**1.- Intro**

GitHub es el mayor proveedor de alojamiento de repositorios Git, y es el punto de encuentro para que millones de desarrolladores colaboren en el desarrollo de sus proyectos. Un gran porcentaje de los repositorios Git se almacenan en GitHub, y muchos proyectos de código abierto lo utilizan para hospedar su Git, realizar su seguimiento de fallos, hacer revisiones de código y otras cosas. Por tanto, aunque no sea parte directa del proyecto de código abierto de Git, es muy probable que durante tu uso profesional de Git necesites interactuar con GitHub en algún momento.

**2.- Colaborando en un proyecto haciendo un fork**

Si quieres participar en un proyecto existente, en el que no tengas permisos de escritura, puedes bifurcarlo (hacer un “fork”). Esto consiste en crear una copia completa del repositorio totalmente bajo tu control: se encontrará en tu cuenta y podrás escribir en él sin limitaciones.  
De esta forma, los proyectos no necesitan añadir colaboradores con acceso de escritura (push). La gente puede bifurcar un proyecto, enviar sus propios cambios a su copia y luego remitir esos cambios al repositorio original para su aprobación, creando lo que se llama un Pull Request, que veremos más adelante. Esto permite abrir una discusión para la revisión del código, donde propietario y participante pueden comunicarse acerca de los cambios y, en última instancia, el propietario original puede aceptarlos e integrarlos en el proyecto original cuando lo considere adecuado.  
Para bifurcar un proyecto, visita la página del mismo y pulsa sobre el botón “Fork” del lado superior derecho de la página. En unos segundos te redireccionarán a una página nueva de proyecto, en tu cuenta y con tu propia copia del código fuente.

**3.- Flujo de trabajo en GitHub**

GitHub está diseñado alrededor de un flujo de trabajo de colaboración específico, centrado en las solicitudes de integración (“pull request”). Este flujo es válido tanto si colaboras con un pequeño equipo en un repositorio compartido como si lo haces en una gran red de participantes con docenas de bifurcaciones particulares. El funcionamiento habitual es el siguiente:

1. Se crea una rama a partir de la rama master.
2. Se realizan algunos *commits* hacia esa rama.
3. Se envía esa rama hacia tu copia (fork) del proyecto.
4. Abres un Pull Request en GitHub.
5. Se participa en la discusión asociada y, opcionalmente, se realizan nuevos commits.
6. El propietario del proyecto original cierra el Pull Request, bien fusionando la rama con tus cambios o bien rechazándolos.

En realidad también puedes abrir un Pull Request entre dos ramas del mismo repositorio. Si estás trabajando en una característica con alguien y ambos tenéis acceso de escritura al repositorio, puedes subir una rama al mismo y abrir un Pull Request con ella de fusión con master para poder formalizar el proceso de revisión de código y discusión.

**4.- Añadir colaboradores**

Si estás trabajando con otras personas y quieres darle acceso de escritura, necesitarás añadirlas como “colaboradores”. Si Ben, Jeff y Louise se crean cuentas en GitHub, y quieres darles acceso de escritura a tu repositorio, los tienes que añadir al proyecto. Al hacerlo le darás permiso de “push”, que significa que tendrán tanto acceso de lectura como de escritura, en el proyecto y en el repositorio Git.

# [Fundamentos de Git](http://programandoandroid.wikispaces.com/Fundamentos+de+Git)

# 1.- 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 modo que puedas recuperar versiones específicas más adelante. Se usa sobre todo para el código fuente de proyectos de desarrollo, pero puedes emplearlo para casi cualquier tipo de archivo que encuentres en un ordenador. El empleo de un sistema de control de versiones (Version Control System o VCS en inglés) es muy útil, ya que te permite revertir archivos a un estado anterior, revertir el proyecto entero a un estado anterior, comparar cambios a lo largo del tiempo, ver quién modificó por última vez algo que puede estar causando un problema, quién introdujo un error y cuándo, y mucho más. Usar un VCS también significa que si fastidias o pierdes archivos, puedes recuperarlos fácilmente. Veamos lo distintos tipos de sistemas de control de versiones.

## 1.1.- Sistemas de control de versiones locales

El sistema de control de versiones local más básico consiste en copiar los archivos a otro directorio (todo lo hemos empleado), pero, aunque sencillo, es un sistema tremendamente propenso a cometer errores. Es fácil olvidar en qué directorio te encuentras, y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías.  
Para hacer frente a este problema, los programadores desarrollaron hace tiempo VCSs locales que contenían una simple base de datos en la que se llevaba registro de todos los cambios realizados sobre los archivos.  
El sistema de control de versiones local más popular fue un sistema llamado rcs, cuyo funcionamiento consistía en guardar conjuntos de parches (es decir, las diferencias entre archivos) de una versión a otra en un formato especial en disco. Así se podía recrear después cómo era un archivo en cualquier momento sumando los distintos parches.

## 1.2.- Sistemas de control de versiones centralizados

Con objeto de permitir la colaboración de varios desarrolladores en un mismo proyecto se desarrollaron los sistemas de control de versiones centralizados (Centralized Version Control Systems o CVCSs en inglés). Estos sistemas, como CVS, Subversion, y Perforce, tienen un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos de ese lugar central. Durante muchos años, éste ha sido el estándar para el control de versiones.  
Ofrece ventajas frente a los sistemas locales, como por ejemplo que todo el mundo sabe hasta cierto punto en qué está trabajando el resto de desarrolladores del proyecto y que los administradores tienen control detallado de qué puede hacer cada desarrollador. Sin embargo tiene una desventaja importante: si el servidor centralizado cae, ningún desarrollador puede colaborar o guardar cambios versionados de aquello en que están trabajando.

## 1.3.- Sistemas de control de versiones distribuidos

En un sistema de control de versiones distribuido (Distributed Version Control Systems o DVCSs en inglés), como Git, Mercurial, Bazaar o Darcs, los clientes no sólo descargan la última instantánea de los archivos: replican completamente el repositorio. Así, si un servidor muere, y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo. Cada vez que se descarga una instantánea, en realidad se hace una copia de seguridad completa de todos los datos.

# 2.- ¿Qué es Git?

## 2.1.- Introducción

En 2005, la comunidad de desarrollo de Linux (y en particular a Linus Torvalds, el creador de Linux) decidió crear su propia herramienta para el control de versiones con objeto de utilizarla en el desarrollo del núcleo de Linux.  
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. Git no modela ni almacena sus datos de este modo. En cambio, Git 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. Esto hace de Git un mini sistema de archivos con algunas herramientas tremendamente potentes construidas sobre él, más que un VCS.  
Al disponer de las distintas instantáneas del estado de los archivos en tu disco local la mayoría de operaciones se realizan también en local, por lo que por lo general no se necesita información de la red. Por ejemplo, para navegar por la historia del proyecto, Git no necesita acceder al servidor para obtener la historia y mostrártela. 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 hacer tus cambios sin problemas hasta que consigas una conexión de red para subirlos.  
Todo en Git es verificado mediante una suma de comprobación antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma (checksum en inglés). Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. 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 en Git. De hecho, Git guarda todo no por nombre de archivo, sino en la base de datos de Git por el valor hash de sus contenidos.  
A continuación se muestra un vídeo que nos explica qué es Git y por qué merece la pena usarlo:  
  
<https://youtu.be/jSJ8xhKtfP4>

## 2.2.- Instalación de Git

En el siguiente vídeo se muestran las distintas formas de instalar Git atendiendo al sistema operativo de nuestro equipo:

<https://youtu.be/1PiYqxog8mc>

Nosotros haremos del cliente gráfico para Git de GitHub, llamado GitHub Desktop, que nos instalará automáticamente Git. Puedes descargarlo en  
  
<https://desktop.github.com/>

## 2.3.- Configuración inicial

Ahora que tienes Git en tu sistema, querrás hacer algunas cosas para personalizar tu entorno de Git. Deberías tener que hacer estas cosas sólo una vez; se mantendrán entre actualizaciones.  
Desde GitHub Desktop tan sólo tendrás que pulsar en el icono del engranaje y seleccionar Options en el menú. Se nos mostrará una pantalla de configuración donde deberemos añadir nuestra cuenta de GitHub (en Accounts).  
Además, deberemos establecer un nombre de usuario y dirección de correo electrónico para configurar Git. Esto es importante porque las confirmaciones de cambios (commits) en Git usan esta información, y es introducida de manera automática en los commits que envías.  
Otro de los aspectos que podemos configurar en este formulario es el shell que debe utilizarse cuando pulsemos en el icono del engranaje y seleccionemos Open in Git Shell. El shell de Git nos van a permitir introducir los comandos (las órdenes) necesarias para llevar a cabo las acciones deseadas.  
De hecho, desde el git shell también podemos configurar git (NO GitHub) con los siguientes comandos:

|  |
| --- |
|  |
| # Configura el nombre de usuario para Git. | |
|  | |

|  |
| --- |
| $ git config --global user.name "Pedro Joya" |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Configura el email de usuario para Git. |
|  |

|  |
| --- |
| $ git config --global user.email pedro.joya@iessaladillo.es |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Muestra los distintos parámetros de configuración y su valor. |
|  |

|  |
| --- |
| $ git config --list |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Muestra el nombre de usuario para Git. |
|  |

|  |
| --- |
| $ git config user.name |

## 2.4.- Obtener ayuda

Todos las órdenes de Git tienen la forma git <comando> <opciones>. Si algún momento necesitas ayuda sobre algún comando tienes las siguientes opciones:

|  |
| --- |
|  |
| # Muestra ayuda sobre el comando indicado. | |
|  | |

|  |
| --- |
| $ git help <comando> |
|  |

|  |
| --- |
| # También sirve para mostrar ayuda sobre el comando indicado. |
|  |

|  |
| --- |
| $ git <comando> --help |

# 3.- Repositorios

## 3.1.- Creación de un repositorio en un directorio existente

Si vas a empezar el seguimiento en Git de un proyecto existente, necesitas ir al directorio del proyecto y escribir:

|  |
| --- |
|  |
| # Inicial el seguimiento Git del directorio actual (crea repositorio .git) | |
|  | |

|  |
| --- |
| $ git init |

Esto crea un nuevo subdirectorio llamado .git que contiene todos los archivos necesarios del repositorio.  
Un proyecto Git tiene tres secciones principales:

* directorio de Git o repositorio (Git directory, repository): 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 (ver en el siguiente apartado) un repositorio desde otro ordenador.
* directorio de trabajo (working directory): consiste en 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.
* área de preparación (staging area): se trata de un sencillo archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación.

## 3.2.- Clonación de un repositorio existente

Si deseas obtener una copia de un repositorio Git existente, por ejemplo, un proyecto en el que te gustaría contribuir, necesitas clonarlo. Al clonar un repositorio, Git recibe una copia de casi todos los datos que tiene el servidor. Cada versión de cada archivo de la historia del proyecto es descargado. 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. Usaremos uno de los siguientes comandos:

|  |
| --- |
|  |
| # Crea un subdirectorio llamado studio, y clona en él el repositorio | |
|  | |

|  |
| --- |
| # correspondiente a la URL indicada. |
|  |

|  |
| --- |
| $git clone https://github.com/pedrojoya/studio.git |
|  |

|  |
| --- |
| # Similar al anterior pero el subdirectorio se llamará pedro. |
|  |

|  |
| --- |
| $git clone https://github.com/pedrojoya/studio.git pedro |

Esto crea un subdirectorio llamado studio (o pedro si usamos la segundo comando), 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, verás que están los archivos del proyecto, listos para ser utilizados.  
Debemos tener en cuenta que si se trata de un repositorio extenso, dado que es necesario descargar toda la historia del mismo, el proceso de descarga puede llevar bastante tiempo.

## 3.3.- Ignorar archivos

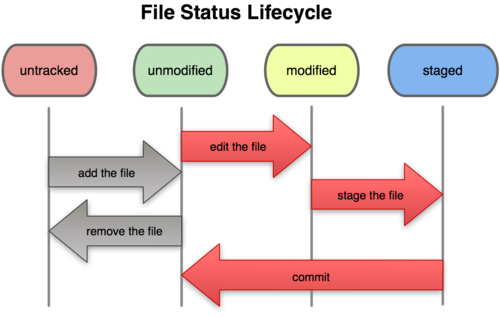
(pdte)  
.gitignore en la raíz del proyecto.  
Generador: <https://www.gitignore.io/>

# 4.- Operaciones básicas

## 4.1.- Los estados de los archivos para Git

Para Git cada archivo puede encontrarse en uno de los siguientes estados:

* sin seguimiento (untracked): El archivo es desconocido para Git. No estaba en la última instantánea ni está en el área de preparación.
* sin modificar (unmodified): El archivo está tal cual estaba en la instantánea (foto) anterior.
* modificado (modified): El archivo ha sido modificado pero todavía no lo has confirmado a tu base de datos.
* preparado (staged): Has marcado el archivo modificado en su versión actual para que vaya en tu próxima confirmación.
* confirmado (committed): Los datos del archivo están almacenados de manera segura en tu base de datos local.

  
Por tanto, el flujo de trabajo básico cuando trabajemos con archivos en Git corresponde aproximadamente a:

1. Modificar una serie de archivos en tu directorio de trabajo.
2. Preparar los archivos, añadiéndolos tu área de preparación (staging area).
3. Confirmar los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esa instantánea de manera permanente en tu directorio de Git.

## 4.2.- Comprobación del estado

Para comprobar el estado de los archivos ejecutaremos el siguiente comando:

|  |
| --- |
|  |
| # Comprueba el estado de los archivos del directorio de trabajo. | |
|  | |

|  |
| --- |
| $ git status |
|  |

|  |
| --- |
| # Similar al anterior pero con salida simplificada. |
|  |

|  |
| --- |
| $ git status --short |
|  |

|  |
| --- |
| # Similar al anterior pero abreviado. |
|  |

|  |
| --- |
| $ git status -s |

## 4.3.- Seguimiento de nuevos archivos

Para empezar el seguimiento de un nuevo archivo, es decir, para que deje de estar untracked, se usa el comando

|  |
| --- |
|  |
| # Prepara el archivo prueba.txt para la próxima confirmación. | |
|  | |

|  |
| --- |
| # pasando al estado staged, tanto si estaba untracked como modified. |
|  |

|  |
| --- |
| $ git add prueba.txt |
|  |

|  |
| --- |
| # Agrega al área de preparación todos los archivos modificados. |
|  |

|  |
| --- |
| $ git add . |

## 4.4.- Preparación de archivos modificados

Si modificamos un archivo que ya aparecía en la última instancia y queremos que dichos cambios sean añadidos al área de preparación para que vayan en tu próxima confirmación ejecutaremos el mismo comando anterior

|  |
| --- |
|  |
| # Prepara el archivo prueba.txt para la próxima confirmación. | |
|  | |

|  |
| --- |
| # pasando al estado staged, tanto si estaba untracked como modified. |
|  |

|  |
| --- |
| $ git add prueba.txt |
|  |

|  |
| --- |
| # Agrega al área de preparación todos los archivos modificados. |
|  |

|  |
| --- |
| $ git add . |

Si una vez en el área de preparación volvemos a modificar el archivo y comprobamos el estado, veremos que aparece listado a la vez como preparado y como no preparado. El motivo es que Git prepara un archivo tal y como estaba en el momento de añadir al área de preparación. Si confirmamos los cambios en este momento (commit), la versión del archivo que se incluirá en la confirmación será la que está contenida en el área de preparación, no la versión que estás viendo ahora en tu directorio de trabajo (la que has modificado después). La solución ha este problema es sencilla: bastará con volver a añadir el archivo al área de preparación, lo que actualizará la versión contenida en ésta.

## 4.5.- Diferencias entre versiones de archivos modificados y de archivos preparados

Para ver los cambios que has realizado pero aún no has preparado, en comparación con la última instantánea (último commit realizado), debes ejecutar el siguiente comando:

|  |
| --- |
|  |
| # Muestra las diferencias entre el área de preparación y el directorio de trabajo. | |
|  | |

|  |
| --- |
| $ git diff |

Debemos tener en cuenta que si hemos enviado al área de preparación todas las modificaciones realizadas, el comando anterior NO mostrará ninguna diferencia, lo que puede resultar bastante desconcertante. Si quieres comparar lo que hay en tu área de preparación con lo que hay en tu última instantánea, tendrás que ejecutar el comando

|  |
| --- |
|  |
| # Muestra las diferencias entre el área de preparación y el directorio de trabajo. | |
|  | |

|  |
| --- |
| $ git diff --staged |

## 4.6.- Confirmación de cambios

Ahora que el área de preparación está como tú quieres, puedes confirmar los cambios. Recuerda que cualquier archivo que hayas creado o modificado, pero que no hayas enviado al área de preparación NO se incluirá en esta confirmación, sino que se mantendrán como modificados. Para realizar la confirmación ejecutaremos el comando

|  |
| --- |
|  |
| # Confirma los cambios contenidos en el área de preparación | |
|  | |

|  |
| --- |
| $ git commit |

Si quieres escribir directamente el mensaje de confirmación sin tener que abrir el navegador, debe usar el siguiente comando

|  |
| --- |
|  |
| # Confirma los cambios contenidos en el área de preparación | |
|  | |

|  |
| --- |
| # e incluye un mensaje de confirmación. |
|  |

|  |
| --- |
| $ git commit -m "Arreglar constantes" |

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.  
Aunque puede ser extremadamente útil para elaborar confirmaciones exactamente a tu gusto, el área de preparación es en ocasiones demasiado complejo para las necesidades de tu flujo de trabajo. Si quieres saltarte el área de preparación, Git proporciona un atajo. Con el siguiente comando Git enviar todos los archivos modificados al área de preparación y después confirma los cambios

|  |
| --- |
|  |
| # Envía todos los archivos en seguimiento al área de preparación y | |
|  | |

|  |
| --- |
| # confirma los cambios, incluyendo un mensaje de confirmación. |
|  |

|  |
| --- |
| $ git commit -a -m "Arreglar constantes" |

## 4.7.- Listar 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. Para ello Git nos proporciona el siguiente comando

|  |
| --- |
|  |
| # Lista las confirmaciones realizadas en orden cronológico inverso. | |
|  | |

|  |
| --- |
| $ git log |

El comando git log proporciona gran cantidad de opciones para mostrarte exactamente lo que buscas. A continuación vemos las más usadas:

|  |
| --- |
|  |
| # Lista las últimas dos confirmaciones realizadas mostrando las | |
|  | |

|  |
| --- |
| # diferencias introducidas en cada una de ellas. |
|  |

|  |
| --- |
| $ git log -p -2 |
|  |

|  |
| --- |
| # Lista las confirmaciones realizadas mostrando estadísticas resumidas. |
|  |

|  |
| --- |
| $ git log --stat |
|  |

|  |
| --- |
| # Muestra cada confirmación de forma resumida en una línea. |
|  |

|  |
| --- |
| $ git log --oneline |
|  |

|  |
| --- |
| # Muestra las confirmación con decoración. |
|  |

|  |
| --- |
| $ git log --decorate |
|  |

|  |
| --- |
| # Muestra las confirmaciones en forma de grafo. |
|  |

|  |
| --- |
| $ git log --graph |
|  |

|  |
| --- |
| # Muestra las confirmaciones de todas las ramas. |
|  |

|  |
| --- |
| $ git log --all |

El siguiente vídeo se muestra cómo crear nuestro repositorio y los pasos a seguir hasta realizar nuestra primera confirmación:  
  
<https://youtu.be/UJGKWMHX038>

## 4.8.- Modificación de la 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

|  |
| --- |
|  |
| # Reemplaza la última confirmación, añadiéndole lo contenido | |
|  | |

|  |
| --- |
| # en el área de preparación y le modifica el mensaje de confirmación. |
|  |

|  |
| --- |
| $ 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.  
NO se recomienda el uso de este comando si la confirmación a enmendar ya ha sido compartida con un servidor, ya que por tanto ha podido ser descargada por otros desarrolladores y basar su trabajo en él, de manera que nuestra confirmación no sería igual que las suyas. Estaríamos por tanto reescribiendo la historia del repositorio. En tal caso se recomienda realizar una nueva confirmación.  
El siguiente vídeo ahonda en el concepto de área de preparación y en cómo modificar la última confirmación realizada:  
  
<https://youtu.be/6CzY3WQVYUo>

## 4.9.- Diferencias entre confirmaciones

Con el siguiente comando podemos ver cuáles son las diferencias entre dos confirmaciones realizadas

|  |
| --- |
|  |
| # Muestra las diferencias entre las dos confirmaciones. | |
|  | |

|  |
| --- |
| $ git diff f5f0388 b8484be |

## 4.10.- Eliminación de archivos

Cuando eliminamos un archivo de nuestro directorio de trabajo del que Git ya estaba haciendo seguimiento (es decir, que no está en estado untracked), debemos informar de dicho evento a Git propagando el cambio al área de preparación, para que cuando se realice una confirmación ésta deje de incluir dicho archivo. Si no propagamos dicho cambio al staging area, será tratado de forma similar a un archivo modificado que aún no haya sido enviado al área de preparación (ver salida del comando git status).  
Podemos propagar posteriormente el cambio al área de preparación mediante el comando git add, pero Git nos proporciona un comando para realizar ambas operaciones (eliminación y propagación) en un solo paso:

|  |
| --- |
|  |
| # Propaga al staging area la intención de borrar el archivo. | |
|  | |

|  |
| --- |
| git rm PROJECTS.md |

Si el archivo eliminado ya se encontraba en el staging area porque antes de ser borrado había sido modificado y enviado con el comando git add, el comando anterior nos lanzará un error. El motivo es que Git proporciona esta 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. Si aún así queremos forzar la eliminación, tendremos que usar un parámetro adicional en el comando anterior:

|  |
| --- |
|  |
| # Fuerza a eliminar del staging area el archivo PROJECTS.md | |
|  | |

|  |
| --- |
| # incluso aunque había sido añadido explícitamente. |
|  |

|  |
| --- |
| git rm -f PROJECTS.md |

## 4.11.- Renombrado de archivos

Al igual que en el caso de la eliminación de un archivo, cuando renombramos un archivo de nuestros directorio de trabajo de que Git ya estaba haciendo seguimiento (es decir, que no está en estado untracked), debemos informar de dicho evento a Git propagando el cambio al área de preparación, para que cuando se realice una confirmación ésta incluya el nuevo nombre del archivo.  
Esta operación puede ser llevada a cabo en los siguientes pasos: renombrar el archivo, eliminar el archivo con el nombre antiguo del staging area mediante git rm y añadir al staging area el archivo con el nombre nuevo con git add. Por fortuna Git nos ofrece un comando para realizar dichas operaciones en un sólo paso:

|  |
| --- |
|  |
| # Renombra el archivo README.md a README, | |
|  | |

|  |
| --- |
| # propagando el cambio al staging area. |
|  |

|  |
| --- |
| git mv README.md README |

# 5.- Deshaciendo operaciones

## 5.1.- Deshacer cambios de un archivo modificado

¿Qué pasa si te das cuenta de que no quieres mantener las modificaciones que has hecho sobre un archivo que aún no has enviado al área de preparación? ¿Cómo puedes deshacerlas fácilmente revirtiendo el archivo al mismo estado en el que estaba cuando hiciste tu última confirmación? Podrás usar el siguiente comando:

|  |
| --- |
|  |
| # Deja el archivo prueba.txt tal y como estaba en la última instantánea | |
|  | |

|  |
| --- |
| # descartando los cambios. |
|  |

|  |
| --- |
| $ git checkout -- prueba.txt |

Debes ser consciente de que las modificaciones han desaparecido, ya que el archivo se ha sobrescribido por la versión de la última instantánea, por lo que NO debe usar este comando a no ser que estés absolutamente seguro de que no quieres las modificaciones.

## 5.2.- Quitar las modificaciones de un archivo del área de preparación

Si hemos realizado modificaciones sobre un archivo y las hemos añadido por error al área de preparación, podemos resetear el estado del archivo en dicho área al estado en el que se encuentra en la última confirmación, apareciendo la versión modificada sólo en nuestro directorio de trabajo. Para ello ejecutaremos el comando:

|  |
| --- |
|  |
| # Saca el archivo prueba.txt del área de preparación. | |
|  | |

|  |
| --- |
| $ git reset HEAD prueba.txt |

En el siguiente vídeo se estudian los comandos para deshacer cambios realizados sobre archivos que aún no están en el área de preparación y para los que sí lo están:  
  
<https://youtu.be/9AkcsMpKYhI>

## 5.3.- Dejar de seguir un archivo

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 (que pase al estado de untracked). Esto resulta particularmente útil cuando olvidaste añadir algo a tu archivo .gitignore y lo añadiste accidentalmente. Para llevar a cabo esta acción deberemos ejecutar el comando:

|  |
| --- |
|  |
| # El archivo readme.txt deja de ser seguido por Git (untracked) | |
|  | |

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

## 5.4.- Eliminación de una confirmación mediante reseteo a una confirmación anterior

Git nos permite deshacer confirmaciones ya realizadas, básicamente reseteando nuestro directorio de trabajo al estado en el que se encontraba cuando se realizó una determinada confirmación anterior. Debemos tener en cuenta que se trata de un proceso destructivo, por lo que debe ser usado con cuidado, especialmente si las confirmaciones eliminadas ya habían sido compartidas remotamente en un servidor, ya que otro desarrollador tendría un árbol de confirmaciones distinta.  
Git permite determinar también qué hacer con los cambios que se habían realizado en las confirmaciones eliminadas, ofreciendo las siguientes opciones: que pasen al estado de archivos modificados, que sean eliminados por completo o que se sitúen en el área de preparación. Veamos los comandos necesarios:

|  |
| --- |
|  |
| # Elimina las confirmaciones posteriores a la que tiene ese hash. | |
|  | |

|  |
| --- |
| # Los cambios que se habían realizado en las confirmaciones borradas |
|  |

|  |
| --- |
| # pasan a estar en el estado de archivos modificados. |
|  |

|  |
| --- |
| $ git reset f5f0388 |
|  |

|  |
| --- |
| # Elimina las confirmaciones posteriores a la que tiene ese hash. |
|  |

|  |
| --- |
| # Los cambios que se habían realizado en las confirmaciones borradas |
|  |

|  |
| --- |
| # son eliminadas. |
|  |

|  |
| --- |
| $ git reset --hard f5f0388 |
|  |

|  |
| --- |
| # Elimina las confirmaciones posteriores a la que tiene ese hash. |
|  |

|  |
| --- |
| # Los cambios que se habían realizado en las confirmaciones borradas |
|  |

|  |
| --- |
| # se dejan en el área de preparación. |
|  |

|  |
| --- |
| $ git reset --soft f5f0388 |

En el siguiente vídeo se muestran dichas operaciones:  
  
<https://youtu.be/h-xN14zPgsQ>

## 5.5.- Eliminación de una confirmación revirtiendo sus cambios

El problema del comando anterior es que es destructivo, ya que podemos llegar a perder los cambios introducidos por las confirmaciones eliminadas. Una alternativa mejor es, en vez de eliminar una confirmación, crear una nueva confirmación que revierta los cambios, es decir, que realice los cambios en el sentido contrario. Por ejemplo, si una línea existente era reemplaza por otra, ahora se realizará el reemplazo al revés. Esta opción no es destructiva, ya que simplemente incorpora una nueva confirmación, evitando los problemas con los repositorios remotos que comentamos en el apartado anterior. Para llevar a cabo esta opción Git nos proporciona el siguiente comando:

|  |
| --- |
|  |
| # Crea una nueva confirmación que revierte los cambios introducidos | |
|  | |

|  |
| --- |
| # en todas las confirmaciones posteriores a la indicada. |
|  |

|  |
| --- |
| $ git revert f5f0388 |
|  |

|  |
| --- |
| # Se revierten todos los commits realizados desde el actual (HEAD) |
|  |

|  |
| --- |
| # hasta el commit f5f0388, y después se realiza un commit con el |
|  |

|  |
| --- |
| # nuevo estado. |
|  |

|  |
| --- |
| $ git revert --no-commit f5f0388..HEAD |
|  |

|  |
| --- |
| $ git commit |

El siguiente vídeo muestra cómo ver las diferencias entre dos confirmaciones y cómo revertir los cambios creando una nueva confirmación que realice los cambios en sentido inverso.  
  
<https://youtu.be/4SwaV29SpIc>

# 6.- Enlaces de interés

En el siguiente enlace disponemos de un tutorial interactivo para aprender los comandos de Git:  
<https://try.github.io/levels/1/challenges/1>  
A continuación se muestra otro tutorial interactivo con comandos más avanzados:  
<http://gitreal.codeschool.com/?utm_source=github&utm_medium=codeschool_option&utm_campaign=trygit>  
Minicurso de Git:  
<https://github.com/aprende-git/curso-git>  
Estupendo tutorial de Git (en perfecto inglés):  
<https://es.atlassian.com/git/tutorials/>

[**Trabajando con ramas**](http://programandoandroid.wikispaces.com/Trabajando+con+ramas)

**1.- Intro**

[1.- Intro](http://programandoandroid.wikispaces.com/Trabajando%20con%20ramas#x1.- Intro)

[2.- Creación de una rama](http://programandoandroid.wikispaces.com/Trabajando%20con%20ramas#x2.- Creación de una rama)

[3.- Posicionamiento en una rama](http://programandoandroid.wikispaces.com/Trabajando%20con%20ramas#x3.- Posicionamiento en una rama)

[4.- Listado de ramas](http://programandoandroid.wikispaces.com/Trabajando%20con%20ramas#x4.- Listado de ramas)

[5.- Fusión de ramas mediante fastforward](http://programandoandroid.wikispaces.com/Trabajando%20con%20ramas#x5.- Fusión de ramas mediante fastforward)

[6.- Eliminación de una rama](http://programandoandroid.wikispaces.com/Trabajando%20con%20ramas#x6.- Eliminación de una rama)

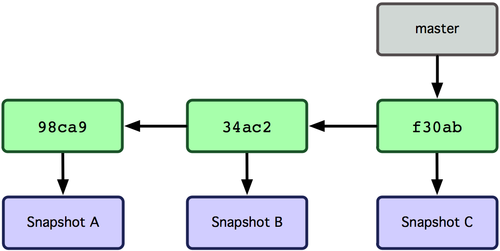
[7.- Fusión de ramas mediante confirmación](http://programandoandroid.wikispaces.com/Trabajando%20con%20ramas#x7.- Fusión de ramas mediante confirmación)

[8.- Conflictos en las fusiones](http://programandoandroid.wikispaces.com/Trabajando%20con%20ramas#x8.- Conflictos en las fusiones)

[9.- Reorganización](http://programandoandroid.wikispaces.com/Trabajando%20con%20ramas#x9.- Reorganización)

Como mencionamos en un apartado anterior, Git no almacena las confirmaciones de forma incremental (guardando solo diferencias), sino que las almacena como una serie de instantáneas (copias puntuales de los archivos completos, tal y como se encuentran en ese momento). En cada confirmación, Git almacena un punto de control que conserva:

* un apuntador a la copia puntual de los contenidos preparados (staged)
* unos metadatos con el autor y el mensaje explicativo
* uno o varios apuntadores a las confirmaciones que sean padres directos de ésta (un padre en los casos de confirmación normal, y múltiples padres en los casos de estar confirmando una fusión de dos o mas ramas, como veremos).

  
Una rama Git es simplemente un apuntador móvil apuntando a una de esas confirmaciones. 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.  
En el siguiente vídeo se nos informa acerca de lo que es una rama y por qué conviene trabajar con ellas  
  
<https://youtu.be/rmO7t35l1XI>

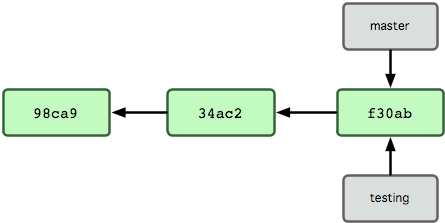
**2.- Creación de una rama**

¿Qué sucede cuando creas una nueva rama? Pues que 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

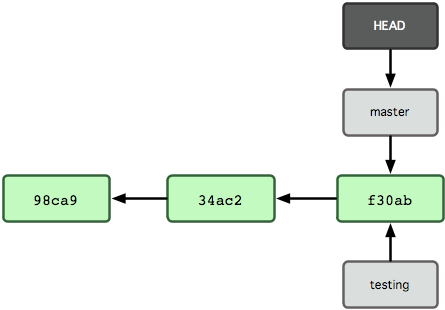
|  |
| --- |
|  |
| # Crea una nueva rama (un nuevo puntero) llamada testing | |
|  | |

|  |
| --- |
| # que apunta a la última confirmación realizada. |
|  |

|  |
| --- |
| $ git branch testing |

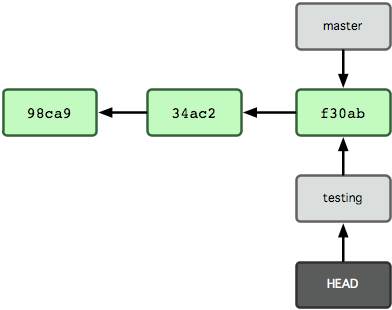
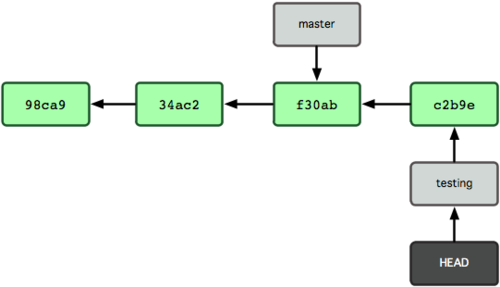
Esto creará un nuevo apuntador apuntando al mismo commit donde estés actualmente, como vemos en la imagen  


**3.- Posicionamiento en una rama**

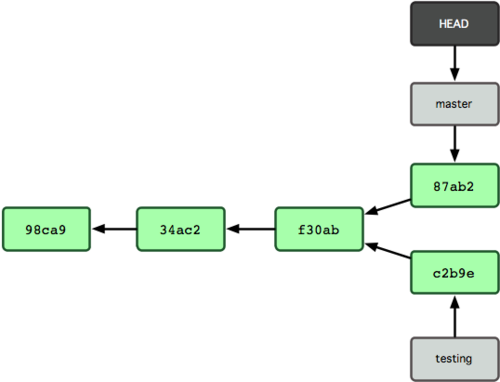
¿Cómo sabe Git en qué rama estás en este momento? Pues mediante un apuntador especial denominado HEAD (cabecera), que apunta a la rama local en la que tú estés en ese momento.  
  
Según la imagen anterior nos encontramos en la rama master, ya que el puntero HEAD apunta a ella.  
Debemos tener en cuenta que, por defecto, al crear una nueva rama con el comando git branch el puntero HEAD NO apunta a esta nueva rama, sino a la rama que ya apuntaba. Por tanto Git nos proporciona un comando para que podamos cambiarnos de rama, es decir para hacer que HEAD apunte a una determinada rama:

|  |
| --- |
|  |
| # Posiciona el puntero HEAD para que apunte a la rama testing | |
|  | |

|  |
| --- |
| $ git checkout testing |

Tras ejecutar este comando, el estado del repositorio quedaría como sigue:  
  
¿Y si estando en la rama testing hacemos una nueva confirmación? Pues el puntero de rama testing avanzará, pero el puntero de rama master no, como vemos en la siguiente figura  
  
Volvamos de nuevo a la rama master mediante el comando git checkout master. Este comando realiza dos acciones:

* mueve el apuntador HEAD de nuevo a la rama master
* revierte los archivos de tu directorio de trabajo; dejandolos tal y como estaban en la última instantánea confirmada en dicha rama master.

Si ahora realizamos una nueva confirmación en la rama master, el estado del repositorio quedaría como apreciamos en la siguiente imagen:  
  
Finalmente, Git también propociona la posibilidad de usando un sólo comando crear una nueva rama y posicionarse en ella:

|  |
| --- |
|  |
| # Crea una nueva rama llamada material y se posiciona en ella | |
|  | |

|  |
| --- |
| $ git checkout -b material |

**4.- Listado de ramas**

Para listar las ramas existentes en un repositorio bastará con ejecutar el siguiente comando:

|  |
| --- |
|  |
| # Lista las ramas del repositorio. La rama apuntada por HEAD | |
|  | |

|  |
| --- |
| # se muestra con un \* |
|  |

|  |
| --- |
| $ git branch |
|  |

|  |
| --- |
| # Lista las ramas que ya han sido fusionadas. |
|  |

|  |
| --- |
| $ git branch --merged |
|  |

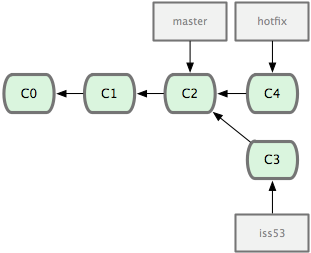
|  |
| --- |
| # Listas las ramas que aún no han sido fusionadas. |
|  |

|  |
| --- |
| $ git branch --mo-merged |

En los siguientes vídeos se nos muestran las operaciones de creación, posicionamiento y listado de ramas, así como la realización de confirmaciones sobre las nuevas ramas  
  
<https://youtu.be/j0U9jBmP3LM>

<https://youtu.be/Pw4TOTIkhfk>

**5.- Fusión de ramas mediante fastforward**

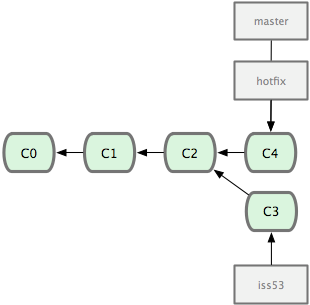
Supongamos que el estado de nuestro repositorio es el que se muestra a continuación. Tenemos la rama principal master y otras dos ramas, llamadas hotfix y iss53, en las que se han realizado confirmaciones respecto a cuando se crearon. Además, en la rama master no se ha realizado ninguna confirmación desde que se crearon las otras dos ramas.  
  
Si ya hemos terminado el trabajo con la rama hotfix podremos fusionarla con la rama master. Para ello ejecutaremos los siguientes comandos:

|  |
| --- |
|  |
| # Nos posicionamos en la rama destino (lo ideal es que sea la rama master). | |
|  | |

|  |
| --- |
| $ git checkout master |
|  |

|  |
| --- |
| # Incorporamos mediante fusión la rama hotfix en la rama master. |
|  |

|  |
| --- |
| $ git merge hotfix |

Si analizamos al respuesta del comando, veremos que nos indica que se ha realizado un *fast forward* (avance rápido). Este término hace referencia a que en realidad la fusión ha consistido en simplemente avanzar el apuntador de la rama destino una o más confirmaciones hacia delante. Este tipo de fusión sólo se puede llevar a cabo cuando desde la confirmación a la que apunta la rama origen se puede acceder directamente a la confirmación a la que apunta la rama de destino, ya que no hay ningún otra confirmación divergente. Así, nuestro repositorio quedaría en el siguiente manera  
  
En el siguiente vídeo se muestra cómo se lleva a cabo una fusión *fast forward*:  
  
<https://youtu.be/vu4Rv1SmzwM>

**6.- Eliminación de una rama**

Una vez que vayamos a dejar de trabajar con una rama, normalmente porque ya ha sido fusionada con la rama master, podemos proceder a eliminarla, de manera que desaparezca el puntero de rama. Para poder eliminar una rama deberemos estar posicionados en otra rama. Además, por defecto no podremos borrar una rama que aún no ha sido fusionada. Veamos los comandos:

|  |
| --- |
|  |
| # Borra la rama hotfix (su puntero de rama). | |
|  | |

|  |
| --- |
| # Si la rama no ha sido anteriormente fusionada, no |
|  |

|  |
| --- |
| # no lo permitirá. |
|  |

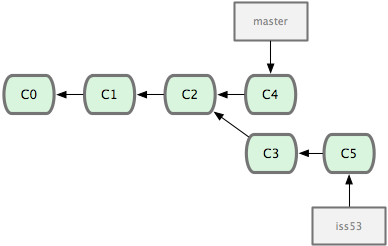
|  |
| --- |
| $ git branch -d hotfix |
|  |

|  |
| --- |
| # Borra la rama hotfix incluso aunque no haya |
|  |

|  |
| --- |
| # sido fusionada. |
|  |

|  |
| --- |
| $ git branch -D hotfix |

**7.- Fusión de ramas mediante confirmación**

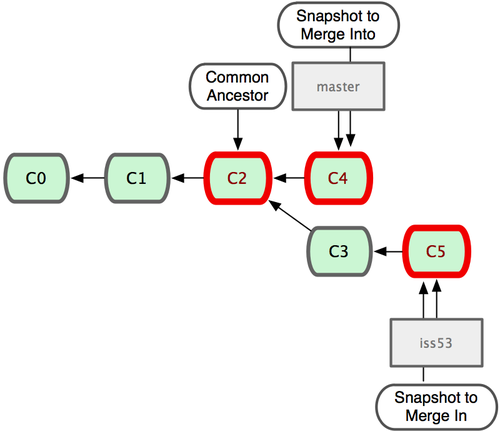
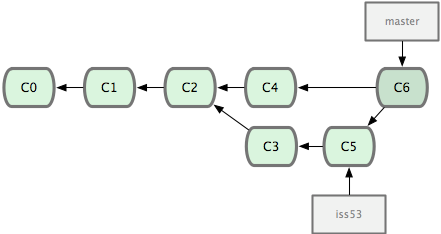
Supongamos ahora que el estado de nuestro repositorio es el que se muestra a continuación. Se creo una rama iss53 sobre la que se han realizado confirmaciones, pero en la rama master también se han hecho confirmaciones  
  
Si ahora ejecutamos los siguientes comandos para llevar fusionar la rama iss53 en la rama master

|  |
| --- |
|  |
| # Nos posicionamos en la rama destino (lo ideal es que sea la rama master). | |
|  | |

|  |
| --- |
| $ git checkout master |
|  |

|  |
| --- |
| # Incorporamos mediante fusión la rama iss53 en la rama master. |
|  |

|  |
| --- |
| $ git merge iss53 |

En este caso, el registro de desarrollo ha divergido en un punto anterior. Debido a que la confirmación en la rama actual no es ancestro directo de la rama que pretendes fusionar, no se puede realizar *fast forward*, sino que Git tiene cierto trabajo extra que hacer. En realidad Git llevará a cabo una fusión a tres bandas, utilizando las dos instantáneas apuntadas por el extremo de cada una de las ramas y por el ancestro común a ambas dos  
  
En lugar de simplemente avanzar el apuntador de la rama, Git crea una nueva instantánea (snapshot) resultante de la fusión a tres bandas; y crea automáticamente una nueva confirmación de cambios (commit) que apunta a ella. Nos referimos a este proceso como "fusión confirmada". Y se diferencia en que tiene más de un padre. Así, el repositorio quedaría de la siguiente manera  
  
En el siguiente vídeo se muestre cómo se lleva a cabo una fusión mediante confirmación:  
  
<https://youtu.be/vu4Rv1SmzwM>

**8.- Conflictos en las fusiones**

En algunas ocasiones, el proceso de fusión de dos ramas no se puede realizar automáticamente. Si hay modificaciones dispares en una misma porción de un mismo archivo en las dos ramas distintas que pretendes fusionar, Git no será capaz de fusionarlas directamente. Por ejemplo, si en el ejemplo anterior has modificado en la rama iss53 una misma porción de código que también había sido modificada en la rama hotfix (posteriormente fusionada con la rama master), cuando intentemos fusionar la rama iss53 con la rama master tendremos un conflicto

|  |
| --- |
|  |
| # Se trata de fusionar la rama iss53 hacia la rama master | |
|  | |

|  |
| --- |
| $ git merge iss53 |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| Auto-merging index.html |
|  |

|  |
| --- |
| CONFLICT (content): Merge conflict in index.html |
|  |

|  |
| --- |
| Automatic merge failed; fix conflicts and then commit the result. |

Como se nos indica en la salida del comando, se ha producido un conflicto en la fusión del archivo index.html, por lo que Git no crea automáticamente una nueva fusión mediante confirmación, sino que hace una pausa en el proceso, esperando a que tú resuelvas el conflicto. Para ver qué archivos permanecen sin fusionar en un determinado momento conflictivo de una fusión, puedes usar el comando git status.  
Para facilitarnos la tarea de arreglar nosotros mismos el conflicto, Git añade a los archivos conflictivos unas marcas especiales, que tendremos que eliminar cuando hayamos resuelto el entuerto. El archivo conflictivo contendrá algo como:

|  |
| --- |
|  |
| <<<<<<< HEAD:index.html | |
|  | |

|  |
| --- |
| <div id="footer">contact : email.support@github.com</div> |
|  |

|  |
| --- |
| ======= |
|  |

|  |
| --- |
| <div id="footer"> |
|  |

|  |
| --- |
| please contact us at support@github.com |
|  |

|  |
| --- |
| </div> |
|  |

|  |
| --- |
| >>>>>>> iss53:index.html |

donde nos dice que la versión en HEAD (la rama master, en la que te habías posicionado 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 iss53 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, o una mezcla de ambos.  
Por ejemplo, puedes optar por cambiar el bloque, dejándolo tal que:

|  |
| --- |
|  |
| <div id="footer"> | |
|  | |

|  |
| --- |
| please contact us at email.support@github.com |
|  |

|  |
| --- |
| </div> |

Esta corrección contiene un poco de ambas partes. Y se han eliminado completamente las líneas <<<<<<< , ======= y >>>>>>>.   
Tras resolver todos los bloques conflictivos, has de lanzar comandos git add para añadir al área de preparación cada archivo modificado, lo que indica a Git que sus conflictos han sido resueltos. Finalmente, debes lanzar el comando git commit para terminar de confirmar la fusión que había quedado pendiente al haberse producido conflictos:

|  |
| --- |
|  |
| # Se añaden los archivos conflictivos arreglados a la //staging area//. | |
|  | |

|  |
| --- |
| $ git add archivo\_conflictivo.txt |
|  |

|  |
| --- |
| # Se realiza la confirmación pendiente de la fusión. |
|  |

|  |
| --- |
| $ git commit |

Si por cualquier motivo no vemos factible realizar la fusión iniciada por ser muy compleja podemos cancelarla mediante el comando

|  |
| --- |
|  |
| # Aborta la fusión ya iniciada. | |
|  | |

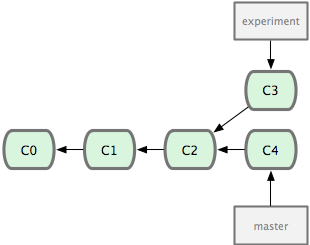
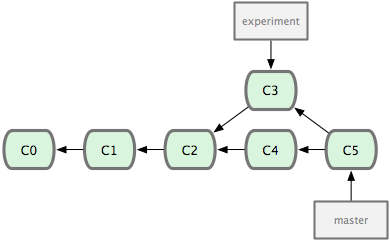
|  |
| --- |
| $ git merge --abort |

En el siguiente vídeo se muestra la resolución de conflictos en las fusiones:

<https://youtu.be/3-rCELg8feQ>

En el siguiente vídeo nos explican por qué no hay que asustarse con los conflictos en las fusiones:  
  
<https://youtu.be/CzOj076YGRY>

**9.- Reorganización**

Además de la fusión, Git nos proporciona otra forma de integrar los cambios de una rama en otra, conocida como reorganización (*rebase*). Supongamos que nuestro repositorio se encuentra en el estado descrito por la siguiente imagen  
  
Como ya hemos vistos, para integrar las confirmaciones de la rama experiment en la rama master podríamos usar el comando git merge, que 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), quedando el repositorio como sigue  
  
Sin embargo, Git nos ofrece otra posibilidad, la reorganización (*rabase*), que consiste básicamente en aplicar los cambios introducidos por las confirmaciones de una rama origen en una determinada rama destino. Como consecuencia, en la rama destino se crean nuevas confirmaciones espejo de las de la rama origen. Además, se hace que el puntero de la rama origen apunte a la confirmación espejo de la que apuntaba. Dicho de un modo sencillo, la reescribe la historia del repositorio para hacer como que la rama origen nunca hubiera divergido de la rama destino, incorporando los cambios introducidos por la primera en la segunda.  
Siguiendo el ejemplo anterior, si ejecutamos los siguientes comandos

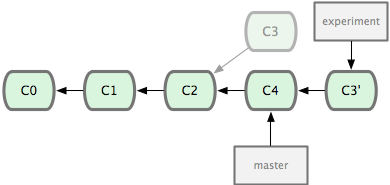
|  |
| --- |
|  |
| # Nos posicionamos en la rama experiment. | |
|  | |

|  |
| --- |
| $ git checkout experiment |
|  |

|  |
| --- |
| # Aplicamos los cambios introducidos por la rama |
|  |

|  |
| --- |
| # experiment en la rama master. |
|  |

|  |
| --- |
| $ git rebase master |

nuestro repositorio quedaría de la siguiente manera:  
  
Internamente, Git realiza las siguientes operaciones:

* Busca el ancestro común de ambas ramas.
* Obtiene las diferencias introducidas por cada confirmación de la rama origen y las guarda en archivos temporales.
* Crea en la rama de destino las confirmaciones espejo de las confirmaciones de la rama origan.
* Resetea la rama origen para que apunte a la confirmación espejo de la que apuntaba.

Debemos tener en cuenta que durante el proceso de reconfiguración se pueden producir conflictos. En este caso deberemos editar los archivos correspondientes para resolver manualmente los conflictos. Una vez resueltos, tendremos que añadir de nuevo dichos archivos al *staging area* y continuar con el proceso de reconfiguración:

|  |
| --- |
|  |
| # Se añade el archivo conflictivo arreglado al área de preparación. | |
|  | |

|  |
| --- |
| $ git add archivo\_conflictivo.txt |
|  |

|  |
| --- |
| # Se continúa con el proceso de reconfiguración. |
|  |

|  |
| --- |
| $ git rebase --continue |

Si no vemos factible resolver los conflictos, podemos deshacer la reconfiguración en marcha mediante:

|  |
| --- |
|  |
| # Cancela la reconfiguración en marcha. | |
|  | |

|  |
| --- |
| $ git rebase --abort |

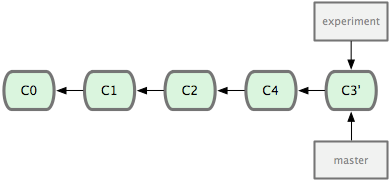
Una vez finalizada correctamente la reconfiguración, puedes volver a posicionarte en la rama de destino y hacer una fusión con avance rápido (*fast-forward merge*):

|  |
| --- |
|  |
| # Nos posicionamos en la rama master | |
|  | |

|  |
| --- |
| $ git checkout master |
|  |

|  |
| --- |
| # Fusionamos la rama experiment (se realizará fast-forward). |
|  |

|  |
| --- |
| $ git merge experiment |

Tras dicha operación nuestro repositorio quedaría finalmente de la siguiente manera:  
  
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 haber optado por la reorganización nos ha dejado un histórico más limpio, lineal, como si todo el trabajo se hubiera realizado en serie, aunque realmente se haya hecho en paralelo.  
La reorganización tiene la ventaja de que no genera una nueva confirmación como ocurre con la fusión, pero tiene el inconveniente de que estamos reescribiendo el historial del repositorio, lo cual puede conllevar problemas con otros desarrolladores. Por tanto, la recomendación es nunca usar la reorganización si las confirmaciones de la rama origen han sido ya enviadas a un servidor, ya que puede que otro desarrollador haya creado su trabajo a partir de ellas.  
Entonces, ¿qué es mejor la fusión (*merge*) o la reorganización (*rebase*). La respuesta depende de lo que entendamos por historial del repositorio.  
Para algunos, el historial de confirmaciones de tu repositorio es un registro de todo lo que ha pasado. Un documento histórico, valioso por sí mismo y que no debería ser alterado. Desde este punto de vista, cambiar el historial de confirmaciones es casi como blasfemar; estarías mintiendo sobre lo que en verdad ocurrió. ¿Y qué pasa si hay una serie desastrosa de fusiones confirmadas? Nada. Así fue como ocurrió y el repositorio debería tener un registro de esto para la posteridad.  
La otra forma de verlo es que el historial de confirmaciones es la historia de cómo se hizo tu proyecto, como si fuera una sucesión lineal de eventos.  
Por tanto, cada equipo y cada proyecto es diferente, y tendrán que decidir cuál de las dos es mejor para la situación en particular. Normalmente, la manera de sacar lo mejor de ambas opciones es reorganizar tu trabajo local, que aún no has compartido con nadie más, antes de enviarlo a algún lugar; pero nunca reorganizar nada que ya haya sido compartido.  
En el siguiente vídeo se nos explican las ventajas e inconvenientes de utilizar este comando:

<https://youtu.be/_kheHCD5m7Q>

**1.- Etiquetado**

[1.- Etiquetado](http://programandoandroid.wikispaces.com/Otras%20herramientas%20de%20Git#x1.- Etiquetado)

[2.- Almacenamiento provisional de cambios](http://programandoandroid.wikispaces.com/Otras%20herramientas%20de%20Git#x2.- Almacenamiento provisional de cambios)

Git tiene la posibilidad de etiquetar como importantes determinadas confirmaciones del historial. Esta funcionalidad se usa típicamente para marcar versiones de lanzamiento (v1.0, por ejemplo).  
Git utiliza dos tipos principales de etiquetas:

* Etiquetas ligeras: Es simplemente un puntero a una determinada confirmación. Es similar a una rama pero que NO se mueve.
* Etiquetas anotadas: Corresponden a objetos completos, que contienen el nombre del etiquetador, correo electrónico y fecha, un mensaje asociado, y pueden ser firmadas y verificadas con GNU Privacy Guard (GPG). Se recomienda usar este tipo de etiquetas.

Para crear un etiqueta ligera tendremos que ejecutar el siguiente comando:

|  |
| --- |
|  |
| # Crea una etiqueta ligera para la posición actual de HEAD. | |
|  | |

|  |
| --- |
| $ git tag v1.4-lw |
|  |

|  |
| --- |
| # Crea un tag ligero para la confirmación indicada. |
|  |

|  |
| --- |
| $ git tag v1.2 9fceb02 |

Si lo que queremos es crear una etiqueta anotada entonces tendremos que usar parámetros adicionales:

|  |
| --- |
|  |
| # Crea una etiqueta anotada para la posición atual de HEAD, | |
|  | |

|  |
| --- |
| # con el nombre v1.4 y le asocia el mensaje "mi version 1.4" |
|  |

|  |
| --- |
| $ git tag -a v1.4 -m 'mi version 1.4' |

Para listar las etiquetas existentes usaremos:

|  |
| --- |
|  |
| # Lista las etiquetas existentes. | |
|  | |

|  |
| --- |
| $ git tag |

Puedes ver la información de la etiqueta junto con la confirmación que está etiquetado de la siguiente manera:

|  |
| --- |
|  |
| # Muestra información sobre la etiqueta v1.4 y | |
|  | |

|  |
| --- |
| # la confirmación asociada a ella. |
|  |

|  |
| --- |
| $ git show v1.4 |

Para eliminar una etiqueta usaremos el comando:

|  |
| --- |
|  |
| git tag -d v1.4 | |

Por defecto, el comando git push no transfiere las etiquetas a los servidores remotos, por lo que debemos hacerlo explícitamente. De esta manera cuando alguien clone o traiga información del repositorio, también obtendrá todas las etiquetas:

|  |
| --- |
|  |
| # Envia al repositorio remoto origin la etiqueta indicada. | |
|  | |

|  |
| --- |
| $ git push origin v1.5 |
|  |

|  |
| --- |
| # Envía al repositorio remoto origin todas las etiquetas |
|  |

|  |
| --- |
| # definidas localmente que no estuvieran en el servidor. |
|  |

|  |
| --- |
| $ git push origin --tags |

Debemos tener en cuenta que una etiqueta NO es un rama, y por tanto no nos podemos posicionar en una etiqueta (git checkout), puesto que la etiqueta no se puede mover. Si quieres colocar en tu directorio de trabajo una versión de tu repositorio que coincida con alguna etiqueta, debes crear una rama nueva en esa etiqueta, sabiendo que si la rama crece la etiqueta NO se moverá:

|  |
| --- |
|  |
| # Crea una nueva rama version2 a partir de la | |
|  | |

|  |
| --- |
| # confirmación apuntada con la etiqueta v2.0.0 |
|  |

|  |
| --- |
| # y se posiciona en ella. |
|  |

|  |
| --- |
| $ git checkout -b version2 v2.0.0 |

En el siguiente vídeo nos hablan de las etiquetas ligeras:

<https://youtu.be/5DkX3HFgklM>

Y el siguiente vídeo sobre etiquetas anotadas:  
  
<https://youtu.be/h145ORRg7Ao>

**2.- Almacenamiento provisional de cambios**

En algunas ocasiones mientras estamos trabajando en unos determinados cambios se nos requiere para que realicemos alguna otra tarea de forma urgente. Como ya hemos comentado, lo habitual es que para esta nueva tarea creemos una nueva rama. El problema es que en este instante nuestro espacio de trabajo está en un estado inconsistente, ya que estábamos trabajando en otra cosa. Esto plantea el problema de, para no perder el trabajo que estábamos haciendo, tener que crea una confirmación con cambios en un trabajo a medio hacer, antes de posicionarnos en la rama correspondiente a la nueva tarea, simplemente para poder volver a ese punto más tarde. Dicha confirmación es del todo menos óptima, como podríamos pensar.  
Con objeto de solucionar este problema Git nos proporciona un comando que nos va a permitir guardar temporalmente el estado del directorio de trabajo, con todas las modificaciones en los archivos bajo control de cambios, de manera dicho estado se guarde en una pila provisional. Posteriormente podremos recuperar el estado desde la pila, aplicándolo de nuevo sobre el directorio de trabajo.

|  |
| --- |
|  |
| # Almacena el estado actual del directorio de trabajo en | |
|  | |

|  |
| --- |
| # un pila temporal. |
|  |

|  |
| --- |
| $ git stash |

Después de ejecutar dicho comando el directorio de trabajo queda limpio, con lo que nos podemos posicionar sin problemas en la rama correspondiente a la nueva tarea.  
Para ver el contenido de dicha pila, se emplea el comando

|  |
| --- |
|  |
| # Lista todos los estados almacenados en la pila temporal. | |
|  | |

|  |
| --- |
| $ git stash list |

Para volver a aplicar al directorio de trabajo un determinado estado almacenado en la pila ejecutaremos el siguiente comando:

|  |
| --- |
|  |
| # Aplica al directorio de trabajo el estado | |
|  | |

|  |
| --- |
| # más reciente de la pila temporal |
|  |

|  |
| --- |
| $ git stash apply |
|  |

|  |
| --- |
| # Aplica al directorio de trabajo un |
|  |

|  |
| --- |
| # determinado estado de la pila temporal. |
|  |

|  |
| --- |
| git stash apply stash@{2} |

Git vueve a aplicar los correspondientes cambios en los archivos que estaban modificados, pero no conserva la información de lo que estaba o no estaba añadido al área de preparación. Para conseguir esto último, es necesario añadir una opción adicional al comando

|  |
| --- |
|  |
| # Aplica al directorio de trabajo el estado | |
|  | |

|  |
| --- |
| # más reciente de la pila temporal, dejando |
|  |

|  |
| --- |
| # el staging area tal y como estaba en el momento |
|  |

|  |
| --- |
| # de almacenar el estado en la pila. |
|  |

|  |
| --- |
| $ git stash apply --index |

El comando git stash apply aplica al directorio de trabajo un estado almacenado en la pila, pero NO elimina dicho estado de la misma. Deberemos eliminarla explícitamente mediante el comando

|  |
| --- |
|  |
| # Elimina de la pila temporal un determinado estado. | |
|  | |

|  |
| --- |
| $ git stash drop stash@{0} |

Si queremos llevar a cabo ambas operaciones, aplicación del estado al directorio de trabajo y eliminación del mismo de la pila, en un solo paso, podemos ejecutar el comando

|  |
| --- |
|  |
| # Aplica al directorio de trabajo el estado | |
|  | |

|  |
| --- |
| # más reciente de la pila temporal, y lo |
|  |

|  |
| --- |
| # elimina de la pila. |
|  |

|  |
| --- |
| $ git stash pop |

Si se almacenamos un estado en la pila duramente mucho tiempo, puede ocurrir que al volver a aplicarlo tengamos problemas si otros desarrolladores han seguido trabajando en la rama a la que correspondía el estado. Se pueden presentar conflictos de fusión, que debería resolver manualmente. En este caso, tenemos disponible otra opción, consistente en crear una nueva rama, a partir de la confirmación sobre la que se estaba trabajando cuando se almacenó el estado, aplicando el estado en el directorio de trabajo resultante en la nueva rama, procediendo a eliminar de la pila el estado usado. Para ello ejecutaremos el siguiente comando:

|  |
| --- |
|  |
| # Elimina de la pila temporal un determinado estado. | |
|  | |

|  |
| --- |
| $ git stash drop stash@{0} |

En el siguiente vídeo se muestra el almacenamiento temporal de cambios:

<https://youtu.be/fD7ZOK5kZMs>

**1.- Interfaz Grafica**

El manejo de la interfaz grafica nos facilita bastante si no se nos da bien usar los comandos para las tareas sencillas de commit pull push pull request.