste fetch). Es importante tener en cuenta que el comando **fetch** sólo recupera la información y la pone en tu repositorio local y no la une automáticamente con tu trabajo ni modifica aquello en lo que estás trabajando. Tendrás que unir ambos manualmente a posteriori.

Si has configurado una rama para seguir otra rama remota como veremos más adelante, puedes usar el comando **git pull** para recuperar y unir automáticamente la rama remota con tu rama actual. Éste puede resultar un flujo de trabajo más sencillo y más cómodo; y por defecto, el comando **git clone** automáticamente configura tu rama local maestra para que siga la rama remota maestra del servidor del cual clonaste (asumiendo que el repositorio remoto tiene una rama maestra). Al ejecutar **git pull**, por lo general se recupera la información del servidor del que clonaste, y automáticamente se intenta unir con el código con el que estás trabajando actualmente.

Enviar datos a tus repositorios remotos

Cuando tu proyecto se encuentra en un estado que quieres compartir, tienes que enviarlo a un repositorio remoto. El comando que te permite hacer esto es sencillo: **git push [nombre-remoto][nombre-rama]**. Si quieres enviar tu rama maestra (**master**) a tu servidor origen (**origin**), ejecutamos esto para enviar tu trabajo al servidor:

```
$ git push origin master
```

Este comando funciona únicamente si has clonado de un servidor en el que tienes permiso de escritura, y nadie ha enviado información mientras tanto. Si tú y otra persona haces la operación de clonado a la vez, y

él envía su información y luego envías tú la tuya, tu envío será rechazado. Tendrás que bajarte (pull) primero su trabajo e incorporarlo en el tuyo para que se te permita hacer un envío.

Revisar un repositorio remoto

Si quieres ver más información acerca de un repositorio remoto en particular, puedes usar el comando **git remote show [nombre]**. Si ejecutas este comando pasándole el nombre de un repositorio, como **origin**, obtienes algo así:

```
$ git remote show origin
* remote origin
  Fetch URL: git@gitlab.com:rollingcodeschool/fullstack-modulol-git.git
  Push  URL: git@gitlab.com:rollingcodeschool/fullstack-modulol-git.git
  HEAD branch: master
  Remote branch:
    master tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Esto lista la URL del repositorio remoto, así como información sobre las ramas bajo seguimiento. Este comando te recuerda que si estás en la rama maestra y ejecutas **git pull**, automáticamente unirá los cambios a la rama maestra del remoto después de haber recuperado todas las referencias remotas. También lista todas las referencias remotas que ha recibido.

Eliminar y renombrar repositorios remotos

Si quieres renombrar una referencia a un repositorio remoto, en versiones recientes de Git puedes ejecutar **git remote rename**. Por ejemplo, si quieres renombrar **jhonny** a **jhonnybcode**, puedes hacerlo de la siguiente manera:

```
$ git remote rename jhonny jhonnybcode
$ git remote
origin
jhonnybcode
```

Conviene mencionar que esto también cambia el nombre de tus ramas remotas. Lo que antes era referenciado en **jhonny/master** ahora está en **jhonnybcode/master**.

Si por algún motivo quieres eliminar una referencia o has movido el servidor o ya no estás usando un determinado mirror, o quizás un contribuidor ha dejado de contribuir, puedes usar el comando **git remote rm**:

```
$ git remote rm jhonny
$ git remote
origin
```

Ramas o branch en Git

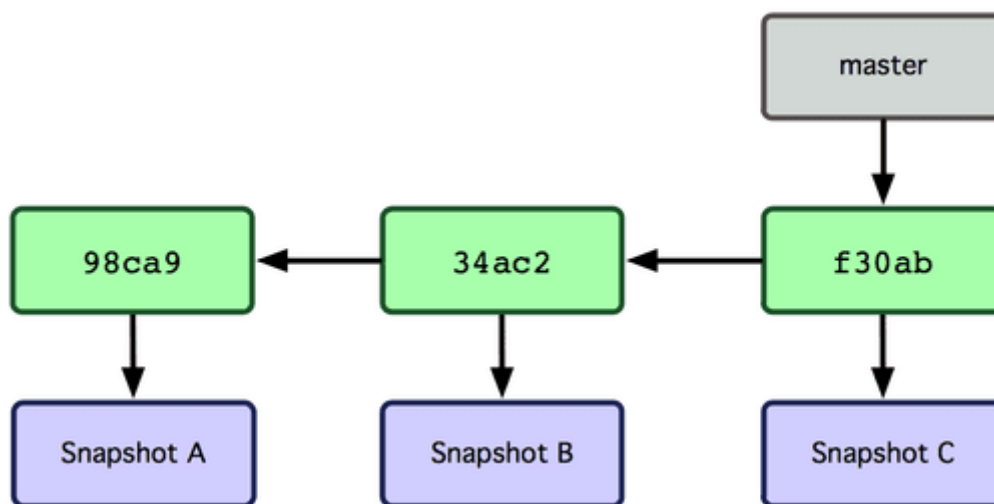
En el día a día del trabajo, una de las cosas útiles que podemos hacer es trabajar con ramas. Las ramas son caminos que puede tomar el desarrollo de un software, algo que ocurre naturalmente para resolver problemas o crear nuevas funcionalidades. En la práctica permiten que nuestro proyecto pueda tener diversos estados y que los desarrolladores sean capaces de pasar de uno a otro de una manera ágil.

Cualquier sistema de control de versiones actual tiene algún mecanismo para soportar distintos ramales. Cuando hablamos de ramificaciones, significa que tu has tomado la rama principal de desarrollo (master) y a partir de ahí has continuado trabajando sin seguir la rama principal de desarrollo.

La forma en la que Git maneja las ramificaciones es increíblemente rápida, haciendo así de las operaciones de ramificación algo casi instantáneo, al igual que el avance o el retroceso entre distintas ramas, lo cual también es tremendamente rápido. A diferencia de otros sistemas de control de versiones, Git promueve un ciclo de desarrollo donde las ramas se crean y se unen ramas entre sí, incluso varias veces en el mismo día. Entender y manejar esta opción te proporciona una poderosa y exclusiva herramienta que puede, literalmente, cambiar la forma en la que desarrollas.

Qué son las ramas

Una rama Git es simplemente un apuntador móvil apuntando a una confirmación (**commit**) en particular. La rama por defecto de Git es la rama **master**. Con la primera confirmación de cambios que realicemos, se creará esta rama principal master apuntando a dicha confirmación. En cada confirmación de cambios que realicemos, la rama irá avanzando automáticamente. Y la rama master apuntará siempre a la última confirmación realizada.

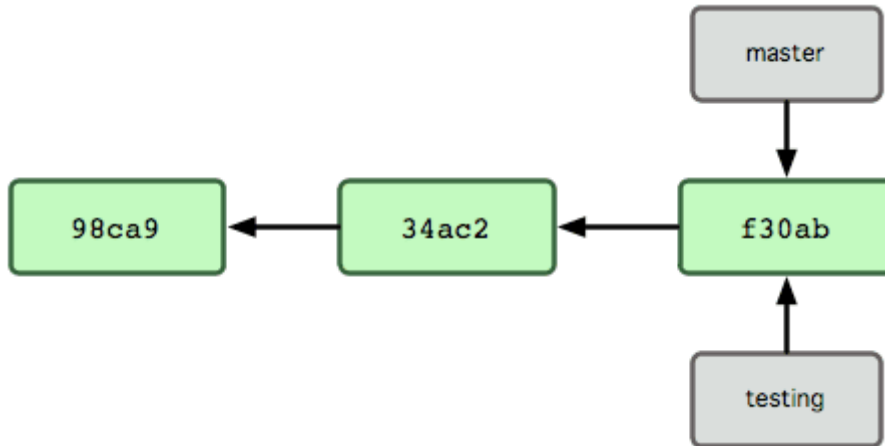


Apuntadores en el registro de confirmaciones de una rama.

¿Qué sucede cuando creas una nueva rama? Simplemente se crea un nuevo apuntador para que lo puedas mover libremente. Por ejemplo, si quieres crear una nueva rama denominada "testing". Usarás el comando git branch:

```
$ git branch testing
```

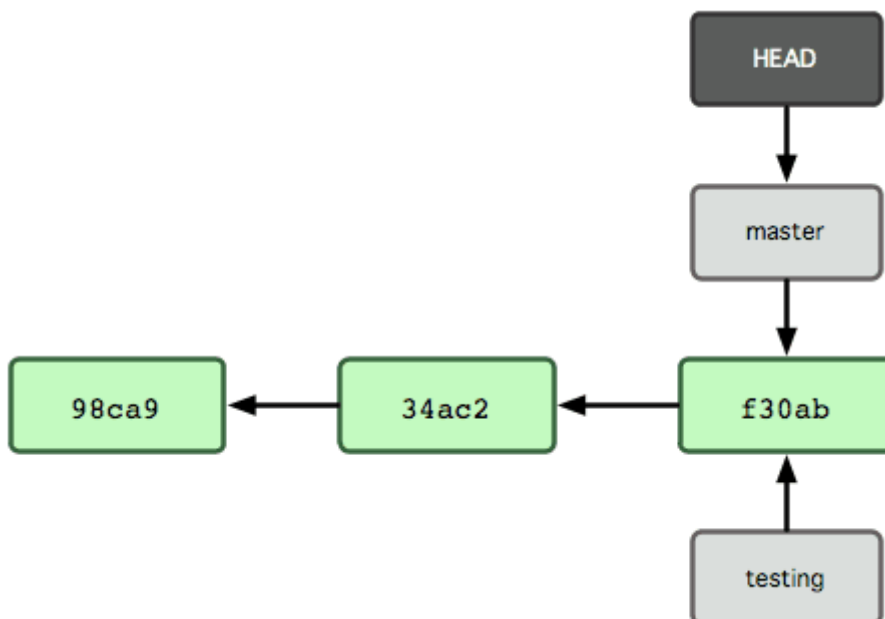
Esto creará un nuevo apuntador apuntando a la misma confirmación donde estés actualmente.



Apuntadores de varias ramas en el registro de confirmaciones de cambio.

HEAD ¿cómo sabe Git en qué rama estás en este momento?

Existe un apuntador especial denominado HEAD que determina en qué rama nos encontramos en un determinado momento. HEAD es simplemente el apuntador a la rama local en la que tú estés en ese momento. En este caso, en la rama master. Puesto que el comando `git branch` solamente crea una nueva rama, y no salta a dicha rama.

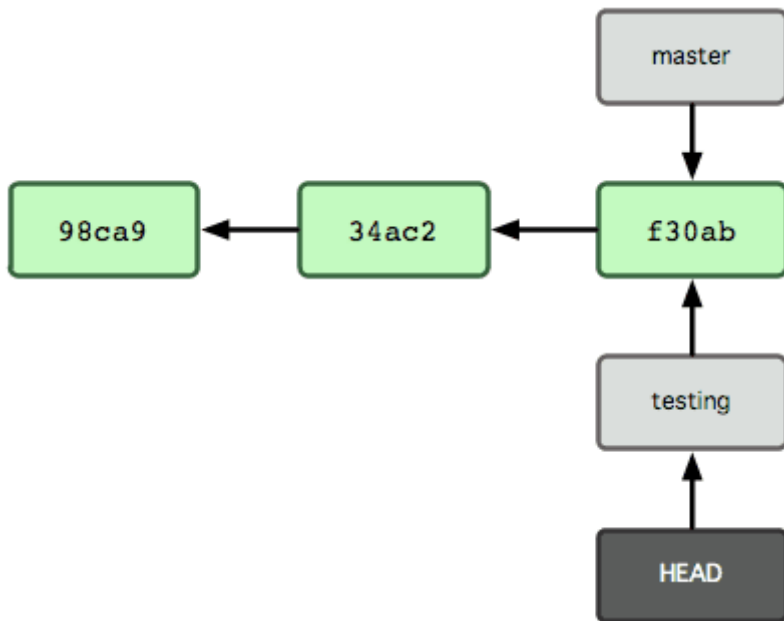


Apuntador HEAD a la rama donde estás actualmente.

Para saltar de una rama a otra, tienes que utilizar el comando `git checkout`. Por ejemplo vamos a pasarnos a la rama `testing` recién creada:

```
$ git checkout testing
```

Esto mueve el apuntador HEAD a la rama `testing`.

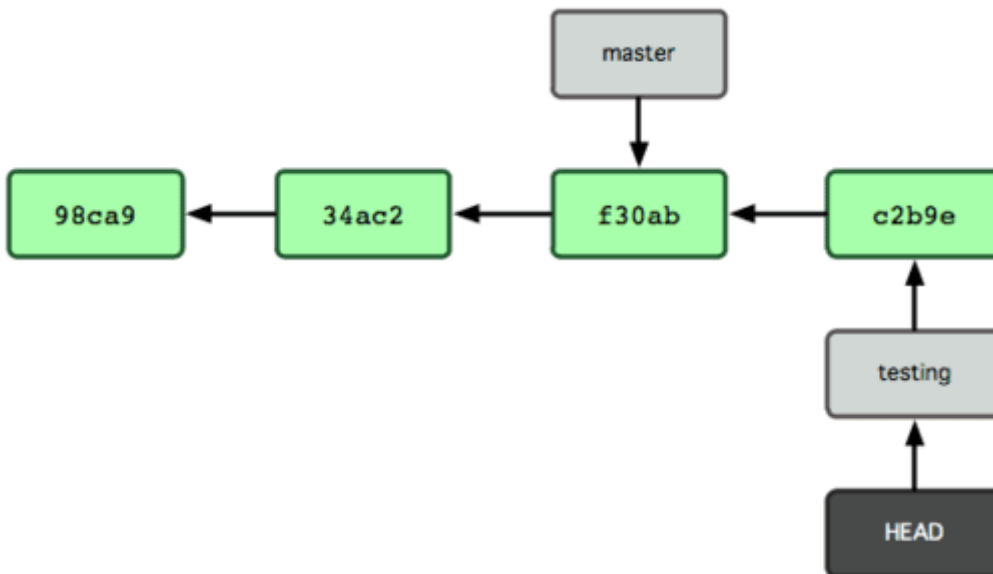


Apuntador HEAD apuntando a otra rama cuando saltamos de rama.

¿Cuál es el significado de todo esto?. Bueno... lo veremos tras realizar otra confirmación de cambios:

```
$ nano index.html  
$ git commit -a -m 'haciendo un cambio'
```

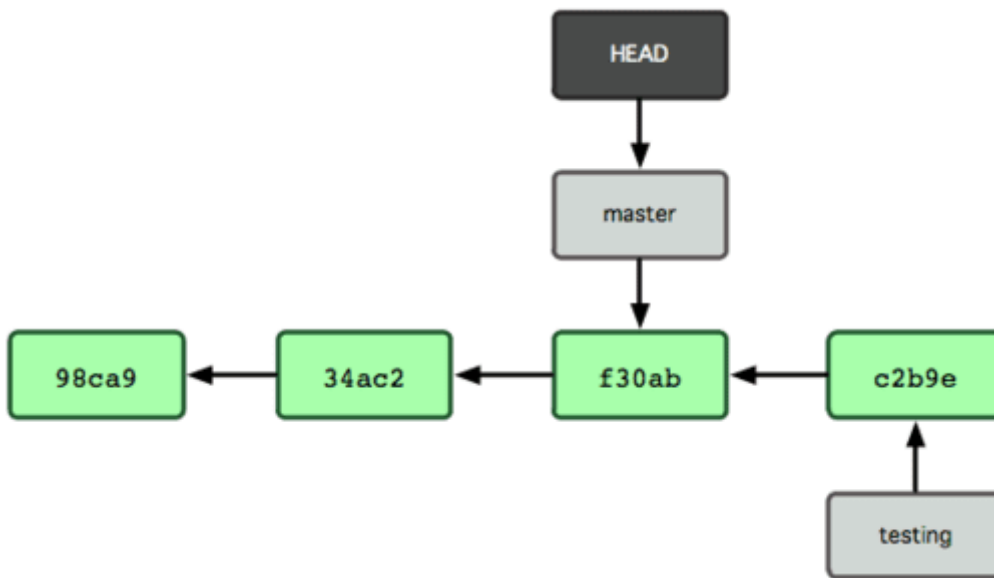
El resultado:



La rama apuntada por HEAD avanza con cada confirmación de cambios.

Observamos algo interesante: la rama **testing** avanza, mientras que la rama **master** permanece en la confirmación donde estaba cuando lanzaste el comando **git checkout** para saltar. Volvamos ahora a la rama **master**:

```
$ git checkout master
```



HEAD apunta a otra rama cuando hacemos un checkout.

Este comando realiza dos acciones: Mueve el apuntador HEAD de nuevo a la rama **master**, y revierte los archivos de tu directorio de trabajo; dejándolos tal y como estaban en la última instantánea confirmada en dicha rama master. Esto supone que los cambios que hagas desde este momento en adelante no van a coincidir del todo de la antigua versión del proyecto. Básicamente, lo que se está haciendo es rebobinar el trabajo que habías hecho temporalmente en la rama **testing**; de tal forma que puedas avanzar en otra dirección diferente.

Si deseas directamente crear un rama y saltarte a la misma en una sola operación pueden usar el siguiente comando: **git checkout -b testing**. El flag **-b** directamente te permite crear la rama que en este caso sería **testing**.

Fusionar ramas

A medida que creas ramas y cambies el estado de las carpetas o archivos de tu proyecto empezará a divergir de una rama a otra. Llegará el momento en el que te interese fusionar ramas para poder incorporar el trabajo realizado a la rama master.

El proceso de fusión se conoce como "**merge**" y puede llegar a ser muy simple o más complejo si se encuentran cambios que Git no pueda procesar de manera automática. Git para procesar los merge usa un antecesor común y comprueba los cambios que se han introducido al proyecto desde entonces, combinando el código de ambas ramas.

Para hacer un merge nos situamos en una rama, en este caso la "master", y decimos con qué otra rama se debe fusionar el código.

El siguiente comando, lanzado desde la rama "master", permite fusionarla con la rama "develop".

```
$ git merge develop
```

Un merge necesita un mensaje, igual que ocurre con las confirmaciones, por lo que al realizar ese comando se abrirá tu editor de código por defecto en la consola para que introduzcas los comentarios que consideres oportuno. Esta acción de indicar el mensaje se puede resumir con el comando:

```
$ git merge develop -m 'Fusionando los cambios de develop a master'
```

En el siguiente fragmento de código puedes ver una secuencia de comandos y su salida. Primero el cambio a la rama master "git checkout master", luego el "git branch" para confirmar en qué rama nos encontramos y por último el merge para fusionarla con la rama develop.

```
$ git checkout master
Switched to branch 'master'
$ git branch
develop
* master
testing
$ git merge develop
Updating 4c6fd83..ecdd97c
Fast-forward
 index.html | 3 +++
 1 file changed, 3 insertions(+)
```

Luego podremos comprobar que nuestra rama master tiene todo el código nuevo de la rama experimental y podremos hacer nuevos commits en master para seguir el desarrollo de nuestro proyecto ya con la rama principal, si es nuestro deseo.

Resolviendo conflictos

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 pretendas fusionar, Git no será capaz de fusionarlas directamente. Por ejemplo si hemos modificado la misma línea estando en la rama master y luego en develop, obtendríamos un conflicto como este al hacer el merge:

```
$ git merge develop
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Al encontrar un conflicto Git hace una pausa en el proceso, esperando a que resuelvas el mismo. Para ver qué archivos permanecen sin fusionar en un determinado momento conflictivo de una fusión, puedes usar el comando **git status**:

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

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:   index.html

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

Todo aquellos archivos que tengan conflicto y no se hayan podido resolver, se marca como "sin fusionar" (unmerged).

Marcadores de conflicto

Git añade a los archivos conflictivos unos marcadores especiales de resolución de conflictos. Estos marcadores te ayudarán a resolver los conflictos dentro del archivo cuando vayas a editarlos manualmente. Un conflicto dentro del archivo se ve de la siguiente manera:

```
<<<<<<< HEAD:index.html
<p>prueba</p>
=====

>>>>>>> develop:index.html
```

Donde nos dice que la versión en HEAD (la rama master, la que habías activado antes de lanzar el comando de fusión), contiene lo indicado en la parte superior del bloque (todo lo que está encima de =====). Y que la versión en la rama develop contiene el resto, lo indicado en la parte inferior del bloque. Para resolver el conflicto, has de elegir manualmente contenido de uno o de otro lado. Por ejemplo, puedes optar por cambiar el bloque, dejándolo tal que:

```
<p>prueba</p>
```

Y recordar que debes eliminar completamente las líneas <<<<<<< , ===== y >>>>>>> . Cuando en un proyecto intervienen muchos desarrolladores, la resolución de conflictos se debería realizar con la persona que edito el archivo justo en la misma línea donde tu la editaste, de esta forma se evitarán posibles bugs que se puedan generar. Tras resolver todos los bloques conflictivos, has de lanzar los siguientes comandos:

git add para marcar cada archivo modificado.

git commit para marcar cada archivo como preparados (staging), indica a Git que sus conflictos han sido resueltos.

Por último para eliminar una rama usa el comando: **git branch -D nombre_de_rama**

GitFlow o Flujos de trabajo ramificados

Flujos de trabajo ramificados

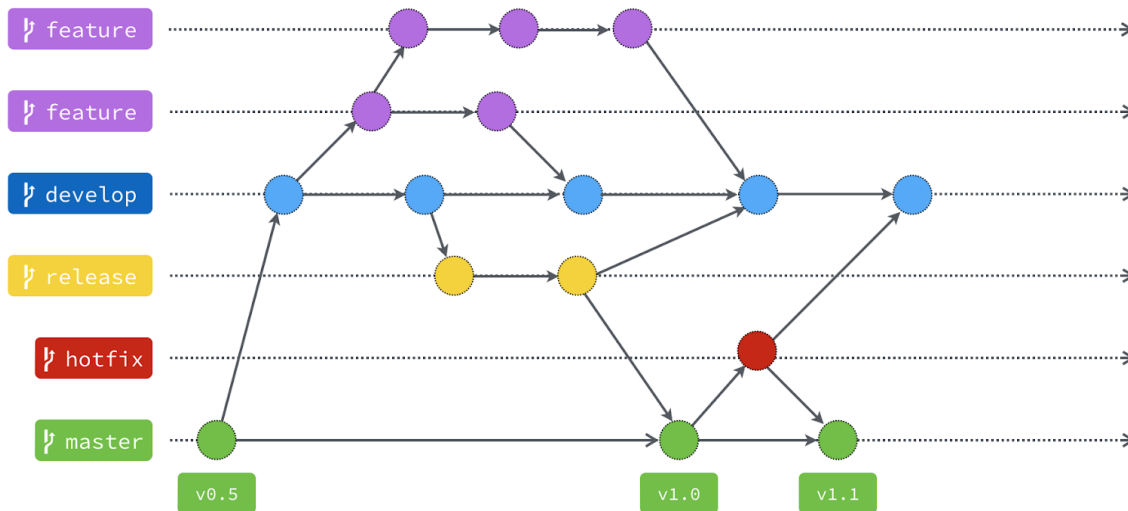
Ahora que ya has visto los procedimientos básicos de ramificación y fusión, veamos ¿qué puedes o qué debes hacer con ellos? A continuación vamos a ver algunos de los flujos de trabajo más comunes, de tal forma que puedas decidir si te gustaría incorporar alguno de ellos a tu ciclo de desarrollo.

Ramas de largo recorrido

Muchos desarrolladores que usan Git llevan un flujo de trabajo manteniendo en la rama **master** únicamente el código totalmente estable (el código que ha sido o que va a ser liberado para producción). Teniendo otras

ramas paralelas denominadas **develop** o **release**, en las que trabajan y realizan pruebas. Estas ramas paralelas no suele estar siempre en un estado estable; pero cada vez que sí lo están, pueden ser fusionadas con la rama master. También es habitual el incorporarle (pull) ramas puntuales (ramas temporales).

En realidad, en todo momento estamos hablando simplemente de apuntadores moviéndose por la línea temporal de confirmaciones de cambio (commit history). Las ramas estables apuntan hacia posiciones más antiguas en el registro de confirmaciones. Mientras que las ramas avanzadas, las que van abriendo camino, apuntan hacia posiciones más recientes.



Las ramas más estables apuntan hacia posiciones más antiguas en el registro de cambio

master: es la rama principal de todo proyecto git. representa una versión estable de nuestro proyecto.

develop: es una rama de desarrollo que contiene el estado más actual del mismo. Es la rama de donde deben partir cualquier desarrollo nuevo o resolución de algún problema.

feature: representa aquellas ramas que contienen nuevas funcionalidades, soluciones de errores o mejoras de funcionalidades pre-existentes. Una vez el desarrollo ha concluido, estas ramas deben ser fusionadas a develop para que esta última rama se encuentre totalmente actualizada.

release: esta rama contiene una versión del proyecto casi estable, donde a partir del cual pueden generarse instaladores para que nuestros clientes puedan ir probando las nuevas funcionalidades o correcciones realizadas.

hotfix: esta rama representa cambios puntuales de errores que no han sido visualizados y que persisten en la rama master. Son errores que deben ser solucionados rápidamente y posteriormente los mismos pueden ser fusionados al mismo branch master y develop respectivamente. En la siguiente sección veremos mas en detalle este tipos de ramas.

Este sistema de trabajo se puede ampliar para diversos grados de estabilidad. La idea es mantener siempre diversas ramas en diversos grados de estabilidad; pero cuando alguna alcanza un estado más estable, la fusionamos con la rama inmediatamente superior a ella. Aunque no es obligatorio el trabajar con ramas de larga duración, realmente es práctico y útil y sobre todo en proyectos que son muy largos o complejos.

Ramas puntuales

Las ramas puntuales, en cambio, son útiles en proyectos de cualquier tamaño. Una rama puntual es aquella de corta duración que abres para un tema o para una funcionalidad muy concretos.

Esta técnica te posibilita realizar rápidos y completos saltos de contexto. Puedes mantener los cambios ahí durante minutos, días o meses; y fusionarlos cuando realmente estén listos. En lugar de verte obligado a fusionarlos en el orden en que fueron creados y comenzaste a trabajar en ellos. Ejemplo de estos tipo de ramas son aquellas que están destinadas a solucionar un problema puntual (bug) o realizar alguna mejora (refactorización). Una buena práctica es tratar de establecer una forma organizada de nombrar a este tipo de ramas. Por ejemplo para casos de solución de incidencias o nuevas funcionalidades podemos nombrar las ramas usando las siguiente nomenclatura:

fix: fixed (resolución de una incidencia o comúnmente llamado bug)

ref: refactoring (refactorización o mejora de una funcionalidad)

feat: feature (nueva característica y/o funcionalidad)

Ejemplos:

`fix/loginUser // indica que es una solución de un problema y está asociado al login de un usuario`

`fix/passwordRecovery // indica que es una solución de un problema y está asociado a la funcionalidad de restablecimiento de contraseña.`

`ref/createUser // indica que es un refactorización o mejora de la funcionalidad de creación de usuarios`

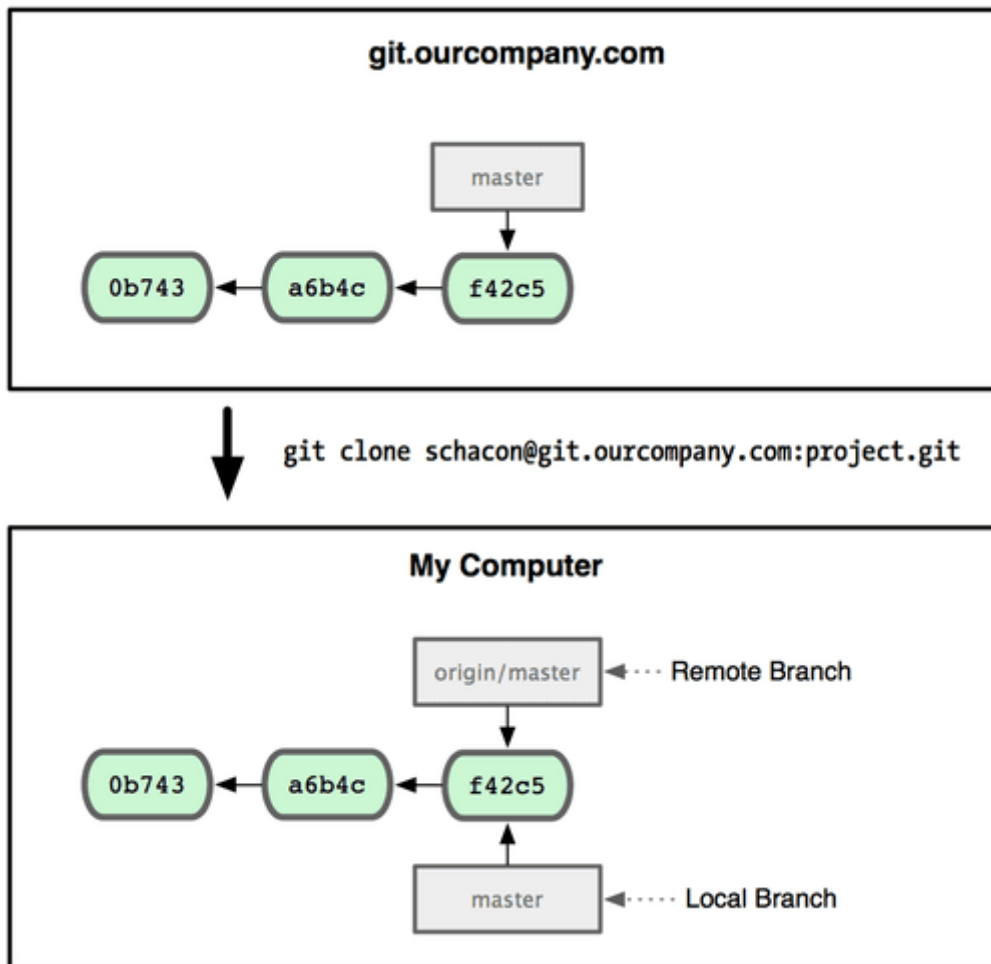
`feat/login // indica que es una característica nueva que se ha agregado`

Ramas Remotas

Las ramas remotas son referencias al estado de ramas en tus repositorios remotos. Son ramas locales que no puedes mover; se mueven automáticamente cuando estableces comunicaciones en la red. Las ramas remotas funcionan como marcadores, para recordarte en qué estado se encontraban tus repositorios remotos la última vez que te vinculaste con ellos.

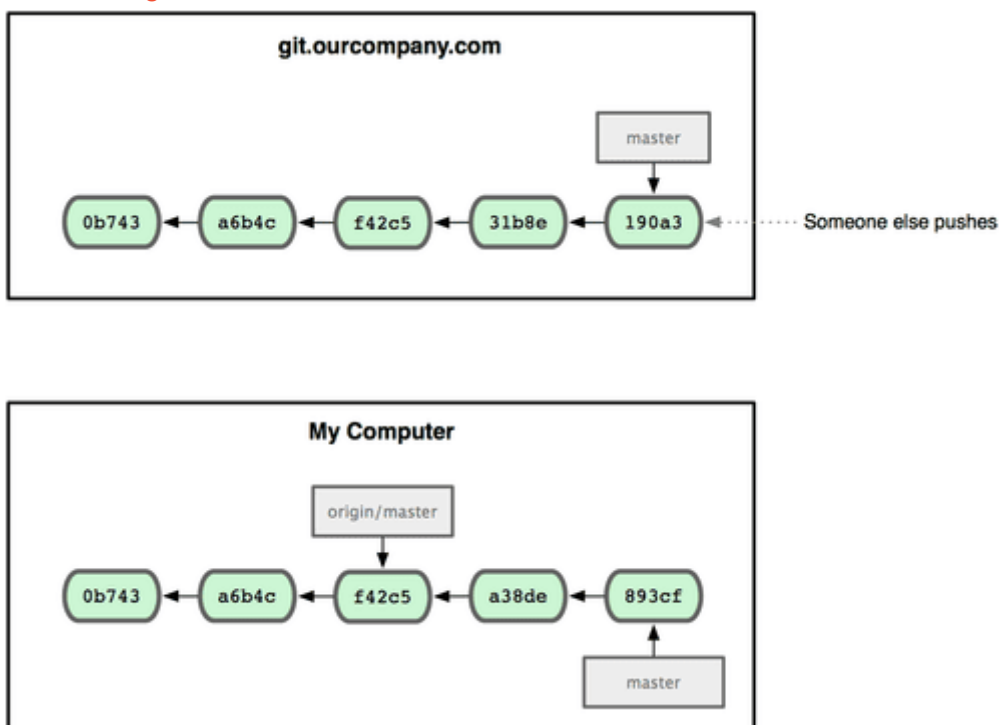
Suelen referenciarse como **(remoto)/(rama)**. Por ejemplo, si quieres saber cómo estaba la rama **master** en el remoto **origin**. Puedes revisar la rama **origin/master**. O si estás trabajando en un problema con un compañero y este envía (**push**) una rama **fix/loginUser**, tendrás tu propia rama de trabajo local **fix/loginUser**, pero la rama en el servidor apuntará a la última confirmación (**commit**) en la rama **origin/fix/loginUser**.

Supongamos que tienes un servidor Git en tu red, en `git.ourcompany.com`. Si haces un clon desde ahí, Git automáticamente lo denominará **origin**, traerá (**pull**) sus datos, creará un apuntador hacia donde esté en ese momento su rama **master**, denominará la copia local **origin/master**; y será inamovible para ti. Git te proporcionará también tu propia rama **master**, apuntando al mismo lugar que la rama **master** de **origin**; siendo en esta última donde podrás trabajar.



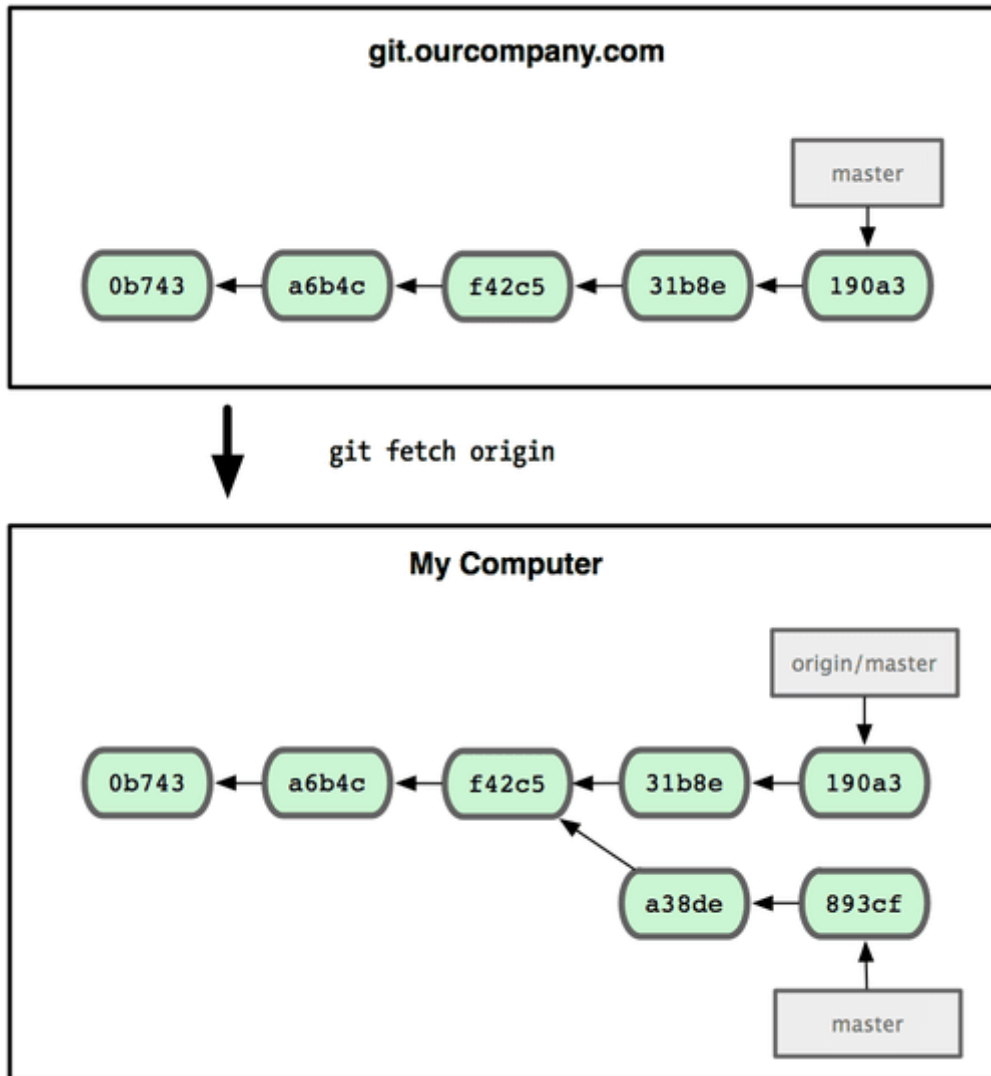
Un clon Git te proporciona tu propia rama **master** y otra rama **origin/master** apuntando a la rama master original.

Si haces algún trabajo en tu rama **master** local, y al mismo tiempo, alguna otra persona lleva (push) su trabajo al servidor **git.ourcompany.com**, actualizando la rama **master** de allí, te encontrarás con que ambos registros avanzan de forma diferente. Además, mientras no tengas contacto con el servidor, tu apuntador a tu rama **origin/master** no se moverá



Trabajando localmente y que otra persona esté llevando (push) algo al servidor remoto, hace que cada registro avance de forma distinta.

Para sincronizarte, puedes utilizar el comando `git fetch origin`. Este comando localiza en qué servidor está el origen (en este caso `git.ourcompany.com`), recupera cualquier dato presente allí que tu no tengas, y actualiza tu base de datos local, moviendo tu rama `origin/master` para que apunte a esta nueva y más reciente posición como se muestra a continuación.



El comando `git fetch` actualiza tus referencias remotas.

Publicar un branch al repositorio remoto

Cuando quieres compartir una rama con el resto del mundo, has de llevarla (push) a un repositorio remoto donde tengas permisos de escritura. Tus ramas locales no se sincronizan automáticamente con los remotos en los que escribes. Sino que tienes que llevar (push) expresamente, cada vez, al remoto las ramas que desees compartir. De esta forma, puedes usar ramas privadas para el trabajo que no desees compartir. Llevando a un remoto tan solo aquellas partes que desees aportar a los demás.

Si tienes una rama llamada `fix/loginUser`, con la que vas a trabajar en colaboración; puedes llevarla al remoto de la misma forma que llevaste tu primera rama. Con el comando `git push (remoto) (rama)`:

```
$ git push origin fix/loginUser
Counting objects: 20, done.
```

```
Compressing objects: 100% (14/14), done.  
Writing objects: 100% (15/15), 1.74 KiB, done.  
Total 15 (delta 5), reused 0 (delta 0)  
To git@github.com:schacon/simplegit.git  
* [new branch]      fix/loginUser -> fix/loginUser
```

Seguimiento a las ramas

Activando (checkout) una rama local a partir de una rama remota, se crea automáticamente lo que podríamos denominar "una rama de seguimiento" (tracking branch). Las ramas de seguimiento son ramas locales que tienen una relación directa con alguna rama remota. Si estás en una rama de seguimiento y tecleas el comando **git push**, Git sabe automáticamente a qué servidor y a qué rama ha de llevar los contenidos. Igualmente, tecleando **git pull** mientras estamos en una de esas ramas, recupera (**fetch**) todas las referencias remotas y las consolida (**merge**) automáticamente en la correspondiente rama remota.

Cuando clonas un repositorio, este suele crear automáticamente una rama master que hace seguimiento de **origin/master**. Y es por eso que **git push** y **git pull** trabajan directamente, sin necesidad de más argumentos. Sin embargo, puedes preparar otras ramas de seguimiento si deseas tener unas que no hagan seguimiento de ramas en **origin** y que no sigan a la rama **master**. El ejemplo más simple, es el que acabas de ver al lanzar el comando **git checkout -b [rama] [nombreremoto]/[rama]**. Si tienes la versión 1.6.2 de Git, o superior, puedes utilizar también el parámetro **--track**:

```
$ git checkout --track origin/fix/loginUser  
Branch fix/loginUser set up to track remote branch  
refs/remotes/origin/fix/loginUser.  
Switched to a new branch "fix/loginUser" Switched to a new branch "fix/loginUser"
```

Para preparar una rama local con un nombre distinto a la del remoto, puedes utilizar:

```
$ git checkout -b fixLoginUser origin/fix/loginUser  
Branch sf set up to track remote branch refs/remotes/origin/fix/loginUser.  
Switched to a new branch "fixLoginUser"
```

Así, tu rama local **fixLoginUser** va a llevar (push) y traer (pull) hacia o desde **origin/fix/loginUser**.

Borrando ramas remotas

Imagina que ya has terminado con una rama remota. Es decir, tanto tú como tus colaboradores haz completado una determinada funcionalidad y la has incorporado (**merge**) a la rama master en el remoto (o donde quiera que tengas la rama de código estable). Puedes borrar la rama remota utilizando la sintaxis: **git push [nombreremoto] :[rama]**. Por ejemplo, si quieres borrar la rama **fix/loginUser** del servidor, puedes utilizar:

```
$ git push origin :fix/loginUser  
To git@github.com:schacon/simplegit.git  
- [deleted]      fix/loginUser
```

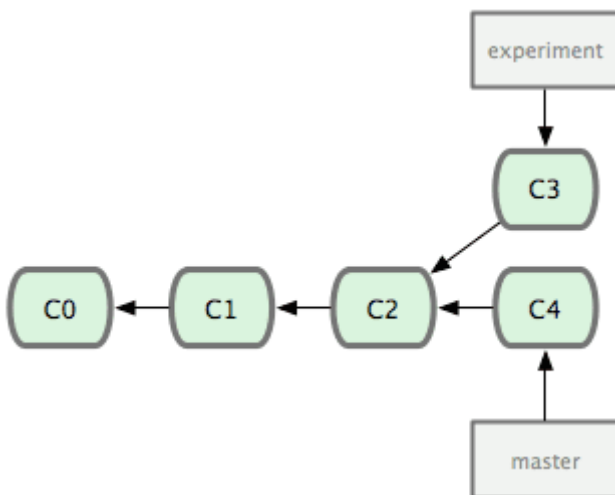
Listo!. La rama en el servidor ha desaparecido.

Reorganizar el trabajo realizado

En Git tenemos dos formas de integrar cambios de una rama en otra: la fusión (merge) y la reorganización (rebase).

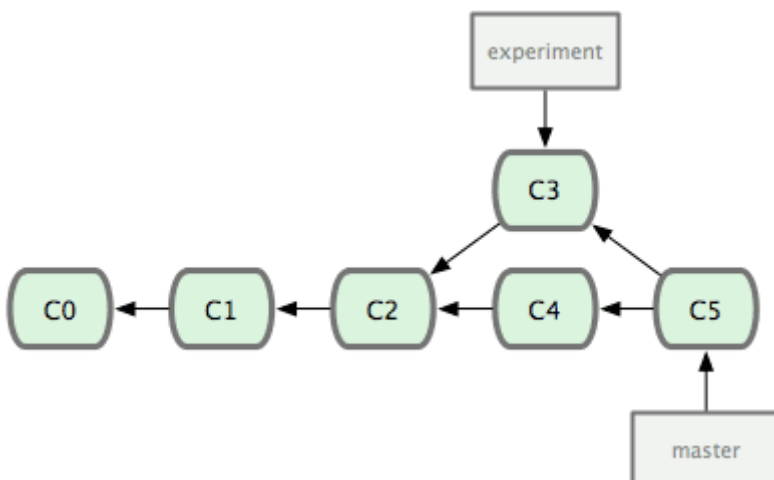
Reorganización básica

En la imagen que se encuentra a continuación nos encontramos en la necesidad de unificar las ramas `experiment` y `master`. Esto quiere decir que queremos tener los cambios realizados en ambas ramas en una sola, en este caso en `experiment`.



El registro de confirmaciones inicial.

La manera más sencilla de integrar ramas, tal y como hemos visto, es el comando `git merge`. El comando `merge` realiza una fusión a tres bandas entre las dos últimas instantáneas de cada rama (C3 y C4) y el ancestro común a ambas (C2); creando una nueva instantánea (snapshot) y la correspondiente confirmación (commit C5), según se muestra a continuación



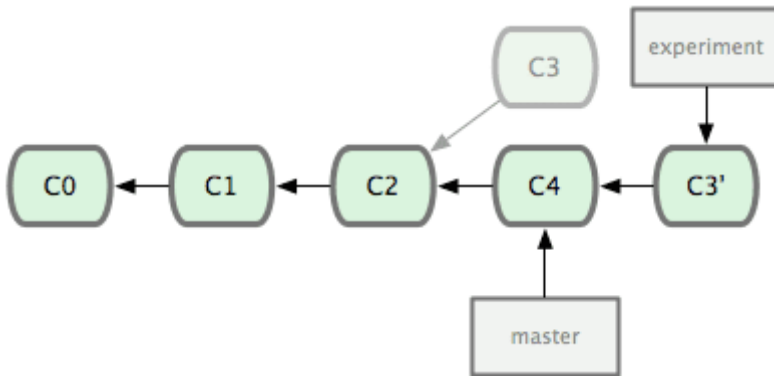
Fusionando una rama para integrar el registro de trabajos divergentes.

Aunque también hay otra forma de hacerlo: puedes agarrar los cambios introducidos en C3 y aplicarlos encima de C4. Esto es lo que en Git llamamos reorganizar (rebase). Con el comando `git rebase`, puedes tomar todos los cambios confirmados en una rama, y reaplicarlos sobre otra.

Por ejemplo, puedes lanzar los comandos:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Haciendo que Git: vaya al ancestro común de ambas ramas (donde estás actualmente y de donde quieres reorganizar), saque las diferencias introducidas por cada confirmación en la rama donde estás, guarde esas diferencias en archivos temporales, reinicie (reset) la rama actual hasta llevarla a la misma confirmación en la rama de donde quieres reorganizar, y, finalmente, vuelva a aplicar ordenadamente los cambios como se muestra a continuación.



Reorganizando sobre C4 los cambios introducidos en C3.

Así, la instantánea apuntada por C3' aquí es exactamente la misma apuntada por C5 en el ejemplo de la fusión. No hay ninguna diferencia en el resultado final de la integración, pero el haberla hecho reorganizando nos deja un registro más claro. Si examinas el registro de una rama reorganizada, este aparece siempre como un registro lineal: como si todo el trabajo se hubiera realizado en series, aunque realmente se haya hecho en paralelo.

Diferencias entre merge y rebase

El rebase unifica las ramas dejando un árbol lineal o más bonito. El merge aún deja el gráfico de las ramas.

El merge a la hora de querer unificar nos toca realizar un commit de más, este es el commit que muchos dicen commit basura ó innecesario . El rebase unifica sin necesidad de crear un nuevo commit .

El rebase unifica las ramas perdiendo el historial de los commit y el merge no . Esto puede resultar bien importante cuando se necesite llevar o saber el historial de commit y se está trabajando con otros desarrolladores en esa rama. Ten cuidado puedes llegar ser odiado por el equipo al usar rebase.

Git en servidores

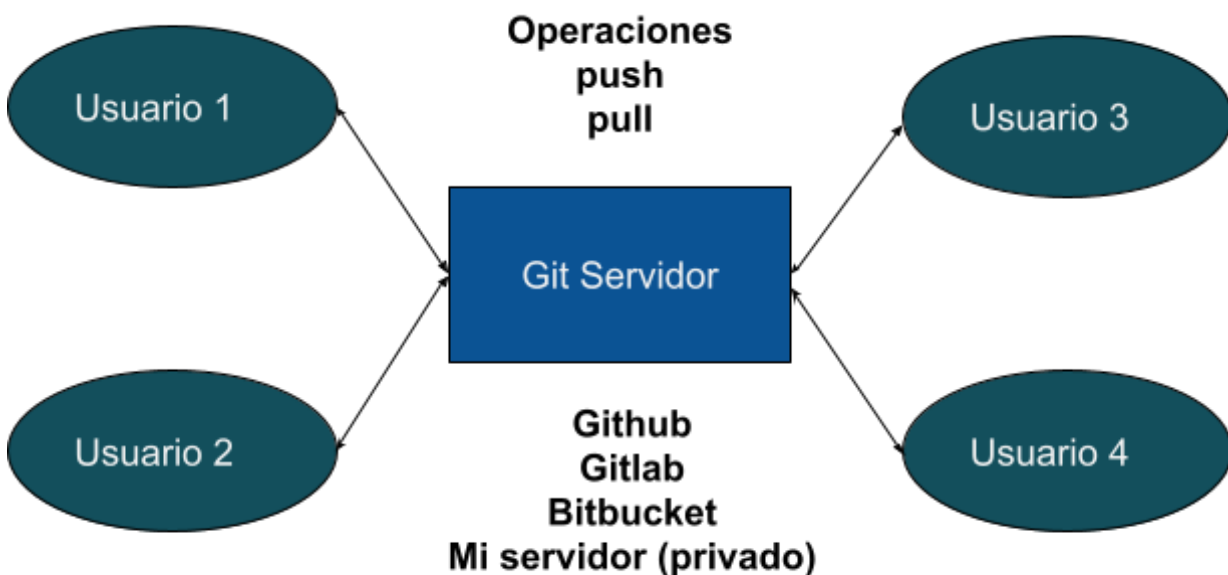
Git se puede instalar en cualquier servidor para que esté disponible en internet o en un red local de una empresa. También se puede trabajar de manera local si es que así uno lo requiera. Existen varios protocolos de comunicaciones con los cuales, uno puede bajarse un repositorio git y trabajar con ellos. Dichos protocolos son HTTPS, SSH y GIT. En esta sección abordaremos cómo trabajar con Git con servidores públicos.

Si no quieres realizar todo el trabajo de preparar tu propio servidor Git, tienes varias opciones para alojar tus proyectos Git en una ubicación externa dedicada. Esta forma de trabajar tiene varias ventajas: un alojamiento externo suele ser rápido de configurar y sencillo de iniciar proyectos en él; además de no ser necesario preocuparse de su mantenimiento.

Actualmente tienes un gran número de opciones del alojamiento, cada una con sus ventajas y desventajas. Para obtener una lista actualizada, puedes mirar en la página GitHosting del wiki principal de Git:

<https://git.wiki.kernel.org/index.php/GitHosting>

A continuación te presentamos un esquema gráfico del concepto de git en servidores.



En la mayoría de las veces que te toque trabajar en las empresas de tecnología actual, es un hecho de que vas a trabajar con grandes equipos de desarrollo. Un gran porcentaje de estas empresas usan alguno de estos servicios externos de git para almacenar allí sus proyectos de software. Nosotros vamos a abordar el uso de git con el servicio de Gitlab.

Gitlab

Para registrarte ingresa a <https://gitlab.com>



GitLab.com

GitLab.com offers free unlimited (private) repositories and unlimited collaborators.

- [Explore projects on GitLab.com](#) (no login needed)
- [More information about GitLab.com](#)
- [GitLab.com Support Forum](#)
- [GitLab Homepage](#)

By signing up for and by signing in to this service you accept our:

- [Privacy policy](#)
- [GitLab.com Terms](#).

Sign in

Register

Username or email

Password

☐ Remember me
 [Forgot your password?](#)

Sign in

Sign in with

Google

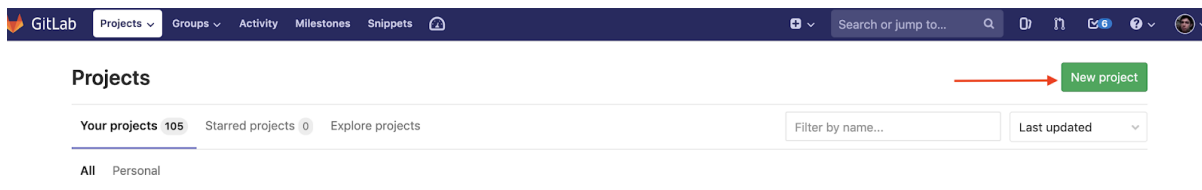
Twitter

GitHub

Bitbucket

☐ Remember me

Allí deberás crearte una cuenta o iniciar sesión si es que ya tienes una. Una vez adentro podrás crearte un repositorio de git de forma muy intuitiva haciendo click en New Project .



Para completar la creacion de un proyecto deberas completar los siguientes campos :

Project name: nombre que se le dará al proyecto

Project slug: nombre personalizado para la url del proyecto

Project description: descripción completa del proyecto

Visibility Level: esto es importante porque indica si el proyecto será visible para todos los usuarios de gitlab, solo para usuarios que tienen cuenta o es privado. Los proyecto privados necesitan permisos para poder ser descargados.

Initialize repository with a README: El objetivo de este fichero es facilitar una guía rápida a los que acaban de descubrir nuestra aplicación, API o librería de cómo empezar a usarla. Es muy importante que vaya al grano, sea conciso y muy claro. Para extendernos y entrar en detalles está la documentación, a la que siempre podemos enlazar desde dentro de nuestro README.

Finalmente para crear el proyecto dar click en el botón Create project .

Blank project
Create from template
Import project
CI/CD for external repo

Project name

→ nombre del proyecto

Project URL

rollingcodeschool

Project slug

↑ este nombre forma parte de la url del repositorio

Want to house several dependent projects under the same namespace? [Create a group](#).

Project description (optional)

Visibility Level ⓘ
☒ Private
Project access must be granted explicitly to each user.
☐ Internal
This project cannot be internal because the visibility of [rollingcodeschool](#) is private. To make this project internal, you must first [change the visibility](#) of the parent group.
☐ Public
This project cannot be public because the visibility of [rollingcodeschool](#) is private. To make this project public, you must first [change the visibility](#) of the parent group.
☒ Initialize repository with a README
Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

Create project
Cancel

Cuando el proyecto es creado, vas a ver la pagina principal del proyecto en Gitalab como se muestra a continuacion. Basicamente vas a ver el contenido del archivo README.md que coincide con la descripción del proyecto que ingresaste al momento de crearlo.

M

mi proyecto
Project ID: 11626327

Star 0
Fork 0
Clone

Add license
0 Commits
1 Branch
0 Tags
0 Bytes Files

esto es un proyecto de una pagina web para un supermercado hecho con html, css y javascript

master
mi-proyecto / +
History
Find file
Web IDE

Initial commit
Johnny Guzman authored 58 seconds ago
e32089f2

README
Add CHANGELOG
Add CONTRIBUTING
Auto DevOps enabled
Add Kubernetes cluster

Name	Last commit	Last update
README.md	Initial commit	58 seconds ago

README.md

mi proyecto

esto es un proyecto de una pagina web para un supermercado hecho con html, css y javascript

Ahora bien, hasta este punto ya tienes tu proyecto de git en un servidor externo y gratuito hasta cierto punto como lo es Gitlab. Ahora es momento de poder clonarlo para que tengas una copia local del mismo en tu computadora y así puedas empezar a trabajar sobre el.

Antes de realizar este proceso, necesitas recordar que Git trabaja con distintos protocolos de comunicación y es en este momento donde cobra más importancia conocer cuál es el más conveniente.

GIT : Lo más eficiente y rápido es usar el demonio Git nativo. Sin embargo, se ofrecen pocas funciones: sin cifrado, sin autenticación. Ideal para los repositorios públicos de solo lectura.

HTTP: La forma más compatible es HTTP. Menos eficiente que el Git nativo, pero tampoco una gran diferencia. El pro más importante de HTTP es la penetración de firewall y el soporte de proxy. Aparece como otro tráfico HTTP normal para la mayoría de las puertas de enlace / cortafuegos.

HTTPS: Más seguro es HTTPS, pero menos eficiente también. Requiere bastante configuración. También necesitarás un certificado TLS de confianza.

SSH: Seguridad similar a HTTPS, pero una forma más común es usar SSH. Es el valor predeterminado si no se especifica ningún protocolo en la línea de comandos. Proporciona un cifrado sólido y autenticación de contraseña y clave. Necesita de un par de llaves públicas y privadas para funcionar.

Vamos a usar SSH para poder clonar el repositorio pero antes debemos configurarlo.

Configurando SSH

Cada usuario del sistema ha de generarse una clave SSH, si es que no la tiene ya. El proceso para hacerlo es similar en casi cualquier sistema operativo. Ante todo, asegurate que no tengas ya una clave. Por defecto, las claves de cualquier usuario SSH se guardan en la carpeta ~/.ssh de dicho usuario. Puedes verificar si tienes ya unas claves, simplemente ubicandote sobre dicha carpeta y viendo su contenido:

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa          known_hosts
config           id_dsa.pub
```

Has de buscar un par de archivos con nombres tales como 'algo' y 'algo.pub'; siendo ese "algo" normalmente 'id_dsa' o 'id_rsa'. El archivo terminado en '.pub' es tu clave pública, y el otro archivo es tu clave privada. Si no tienes esos archivos (o no tienes ni siquiera la carpeta '.ssh'), has de crearlos; utilizando un programa llamado 'ssh-keygen', que viene incluido en el paquete SSH de los sistemas Linux/Mac o en el paquete MSysGit en los sistemas Windows:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/jhonny/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/jhonny/.ssh/id_rsa.
Your public key has been saved in /Users/jhonny/.ssh/id_rsa.pub.
The key fingerprint is:
43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a guzmanjhonny@gmail.com
```

Como se vé, este comando primero solicita confirmación de dónde van a guardarse las claves ('.ssh/id_rsa'), y luego solicita, dos veces, una contraseña (passphrase), contraseña que puedes dejar en blanco si no deseas tener que teclearla cada vez que uses la clave.

Tras generarla, cada usuario ha de encargarse de enviar su clave pública a quienquiera que administre el servidor Git (en el caso de que este esté configurado con SSH y así lo requiera). Esto se puede realizar

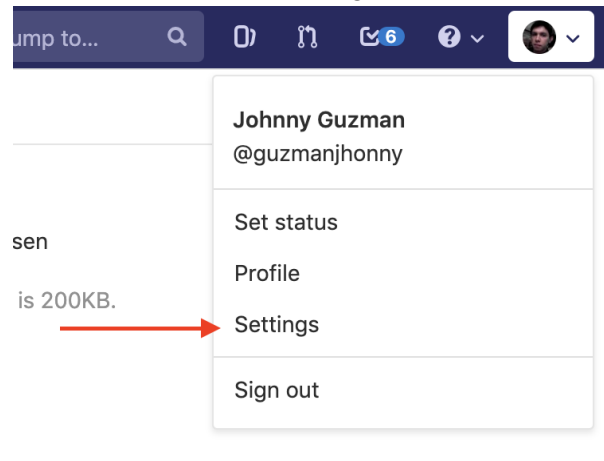
simplemente copiando los contenidos del archivo terminado en '.pub' y enviárselos por correo electrónico. La clave pública será una serie de números, letras y signos, algo así como esto:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpKDhRfHY17SbrmTIpNLTKG9Tjom/BWDSU
GP1+nafz1HDTYW7hdi4yZ5ew18JH4JW9jbhUfrviQzM7x1ELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUF1jQJKprX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+lnKatmIkjn2sold0lQratlMqVSsbx
NrRFi9wrf+M7Q== guzmanjhonny@gmail.com
```

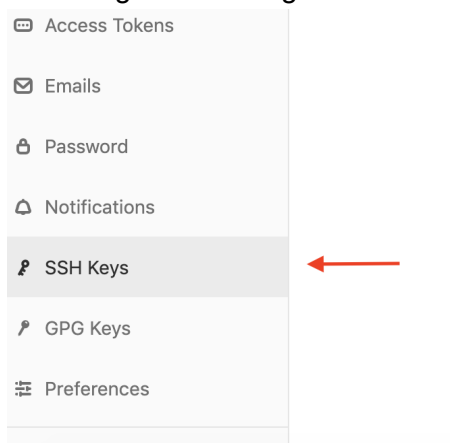
Dar de alta en GitLab el acceso con ssh

Como mencionamos anteriormente, la clave pública es la que permitirá a GitLab poder autenticarte y de esa forma poder realizar operaciones de lectura y escritura sobre tus proyectos de git.

Asegura te de copiar el contenido de tu archivo **id_rsa.pub** y a continuación dirígite a tu cuenta de Gitlab en la opciones de Settings de tu perfil de usuario , luego haz click donde dice SSH Keys como se muestra



en las siguientes imágenes:



Luego se nos presentará el siguiente formulario donde deberás pegar el contenido de la clave pública.

Entonces los campos a completar son:

key: clave pública copiada anteriormente

title: nombre unico para identificar tu clave en gitlab.

Luego de completar todos los campos se deberá dar click en Add Key y de esta manera tu clave ya estará lista para usarse.

User Settings > SSH Keys

SSH Keys

SSH keys allow you to establish a secure connection between your computer and GitLab.

Add an SSH key

To add an SSH key you need to [generate one](#) or use an [existing key](#).

Key

Paste your public SSH key, which is usually contained in the file '~/.ssh/id_rsa.pub' and begins with 'ssh-rsa'. Don't use your private SSH key.

pegar aqui tu clave pública

```
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAkIOUkDHfHY17SbrmTlpNLTK9Tiom/BWDSU
GPI+nafzIHDTYW7hdi4yZ5ew18JH4JW9jhbUFRviQzM7xlELEvf4h9IFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBIWXCFCR+HAo3FXRitBqxiX1nKhXpHAZsMcilQ8V6RisNAQwdsdMFvSIVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/elFmb1zuUFIQJKprrX88XypNDviYNbv6vw/Pb0rwert/En
mZ+AW4QZPnTPI89ZPmVMLuayrD2cE86Z/II8b+gw3r3+1nKatmlkin2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== guzmanjhony@gmail.com
```


Title

Name your individual key via a title

Add key

Muy bien, ahora es momento de clonar el repositorio para ello vamos elegir la opcion donde dice Clone en la página principal del proyecto que hemos creado anteriormente.

rollingcodeschool > mi proyecto > Details


mi proyecto

Project ID: 11626327

1 Commit

1 Branch

0 Tags

0 Bytes Files

Clone

Star 0

Fork 0

esto es un proyecto de una pagina web para un supermercado hecho con html, css y

master

mi-proyecto / +

Initial commit

Johnny Guzman authored 2 hours ago

e32089f2

README

Add CHANGELOG

Add CONTRIBUTING

Enable Auto DevOps

Add Kubernetes cluster

Set up CI/CD

Name	Last commit	Last update
README.md	Initial commit	2 hours ago

Vamos a copiar la dirección de ssh del proyecto como se indica en la imagen anterior. Dicha dirección la usaremos a continuación desde la consola de comandos. Si estás en windows pueden usar la herramienta cmd o si estas en linux o mac puedes usar la terminal de comandos. Antes de ejecutar el comando git clone asegurate de situarte en un directorio donde quieres almacenar el proyecto que vas a descargar de gitlab. En nuestro caso nos situaremos en la carpeta rollingcode.

```
$ cd /Users/johnnyguzman/Documents/projects/rollingcode
$ git clone git@gitlab.com:rollingcodeschool/mi-proyecto.git
Cloning into 'mi-proyecto'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
```

```
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
```

Esto nos creará una carpeta llamada mi-proyecto y dentro de ella estará nuestro proyecto de git.

```
$ cd /Users/johnnyguzman/Documents/projects/rollingcode/mi-proyecto
ls
README.md
```

Verás que sólo existe el archivo README.md que el que se creo de forma automática al momento de crear el repositorio en Gitlab.

Si ahora ejecutamos el comando git branch veremos:

```
$ git branch
* master
```

Como mencionamos al inicio de este manual, el comando git clone crea automáticamente una rama local llamada master y la asocia con la rama remota origin/master y esto lo puedes comprobar con el comando git checkout:

```
$ git checkout
Your branch is up to date with 'origin/master'.
```

Trabajando con el proyecto creado en Gitlab

Vamos a crear un archivo denominado prueba.html y vamos poner el siguiente fragmento de código html:

```
<!DOCTYPE html>
<head>
  <meta charset="utf-8">
  <title>Mi página</title>
</head>
<body>
  <div>
    <h1>Mi página web</h1>
  </div>
</body>
</html>
```

Ahora si ejecutamos el comando git status vamos a ver lo siguiente:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    prueba.html
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

Git nos dice que existe un archivo sin seguimiento prueba.html. Esto significa que para añadirlo al árbol de seguimiento de git necesitamos añadirlo a la zona de preparación y al área de confirmación. Para ello como

vimos anteriormente en este manual vamos a usar los siguientes comandos:

```
$ git add prueba.html
$ git commit -m "añadimos un archivo de pruebas"
[master 0ba8d0d] añadimos un archivo de pruebas
 1 file changed, 11 insertions(+)
 create mode 100644 prueba.html
```

Listo! ya tienes tu primer commit (confirmación) realizado y se encuentra dentro de la rama master. Ahora lo que vamos a realizar es subir (push) esos cambios al repositorio remoto.

Entonces primero necesitamos saber que remotos tenemos, para ello usa el comando **git remote -v**

```
$ git add prueba.html
origin      git@gitlab.com:rollingcodeschool/mi-proyecto.git (fetch)
origin      git@gitlab.com:rollingcodeschool/mi-proyecto.git (push)
```

Vemos que tenemos el remote origin disponible tanto para push como para fetch. Recuerda que el comando fetch te permite actualizar tu copia local de tu proyecto con el remoto origin.

Entonces para subir los cambios confirmados sobre la rama **master** a la rama **origin/master** del proyecto remoto alojado en gitlab, hacemos lo siguiente:

```
$ git push -u origin master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 415 bytes | 207.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To gitlab.com:rollingcodeschool/mi-proyecto.git
 e32089f..0ba8d0d  master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

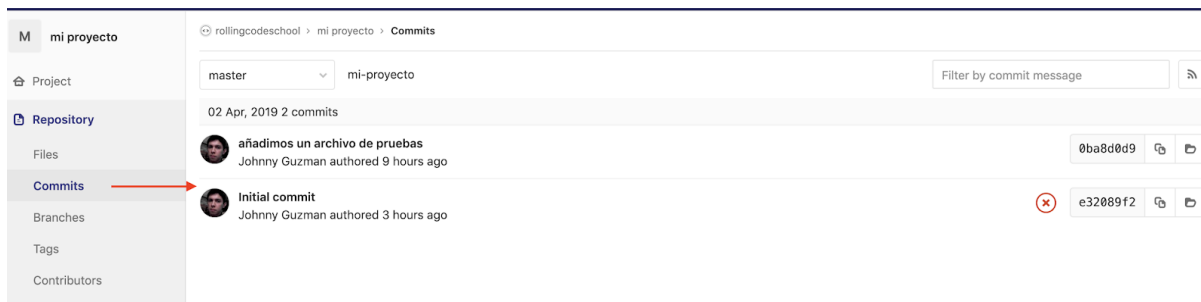
Ahora si nos vamos a Gitlab vamos a ver lo siguiente en la página principal del proyecto:

The screenshot shows the GitLab interface for a project named 'mi proyecto'. At the top, there's a navigation bar with 'rollingcodeschool > mi proyecto > Details'. Below this, the project name 'mi proyecto' is displayed with a lock icon and 'Project ID: 11626327'. To the right, there are buttons for 'Star', 'Fork', and 'Clone'. Below the project name, it says 'esto es un proyecto de una pagina web para un supermercado hecho con html, css y javascript'. A red arrow points to the 'master' branch in the dropdown menu, with a label 'rama master del remoto origin'. The main section shows the latest commit by Johnny Guzman, titled 'añadimos un archivo de pruebas', with a commit hash '0ba8d0d9'. A red arrow points to this hash with a label 'commit subido recientemente'. Below the commit, there are buttons for 'README', 'Add CHANGELOG', 'Add CONTRIBUTING', 'Enable Auto DevOps', and 'Add Kubernetes cluster'. At the bottom, there's a table with columns 'Name', 'Last commit', and 'Last update'.

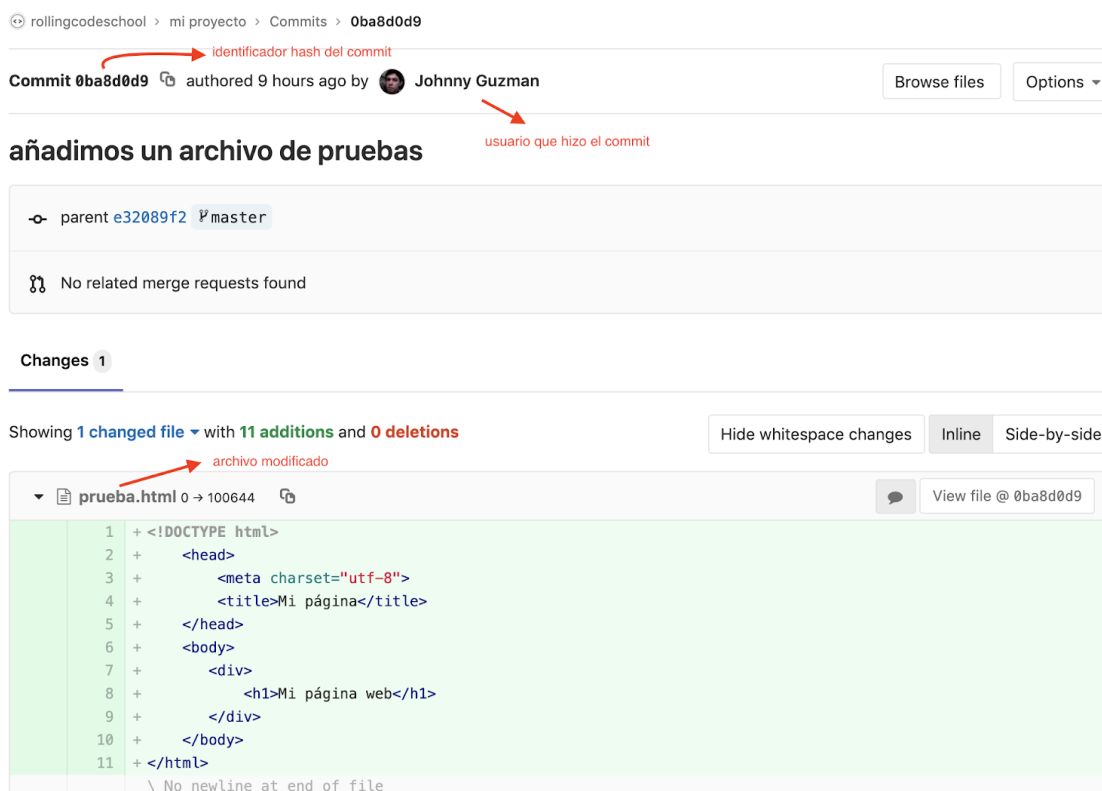
Name	Last commit	Last update
README.md	Initial commit	3 hours ago
prueba.html	añadimos un archivo de pruebas	12 minutes ago

En la página principal podemos ver la última confirmación realizada así como también detalles de la misma como la descripción que le dimos. También vemos la rama sobre la cuales aplicaron los cambios que es master. En Gitlab siempre que ingresemos a la pantalla principal del proyecto, siempre veremos master

como la rama por defecto. Si lo que queremos es ver el historial de commits realizados sobre alguna rama, entonces deberemos dirigirnos en la opción Repository -> Commits en Gitlab.



Las confirmaciones(commits) van a ir apareciendo en orden cronológico y si deseamos ver qué cambios se aplicaron sobre uno en especial, tenemos que hacer click sobre la descripción de alguno.



Dentro del detalle de un commit podrás ver mucha información del mismo. Por ejemplo el usuario que realizó la confirmación, la fecha, el hash del commit, la descripción. También podremos ver que archivos se agregaron y/o modificaron entre otras opciones disponibles.

Analogía de un sistema de control de versiones (Recetas de cocina)

Imaginemos que un chef te invita a su cocina. Una en la que no sólo puedes darte una recorrido, mirar y nada mas: se te permite leer sus libros de recetas de primera mano y conocer todos sus secretos. La cocina cuenta por ejemplo con un sistema Git mediante el cual puedes realizar sugerencias de mejora de estas recetas, copiártelas y llevártelas a tu cocina local, continuar mejorándolas, practicar, etc; pero sin la posibilidad de cambiar aquello que planteaba la chef de inicio en su receta, a no ser que te lo permita.

Has ido de visita con unas cuantas personas igual de curiosas que tú y han descubierto una fabulosa receta: cómo hacer pasta. La receta consiste en el proceso de cocinado completa de la pasta hasta que está lista para servir. La receta acaba cuando la hierve y la sirve en el plato

Glosario básico que debe manejar un aportador de ideas usando SCVs.

Repositorio (Lugar donde guardar la recetas)

Local (Cocina Local donde guardo mis recetas)

Remoto (Cocina compartida donde se guardan mis recetas para que otros puedan o no tomarlas y/o aportar. Nuevas ideas)

Commits (Idea o cambio nuevo sobre la receta)

Branch (Ideas o cambios aplicados sobre una copia de la receta original)

Push (compartir el conjunto de ideas o cambios que hice de la receta con los demás)

Pull (traer los cambios que otros aplicaron sobre la receta compartida)

Pull Request (enviar el conjunto de cambios que se realizaron sobre la receta original al dueño de la misma para ver si la aplica o no)

Issues (duda sobre algún procedimiento al cocinar la receta)

Github, Gitlab, Bitbucket (instituto de chefs donde se guardan infinidad de recetas. Algunas pueden ser de libre acceso y otras pueden ser solo para chefs autorizados)