Guía de Git

```
 _____ ___ _____
/ ___|_ _|_   _|
| |  _ | |  | |
| |_| || |  | |
\____|___| |_|
```

Versión Guía 0.161

El contenido de este documento es una recopilación de la documentación de Git disponible en su página oficial, de la documentación de Github y la incorporación del material de clases de desafiolatam

Objetivos

1. Conocer las bondades de Git
2. Entender como funcionan los sistemas de cambio descentralizados
3. Utilizar Git para controlar cambios
4. Trabajar sobre branches

Acerca del 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.

Ventajas del control de versiones.

- Permite recuperar versiones específicas de un código
- Ayuda a gestionar los cambios hechos por varias personas
- Permite encontrar en que momento se introdujo un cambio que causa errores.

Sistemas de control de versiones centralizados

los CVSs (Centralized Version Control Systems) como es el caso de Subversion, y Perforce, tienen un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos desde ese lugar central. Durante muchos años éste fue el estándar para el control de versiones.

Ventajas

Todo el mundo puede saber (hasta cierto punto) en qué están trabajando los otros colaboradores del proyecto. Los administradores tienen control detallado de qué

puede hacer cada uno; y es mucho más fácil administrar un CVCs que tener que lidiar con bases de datos locales en cada cliente.

Desventajas

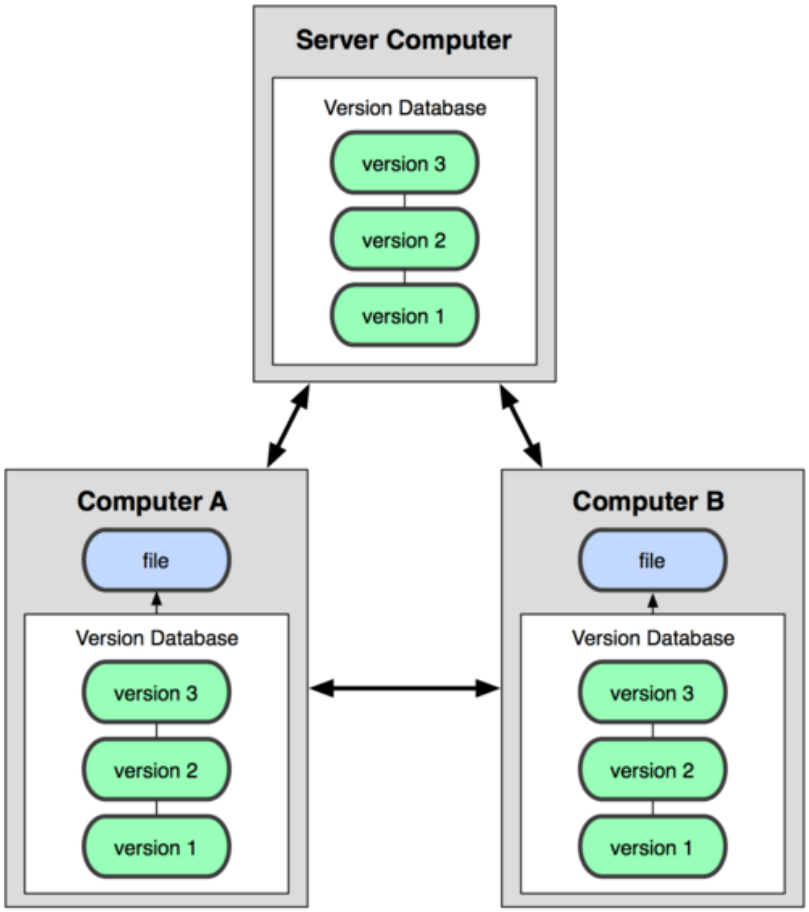
La más obvia es el punto único de fallo que representa el servidor centralizado. Si ese servidor se cae durante una hora, entonces durante esa hora nadie puede colaborar o guardar cambios versionados de aquello en que están trabajando. Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han llevado copias de seguridad adecuadamente, pierdes absolutamente toda la historia del proyecto salvo aquellas instantáneas que la gente pueda tener en sus máquinas locales.

Sistemas de control de versiones distribuidos

En un DVCS (Distributed Version Control Systems), 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.

Una gran ventaja

Puedes colaborar con distintos grupos de gente simultáneamente dentro del mismo proyecto. Esto te permite establecer varios flujos de trabajo que no son posibles en sistemas centralizados, como pueden ser los modelos jerárquicos.

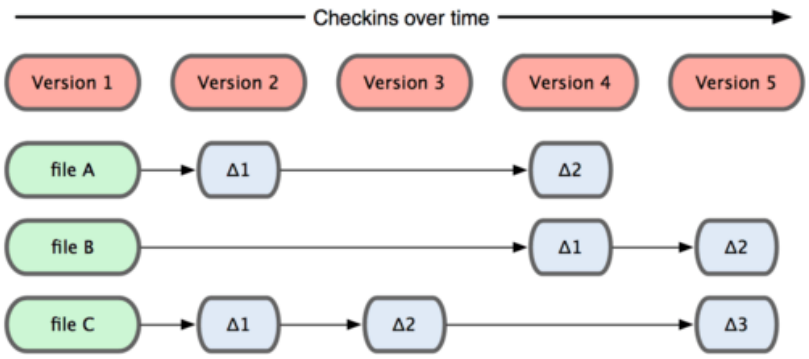


Fundamentos de Git

Instantáneas, no diferencias

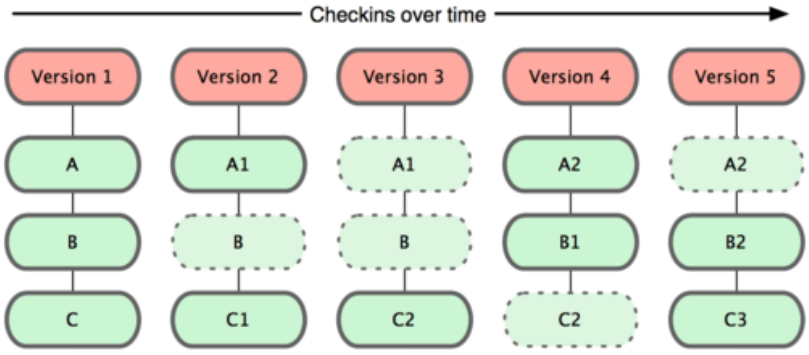
Conceptualmente, la mayoría de los demás sistemas de versionamiento almacenan la información como una lista de cambios en los archivos. Estos sistemas modelan la

información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo, como se ilustra a continuación:



Por el contrario, 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, solo un enlace al archivo anterior idéntico que ya tiene almacenado. El diagrama sería así:



Casi cualquier operación es local

La mayoría de las acciones en Git solo requieren de recursos y archivos locales para funcionar.

Algunos ejemplos de acciones que no requieren conexión a un servidor:

- Navegar por la historia del proyecto.
- Ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes.
- Confirmar los cambios a tu base de datos.

Integridad

Es imposible cambiar los contenidos de cualquier archivo sin que Git lo sepa. Esto se logra gracias a que Git verifica todo mediante una checksum (suma de comprobación). Por esto mismo no puedes perder información durante la transmisión o sufrir corrupción de archivos sin que Git lo detecte.

El mecanismo que usa Git para generar el checksum 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 y el tamaño de los archivos, más información en http://alblue.bandlem.com/2011/08/git-tip-of-week-objects.html

Este es ejemplo de hash SHA-1:

```
d250f7fa22f573efc873e3650663494834e23a95
```

Esta verificación por hash la utiliza Git muy seguido, de hecho no guarda nombres de archivo, sino que por el valor hash de su contenido.

Casi siempre solo añade información:

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. 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 nos permite experimentar con seguridad en nuestros proyectos, porque siempre está la posibilidad de volver a un estado anterior.

Los tres estados

Esta es la parte más importante de aprender. Git tiene tres estados principales en los que se pueden encontrar tus archivos:

- **Confirmado** (committed) Significa que los datos están almacenados de manera segura en tu base de datos local.
- **Modificado** (modified) Significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos.
- **Preparado** (staged) Significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Las tres áreas de trabajo

En base a esto es que tenemos tres áreas principales en todo proyecto con Git

- **El directorio de Git** (Git directory) 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** (working directory) 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** (staging area). Es un sencillo archivo, ¡ contenido en el directorio .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.

Flujo de trabajo

El flujo de trabajo básico en Git consiste principalmente en tres pasos.

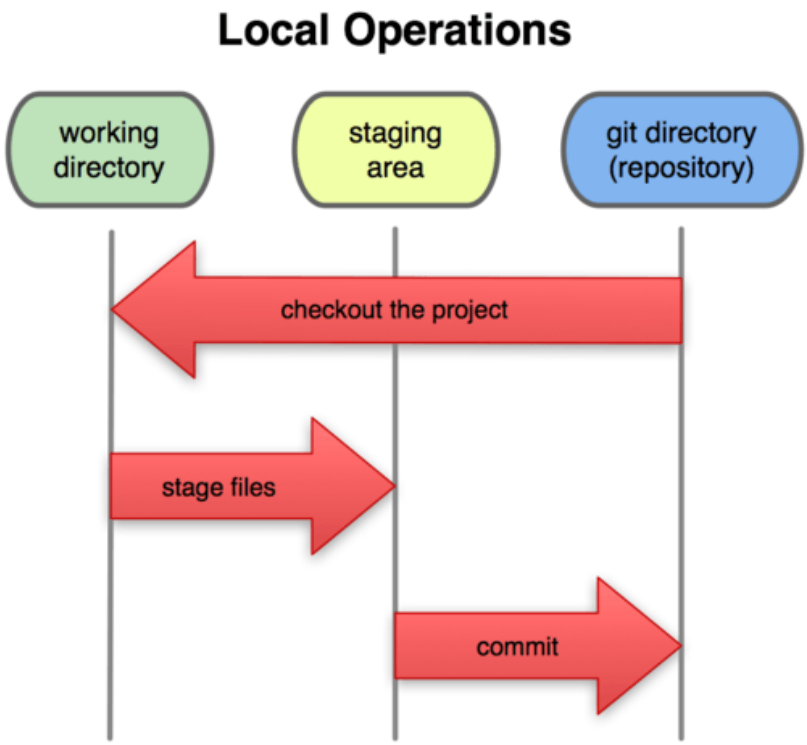
Modificas una serie de archivos en tu directorio de trabajo.

Preparas los archivos, añadiendolos 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).



Preguntas

1. ¿Cómo funciona un sistema de control centralizado?
2. ¿Cómo funciona un sistema de control distribuido?
3. ¿Cuáles son los tres estados que puede tener un archivo en Git?
4. ¿Cuál es el flujo de trabajo en Git?
5. ¿Para qué sirve el area de preparación?

Utilizando GIT

Instalación

Linux (Debian)
En Debian (Ubuntu) podemos usar apt-get

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

OS X

En OS X, si tienes instalado Homebrew puedes instalar Git así:

```
brew install git
```

Configuración

Lo primero después de instalar 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 "Tu Nombre"
git config --global user.email tucorreo@mail.com
```

Esta configuración la debes hacer solo una vez si utilizas –global , ya que Git usará esta información para todo lo que ejecutes. A Git también le puedes especificar el editor de texto y la herramienta de diferencias que usará por defecto.

```
git config --global core.edit emacs
git config --global merge.tool vimdiff
```

En el ejemplo definimos el emacs y vimdiff como editor y herramienta de diferencias por defecto.
Para saber el estado de configuración de Git puedes consultarlo así

```
git config --list
```

Inicializando un proyecto

Si estás comenzando el seguimiento con git de un proyecto necesitas **ir dentro del directorio del proyecto** y ejecutar:

```
git init
```

Al momento de hacer Git init obtendras un mensaje como el siguiente:

```
Initialized empty Git repository in /Users/gonz
```

Por ahora lo que sabremos es que esto crea un nuevo subdirectorio llamado .git que contiene todos los archivos necesarios del repositorio: un esqueleto de un repositorio Git.

En los sistemas unix las carpetas que empiezan con el caracter `.` quedan escondidas, para poder ver el resultado puedes hacer

```
ls -a
```

Un error de principiante es crear el repositorio en el home o en la raíz del computador, de esa forma pierdes la capacidad de trabajar con distintos git, y deberías tener uno distinto por proyecto, para solucionarlo borra la carpeta .git

Un segundo error, es hacer git init en un proyecto que ya está bajo control de versiones, en ese caso no tendremos

problemas, solo un mensaje que dice:

```
Reinitialized existing Git repository in /Users
```

Clonando un repositorio existente

Si deseas obtener una copia de un repositorio Git que ya exista, como por ejemplo un proyecto en el que desees colaborar, el comando necesario es `git clone [url]`. Por ejemplo para clonar el proyecto Ruby debemos ejecutar:

```
1 git clone https://github.com/ruby/ruby.git
```

Esta acción crea un directorio llamado ruby, inicializará GIT creando el archivo .git y descargará todos los archivos del repositorio dentro del directorio ruby. Puedes especificar cómo se llamará el directorio de destino así:

```
1 git clone https://github.com/ruby/ruby.git micopiaderuby
```

Con esto ocurrirá lo mismo que con el comando anterior, pero ahora el directorio de destino será micopiaderuby.

Podemos saber en que estado están nuestro archivos ocuparemos

```
1 git status
```

Como nuestro proyecto es nuevo obtendremos un mensaje diciendo:

```
On branch master
nothing to commit, working directory clean
```

Sumando los primeros archivos al control de cambio

Si estamos en un carpeta vacía no tenemos nada para agregar al control de cambios, pero podemos crear un archivo vacío con `touch`

```
1 touch prueba1
```

Luego al hacer git status veremos un mensaje distinto.

On branch master
Untracked files:
(use "git add ..." to include in what will be committed)

```
1 prueba1
```

Esto muestra que nuestro directorio de trabajo tiene un archivo que no está bajo tracking, podemos agregar el archivo ocupando `git add`

```
1 git add prueba1
```

Al hacer `git status` obtendremos:

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

    new file:   prueba1
```

Esto quiere decir que el archivo prueba1 ahora está en la area de staging, y listo para el commit.

Entonces el siguiente paso es hacer el commit, cuando uno hace un commit debe agregar un mensaje, estos son muy útiles a la hora de rastrear algún problema, el primer mensaje suele ser first commit

```
git commit -m "first commit"
```

al hacerlo obtendremos:

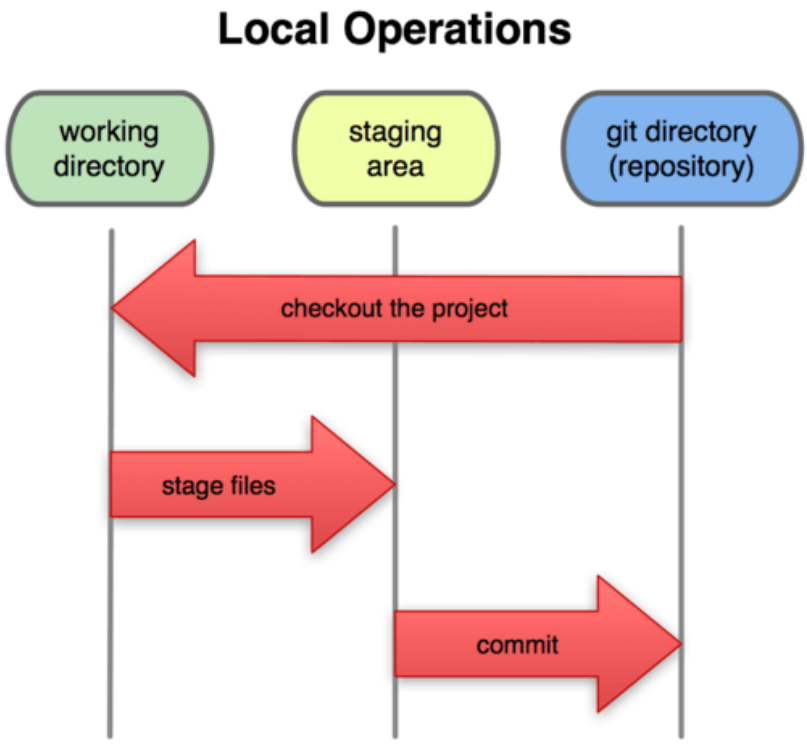
```
[master 8dd4c74] first commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 prueba1
```

luego al hacer git status obtendremos

```
On branch master
nothing to commit, working directory clean
```

Aquí vemos el flujo de trabajo, modificamos el working directory ya sea creando o editando archivos existentes, luego los agregamos al area de staging via add, y finalmente confirmamos los cambios via commit.

Si ahora vemos esta imagen que estudiamos previamente, veremos que tiene más sentido.



Luego podemos revisar todos los cambios hechos ocupando git log

```
commit 8dd4c7436c691968b3512a26bfdf2b654dfa5d9a
Author: Gonzalo Sanchez <gonzalo.sanchez.d@gmail.com>
Date:   Wed Dec 23 11:28:59 2015 -0600

    first commit
```

Existe un atajo, que permite agregar los cambios realizados y confirmar los cambios.

```
git commit -am "mensaje"
```

git commit -am a diferencia de git add no agrega archivos nuevos, solo cambios de archivos que ya esten siendo trackeados

Es posible corregir el último commit hecho utilizando el argumento `--amend` esto permite cambiar el último mensaje utilizado o agregar archivos que debieron haber sido confirmado pero se olvidaron.

Nunca debe usarse –amend después de haber hecho push, un cambio enviado no debe ser corregido localmente, causa inconsistencia.

Deshaciendo cambios

Podemos deshacer un commit ocupando

```
git reset --soft HEAD~
```

Al hacerlo veremos que quedamos en el caso anterior despues del add y antes del commit.

```
On branch master
Changes to be committed:
  (use "git reset HEAD ..." to unstage)

    new file:   prueba1
```

Luego si queremos sacar los archivos del area de staging podemos hacerlo utilizando

```
git reset HEAD prueba1
```

También podemos hacer el reset y sacarlos del area de staging utilizando git reset –hard HEAD~, pero para que esto funcione debemos hacer el add y el commit nuevamente (ya que en el paso anterior lo revertimos)

```
git add prueba1
git commit -m "first commit"
git reset --hard HEAD~
git status
```

```
On branch master
nothing to commit, working directory clean
```

En el uso diario de Git es poco frecuente deshacer un commit, es más común hacer un ammend

Actividad 1

- Crear un nuevo repositorio
- Crear tres archivos (1.txt, 2.txt, 3.txt)
- Crear un commit 1.txt
- Agregar los otros dos archivos, pero sacar 3.txt del staging area y commitear sólo 2.txt
- Modificar 1.txt, escribiendo algo en el archivo
- Descartar estos cambios

- Modificar 1.txt de nuevo pero esta vez commitear.
- Borrar 2.txt y commitear este cambio.

Preguntas

1. ¿Cómo se crea un proyecto nuevo con Git?
2. ¿Cómo se clona un proyecto con Git?
3. ¿Cuál es la diferencia entre git add y git commit?
4. ¿Como se agrega un archivo a staging?
5. ¿Como se saca un archivo de staging?
6. ¿Como se confirma un set de cambios?
7. ¿Como se puede ver el historial de cambios?
8. ¿Como se puede ver el estado de los archivos?
9. ¿Qué hace git status?
10. ¿Cómo se pueden ver todos los cambios confirmados?
11. ¿Cómo se puede deshacer un cambio confirmado?
12. ¿Cuál es la diferencia entre git reset –soft HEAD~ y git reset –hard HEAD~?

Branches

En git es posible crear ramificaciones del código, esto nos permite generar uno o mas commits fuera de la línea central y poder volver a esta en caso de algún problema.

La dinámica de uso de branches consiste en:

- crear un branch

- cambiar el branch actual al nuevo
- crear y modificar archivos (git add)
- confirmar cambios (git commit)
- volver al branch
- realizar un merge (unir las ramas)
- resolver conflictos si es que existen
- agregar los conflictos resueltos
- borrar el branch si ya no es requerido.

Existen diferentes flujos de trabajo con GIT, uno sencillo pero muy útil consiste en tener dos ramas, el master en el cual sólo deberíamos tener código funcionando, el de desarrallo sobre el cual se trabaja habitualmente, cuando implementemos completamente una funcionalidad con existo hacemos el merge con el master.

Podemos ver todos los branch que tenemos en nuestro repositorio con

```
git branch
```

Creando un branch

Para crear un branch debemos primero utilizar el comando

```
git branch nombre_branch
```

Donde el nombre usualmente es la funcionalidad que se va implementar.

Necesitamos al menos un commit para poder crear branches

Cambiando el branch

Para cambiar el branch utilizaremos el comando

```
git checkout nombre_branch
```

También podemos crear un nuevo branch y cambiarnos directamente pasando el argumento -b

```
git checkout -b nuevo_branch
```

Merge

Podemos unir una rama utilizando la instrucción git merge, lo importante aquí es saber en que rama se va a converger, por lo usual es la contraria a la que se está trabajando.

Ejemplo: Diego y Camila están trabajando en development, ahora Camila quiere implementar una funcionalidad nueva, entonces hace la rama funciónX, luego de implementar la funcionalidad ahora quiere hacer el merge, entonces tiene que volver a la rama development y ahí hacer el merge.

Manejando los errores

Si dos personas modifican la misma línea del mismo archivo o una persona modifica la misma línea del mismo archivo en dos branches distintos, entonces la hacer un merge se obtendrá un conflicto.

La persona que hace el merge es la encargada de resolver el conflicto, para hacerlo debe escoger cual de las dos versiones queda, y borrar la otra, luego hacer un commit con el conflicto resuetlo.

Supongamos que estamos trabajando en un archivo html, luego de hacer el branch modificamos tanto el master como el branch, en los dos agregamos contenido dentro de un div footer

branch master

```
<div id="footer">
    contacto : email@misitio.com
</div>
```

otro branch

```
<div id="footer">
    Para contactarnos, escríbanos a
    contacto@misitio.com
</div>
```

Al hacer el merge sobre master obtendremos:

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.htm
Automatic merge failed; fix conflicts and then
```

al abrir el archivo index.html veremos que el conflicto luce así:

```
<div id="footer">
<<<<<<< HEAD
    contacto : email@misitio.com
=======
  Para contactarnos, escríbanos a
  contacto@misitio.com
>>>>>>> iss53
</div>
```

Borrando el branch

Podemos borrar un branch que ya esté mergeado con:

```
git branch -d branch_a_borrar
```

Si por algún motivo queremos borrar un branch que todavía no se la haya hecho merge debemos ocupar el argument -D (d mayúsculas)

```
git branch -D branch_a_borrar
```

Esto sólo sucede si comenzamos a implementar una funcionalidad y en el camino nos damos cuenta de que lo hicimos mal o ya no es necesaria.

Actividad 2

- Generar un branch branch-1 en su repositorio y cambiarse a éste
- Hacer un cambio en un archivo y commitear
- Cambiarse a master y crear un branch branch-2 y cambiarse a éste
- Hacer cambios (ojalá varios commits)
- Cambiarse a master y hacer el merge de branch-1
- Hacer el merge de branch-2 y resolver los conflictos que aparezcan

diff

git diff permite mostrar los cambios entre dos versiones, cuando se utiliza sin argumentos muestra la diferencia entre el índice (staged) y el working directory, en otras palabras muestra los cambios que has hecho desde el último add en cada uno de los archivos.

```
git diff
```

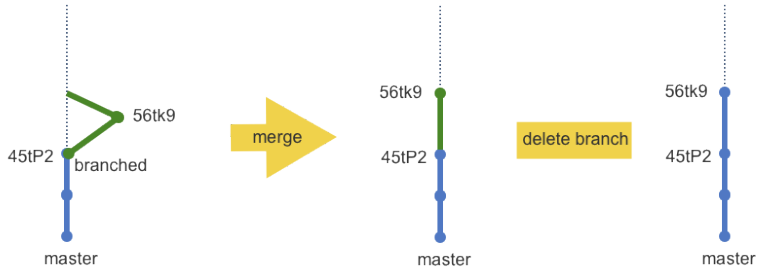
También es posible utilizarlo en conjunto con un checksum para mostrar todas las diferencia entre la versión actual y una específica.

Para eso, git log, escogemos una versión por su checksum y luego

```
git diff df237e6dcf70b7ffae2879c8fafd530fd9e9f810
```

Fast Foward

Un fast forward ocurre cuando haces un merge sobre un punto que no se ha modificado desde la ramificación, de esta forma es como si todos los cambios se hubieses implementado sobre la rama original.



Rebase

Existe otra forma de unir branches y es a través de rebase, este comando modifica la historia de los commits y es peligrosa, sin embargo puede generar un historial de commits más limpios que rebase, más información en: https://git-scm.com/docs/git-rebase

Preguntas

1. ¿Para qué sirven los branches en git?
2. ¿Qué son los fast fowards?
3. ¿Cómo se hace un merge?
4. ¿Cómo se cambia de rama?
5. ¿Cómo se borra un branch que ya ha sido mergeado?
6. ¿Cómo se borra un branch que no ha sido mergeado?

7. ¿Cuál es la diferencia entre un fast forward y un rebase?

Resumen comandos útiles

Comando	Explicación
git init	Inicializa el repositorio git
git add archivo	Agrega el archivo a staging
git add .	Agrega todos los archivos a staging
git commit -m “ ”	Confirma el set de cambios
git commit -m “ ” -amend	Corrige el último set de cambios hechos
git status	Muestra el estado de los archivos
git log	Muestra todos los cambios confirmados
git reset –soft HEAD~	Cancela el último commit y vuelve a staging
git reset –hard HEAD~	Cancela el último commit y saca los archivos de staging
git branch	Muestra los branches disponibles
git branch -d nombre_branch	Borra el branch nombrado
git branch -D nombre_branch	Borra el branch aunque no haya hecho merge

Otros tutoriales:
https://try.github.io/