

GIT

CLASE 2 : RAMAS

TRABAJANDO CON CLIENTES DE GIT

Cómo hemos visto en la clase anterior, podemos trabajar con clientes de **GIT** tales como GitHub, configurarlo como nuevo remoto y empezar a compartir nuestro trabajo con otros desarrolladores. Veamos ahora el caso en donde queremos empezar a colaborar con alguien que ya tiene un proyecto subido a algún cliente y donde no tenemos la necesidad de comenzar un repositorio nuevo vacío en nuestra máquina local.

EL COMANDO CLONE

Si queremos obtener una copia de un repositorio **GIT** existente vamos a necesitar el comando *git clone*. Si estamos familiarizados con otros sistemas VCS como Subversion, podemos notar que el comando es “clone” y no “checkout”.

Cada versión de cada archivo del historial del proyecto es traída por defecto cada vez que ejecutamos este comando. De hecho, si el disco de nuestro servidor se corrompe, podemos utilizar básicamente cualquiera de los clones para restaurar nuestro trabajo.

Podemos entonces clonar un repositorio de la siguiente manera :

```
> git clone <url>
```

Esto crea un directorio llamado de la misma manera que el repositorio del cliente, inicializa un directorio .git dentro del mismo y nos trae toda la información de ese repositorio mostrándonos la última versión del mismo.

Alternativamente podemos elegir el nombre del directorio que queremos crear con el siguiente comando :

```
> git clone <url> <nombre_del_directorio_nuevo>
```

Una vez que un repositorio es clonado, **GIT** crea punteros que le dan seguimiento a cada branch remoto dentro del repositorio local clonado (las cuales son visibles con `git branch -r`) y crea una rama inicial posicionandones en ella.

CLONE VS FORK

Cuando queremos contribuir con un proyecto existente al cual no tenemos permisos de escritura(push access) podemos realizar un “fork” de dicho proyecto. Cuando realizamos un fork de un repositorio, GitHub intentará hacer una copia entera del proyecto en nuestra cuenta de usuario; vive en nuestro nombre de espacio, es decir que podemos realizar cualquier cambio en él.

De esta manera, los proyectos no tienen que preocuparse por agregar usuarios como colaboradores ni darles permisos especiales. Cualquier persona puede entonces hacer un fork de cualquier proyecto open-source, hacerle cambios y contribuir con esos cambios al proyecto original creando lo que se denomina un “Pull Request”.

“FORKEANDO” UN REPOSITORIO

Para crear un fork de un repositorio, solo tenemos que visitar la página del proyecto original y hacer click en el botón que dice “Fork” en el margen superior derecho del sitio :



Luego de unos instantes, vamos a ser redirigidos a la página de nuestro nuevo proyecto el cual contiene una copia exacta del original a la cual podemos hacerle cambios.

DESARROLLO COLABORATIVO

GitHub está diseñado alrededor de un flujo de trabajo focalizado en Pull Requests, lo cual le permite a los desarrolladores a colaborar tanto en repositorios unificados por grupo como en aquellos que son distribuidos globalmente a través de compañías o incluso personas desconocidas.

Por lo general esto funciona de la siguiente manera :

1. Hacemos un Fork de un proyecto
2. Creamos un branch nuevo desde el branch *master*
3. Hacemos algunos commits para mejorar el proyecto
4. Hacemos un push de nuestro branch a nuestro proyecto de GitHub
5. Abrimos un Pull Request en GitHub
6. Generamos una discusión al respecto de nuestro trabajo realizado y alternativamente seguimos haciendo más commits
7. Sincronizamos el branch *master* nuevamente a nuestro Fork

EL COMANDO FETCH

Como hemos mencionado anteriormente, el comando *git clone* agrega implícitamente un remoto *origin* en nuestro repositorio local. Veamos el caso en donde estamos trabajando con más desarrolladores y los mismos estuvieron realizando cambios a nuestro proyecto remoto. En este caso, si queremos traer toda la información de los cambios hechos por alguien que actualmente no tenemos en nuestro repositorio local podemos usar el siguiente comando :

```
> git fetch <alias_remoto>
```

Este comando nos va a traer todas las referencias a objetos commit que estén presentes en el repositorio remoto que no tengamos en el nuestro local, lo cual nos da la posibilidad de poder inspeccionarlo posteriormente o bien, si estamos seguros de que es un trabajo testeado, unirlo con nuestro trabajo actual.

En caso de estar trabajando con un repositorio clonado, simplemente podemos correr el comando **git fetch origin** (ya que como vimos, clonar un repositorio remoto nos crea automáticamente un remoto local con alias “origin”).

Es importante notar que el comando *git fetch* solo descarga la información nueva a nuestro repositorio local pero no junta este nuevo trabajo traído con nuestro trabajo actual local. Para ello tenemos que integrar ambas ramas de trabajo manualmente.

EL COMANDO PULL

Si nuestra rama actual está configurada para apuntar a una rama remota, podemos alternativamente usar el siguiente comando :

```
> git pull
```

A diferencia del comando *git fetch*, el *git pull* va a descargar todos los cambios nuevos que no tengamos en nuestro repositorio local desde el repositorio remoto y va a intentar integrar automáticamente nuestro trabajo local actual con el que acaba de traer.

Esto puede ser de gran ventaja a nuestro flujo de trabajo ya que no dependemos de una integración manual en caso de estar seguros de lo que estemos descargando desde el proyecto remoto.

PULL REQUEST : SUGERENCIAS DE CAMBIOS

Vamos a imaginar por un momento que tenemos un Fork de algún proyecto en nuestra cuenta de usuario y queremos realizar cambios en él y sugerirlos como cambios potenciales dentro del proyecto original.

Los pasos a seguir, desde el inicio, serían :

1. Generamos el Fork del proyecto
2. Clonamos nuestro fork localmente
3. Creamos una rama de trabajo nueva
4. Realizamos cambios del código
5. Verificamos que los cambios estén bien y hayan sido testeados
6. Confirmamos nuestros cambios con un commit
7. Realizamos un push de dicha rama al Fork que tenemos en GitHub

CREANDO NUESTRO PRIMER PULL REQUEST

Una vez dentro de GitHub podemos ver como la página reconoce que un nuevo branch fue creado y subido, y además va a aparecer con un gran botón verde el cual nos va a permitir generar nuestro primer Pull Request :

Example file to blink the LED on an Arduino — Edit

2 commits 2 branches 0 releases 1 contributor

Your recently pushed branches:

slow-blink (less than a minute ago) Compare & pull request

branch: master ↗ blink / +

This branch is even with schacon:master

Create README.md

schacon authored on Jun 12 latest commit bbc80f9b29

README.md Create README.md 4 months ago

blink.ino my arduino blinking code (from arduino.cc) 4 months ago

README.md

Blink

This repository has an example file to blink the LED on an Arduino board.

Code Pull Requests 0 Wiki Pulse Graphs Settings HTTPS clone URL https://github.com/... You can clone with HTTPS, SSH, or Subversion. Clone in Desktop Download ZIP

Si le hacemos click al botón, vamos a poder ver una pantalla que nos va a pedir un título y una descripción de nuestro Pull Request. Usualmente vale la pena esforzarnos en pensar ambos conceptos ya que va a ser de mucha ayuda para el dueño del proyecto original a determinar qué cambios estuvimos haciendo y si los tiene que aceptar o no.

También podemos ver una lista de uno o más commits en nuestro branch que están por sobre ("ahead") el branch *master* y un visor de diferencias unificadas que van a tomar efectos en caso de ser aceptado nuestro trabajo.

The screenshot shows a GitHub pull request interface. At the top, it displays the repository name "tonychacon / blink" and a note that it's "forked from schacon/blink". To the right are buttons for "Unwatch", "Star", "Fork", and a "Create pull request" button. The main area shows a diff between the "schacon:master" branch and the "tonychacon:slow-blink" branch. The commit message is "Three seconds is better" and the commit body contains the text "Studies have shown that 3 seconds is a far better LED delay than 1 second." followed by a link: "http://studies.example.com/optimal-led-delays.html". A sidebar indicates that the branches are "Able to merge". Below the commit details, there are statistics: 1 commit, 1 file changed, 0 commit comments, and 1 contributor (tonychacon). The commit itself is dated "Commits on Oct 01, 2014" and has a hash "db44c53". The bottom section shows the diff for the file "blink.ino", highlighting changes in lines 18, 21, 22, and 23. The "Unified" view is selected.

Cuando hagamos click en el botón que dice “Create pull request” en la pantalla, el dueño del proyecto del cual hicimos el Fork va a recibir una notificación de que alguien está sugiriendo cambios y va a poder acceder a toda la información de los cambios propuestos por nosotros.

En este punto, el dueño del proyecto puede ver nuestros cambios sugeridos, integrarlos con el repositorio original, rechazarlos o realizar comentarios sobre ellos. Mirando en la vista de diferencias, el dueño del proyecto puede hacer click en cualquier línea de cambio propuesta para dejar un comentario sobre alguna de ellas.

The screenshot shows a GitHub Pull Request page. At the top, it says "Three seconds is better #2". Below that, it shows a merge commit from "tonychacon" into "schacon:master" from "tonychacon:slow-blink". The main area shows a diff of the "blink.ino" file. The diff highlights changes in lines 18, 21, and 22. A comment from Scott Chacon is shown in the interface:

```

I believe it would be better if the light was off for 4 seconds and on for just 3.

```

Una vez que este cambio fue comentado, la persona que abrió el Pull Request (y cualquier otra persona que le esté dando seguimiento al repositorio) va a recibir una notificación. Si tenemos configurada nuestra cuenta para poder recibir notificaciones por email, podríamos ver un correo similar al siguiente :

The screenshot shows an email inbox with a reply to a GitHub Pull Request. The subject is "Re: [blink] Three seconds is better (#2)". The message is from Scott Chacon <notifications@github.com> and was sent at 10:55 AM (18 minutes ago). The message content includes the GitHub comment and the original code diff:

```

In blink.ino:

>   digitalWrite(led, LOW);      // turn the LED off by making the voltage LOW
> - delay(1000);                // wait for a second
> + delay(3000);                // wait for a second

I believe it would be better if the light was off for 4 seconds and on for just 3.

—
Reply to this email directly or view it on GitHub.

```

Cualquier persona, además, puede dejar comentarios en generales en el Pull Request.

Ahora la persona que está contribuyendo al proyecto original simplemente puede realizar más commits en el branch en cuestión y volver a hacer un push, lo cual va a actualizar automáticamente el Pull Request. Si afectamos líneas de código que habían sido previamente comentadas por alguien, las mismas se verán colapsadas en la página de la discusión del Pull Request.

Agregar commits a un Pull Request existente no activa una notificación, entonces una vez que hayamos hecho cambios o corregido nuestro trabajo, lo ideal es dejar un nuevo comentario para informar al dueño del proyecto sobre los mismos.

Three seconds is better #2

The screenshot shows a GitHub Pull Request interface. At the top, it says "tonychacon wants to merge 3 commits into `schacon:master` from `tonychacon:slow-blink`". Below this are three tabs: "Conversation 3", "Commits 3", and "Files changed 1".

Comment by tonychacon: commented 11 minutes ago
Studies have shown that 3 seconds is a far better LED delay than 1 second.
<http://studies.example.com/optimal-led-delays.html>

Commit: `three seconds is better` db44c53

Comment by schacon: commented on an outdated diff 5 minutes ago
Show outdated diff

Comment by schacon: commented 5 minutes ago
Owner If you make that change, I'll be happy to merge this.

Commit: `longer off time` 0c1f66f
`remove trailing whitespace` ef4725c

Comment by tonychacon: added some commits 2 minutes ago

Comment by tonychacon: commented 10 seconds ago
I changed it to 4 seconds and also removed some trailing whitespace that I found. Anything else you would like me to do?

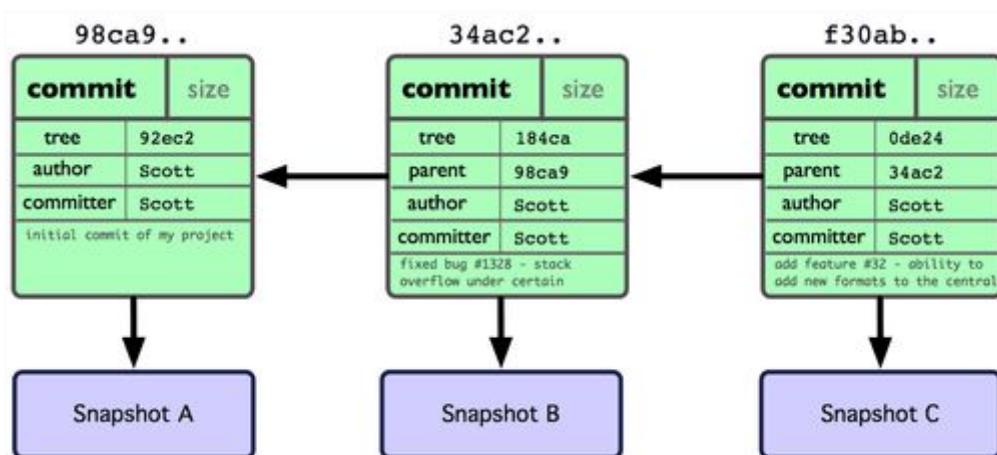
Info box: This pull request can be automatically merged. You can also merge branches on the command line. Merge pull request

Algo interesante de notar es que GitHub verifica que el Pull Request pueda ser integrado de manera limpia y sin problemas; en dicho caso, va a mostrar un botón de atajo para realizar esta operación.

FLUJO DE TRABAJO : RAMAS

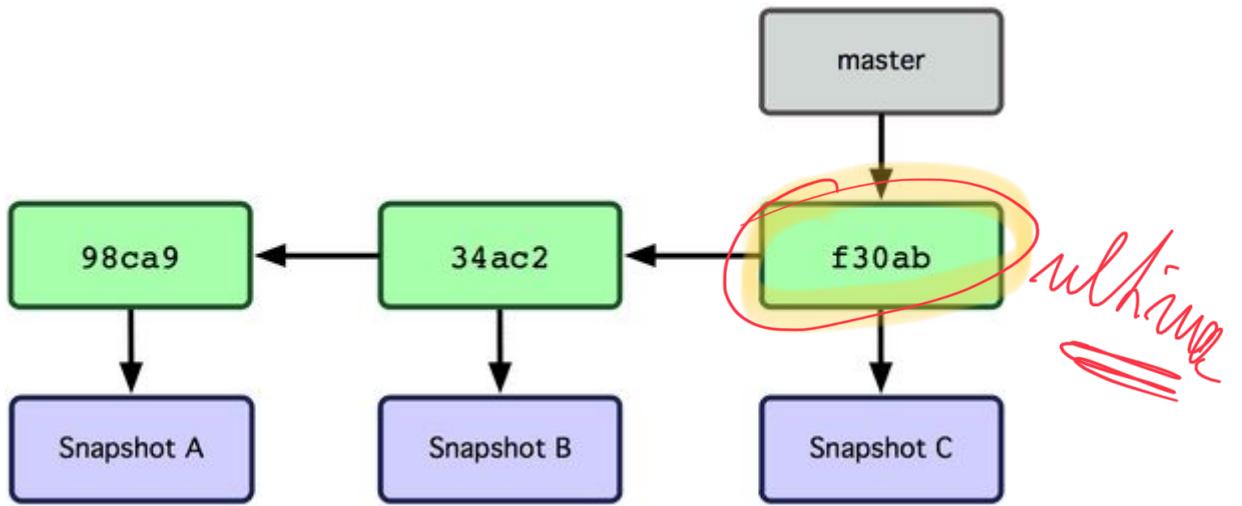
Para entender realmente cómo ramifica Git, previamente hemos de examinar la forma en que almacena sus datos. Recordando lo citado en el capítulo 1, Git no los almacena de forma incremental (guardando sólo diferencias/deltas), sino que los almacena como una serie de snapshots (copias puntuales de los archivos completos, tal y como se encuentran en ese momento).

En cada confirmación de cambios (commit), 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, y uno o varios apuntadores a las confirmaciones (commit) que sean padres directos de esta (un parent en los casos de confirmación normal, y múltiples padres en los casos de estar confirmando una fusión (merge) de dos o más ramas).



QUÉ ES UN BRANCH ?

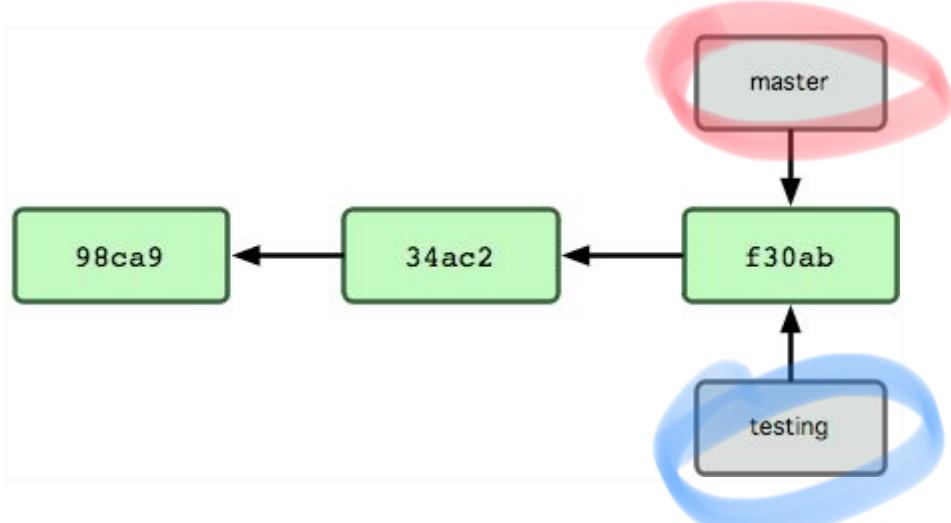
Un branch o rama en **GIT** es simplemente un apuntador móvil apuntando a una de esas confirmaciones. La rama por defecto de Git es la rama master. Con la primera confirmación de cambios que realicemos, se creará esta rama principal master apuntando a dicha confirmación. En cada confirmación de cambios que realicemos, la rama irá avanzando automáticamente. Y la rama master apuntará siempre a la última confirmación realizada.



¿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`:

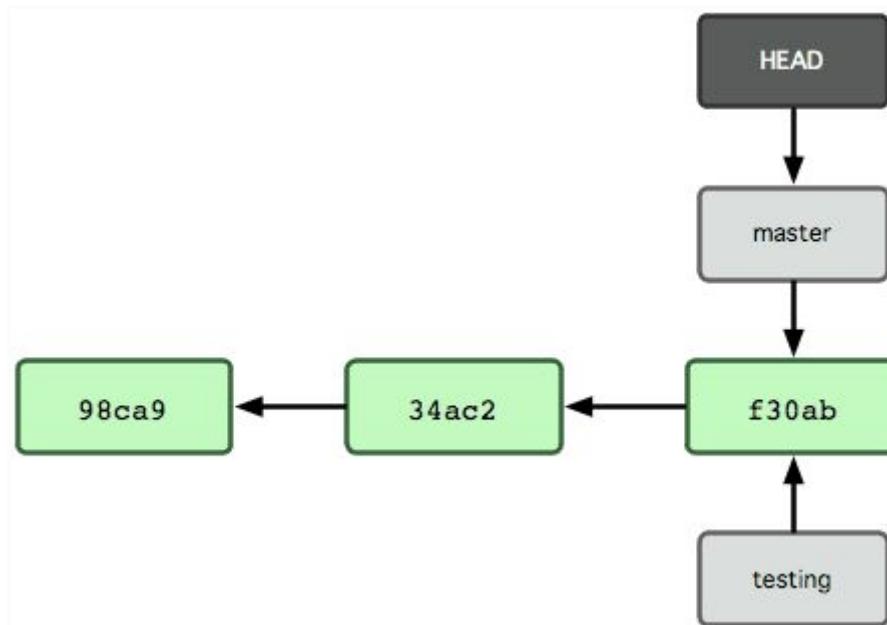
`git branch testing`

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



EL ARCHIVO HEAD

¿cómo sabe **GIT** en qué rama estás en este momento? Mediante un apuntador especial denominado HEAD. Aunque es preciso comentar que este HEAD es totalmente distinto al concepto de HEAD en otros sistemas de control de cambios como Subversion o CVS. En **GIT**, 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.



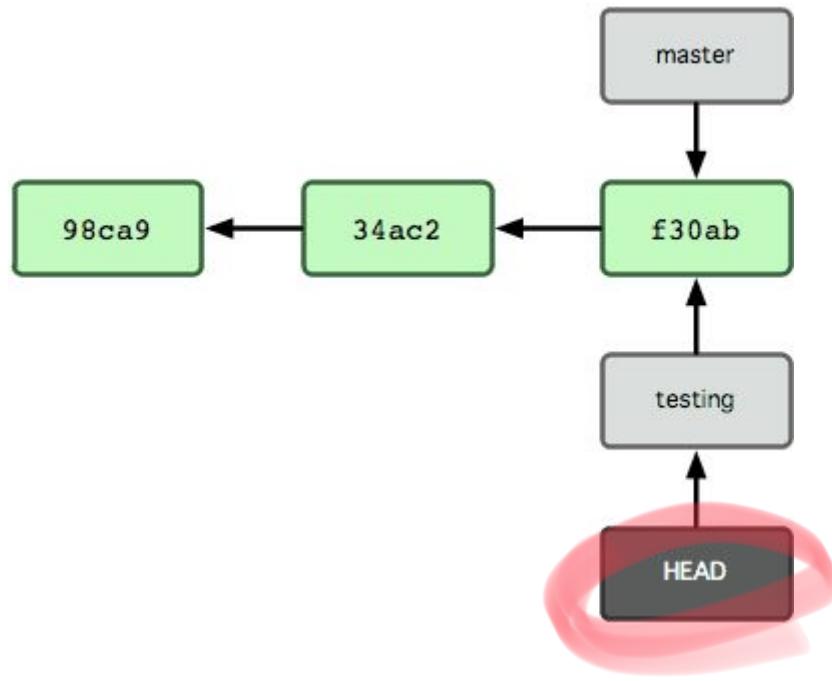
Switch EL COMANDO ~~CHECKOUT~~

Para saltar de una rama a otra, tienes que utilizar el comando *git checkout*. Hagamos una prueba, saltando a la rama *testing* recién creada:

```
git checkout testing
```

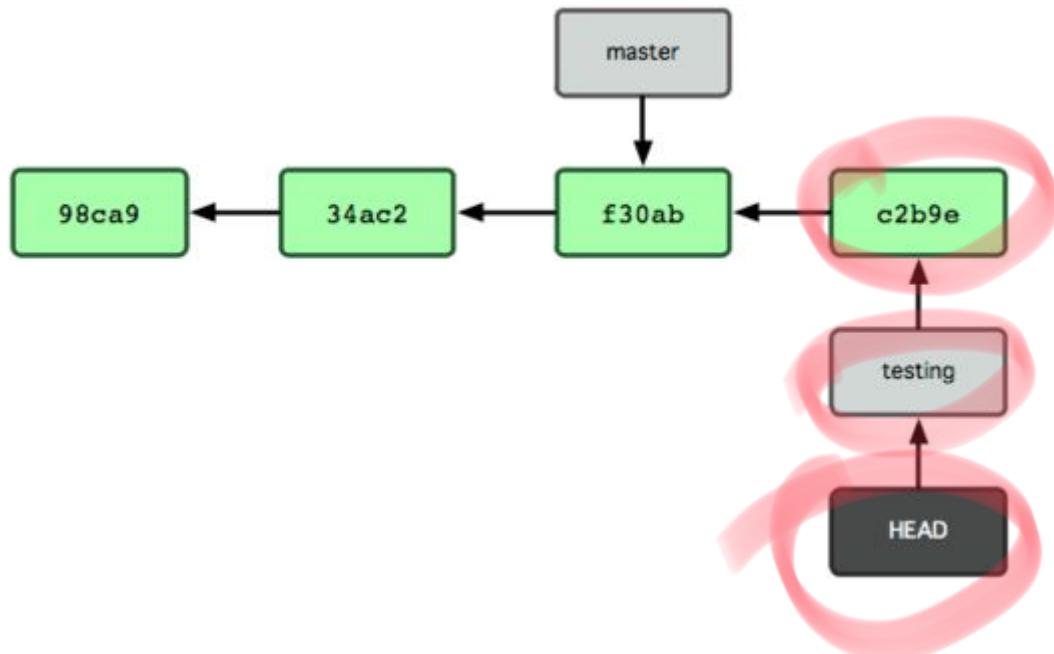
Switch

Esto mueve el apuntador HEAD a la rama *testing*:



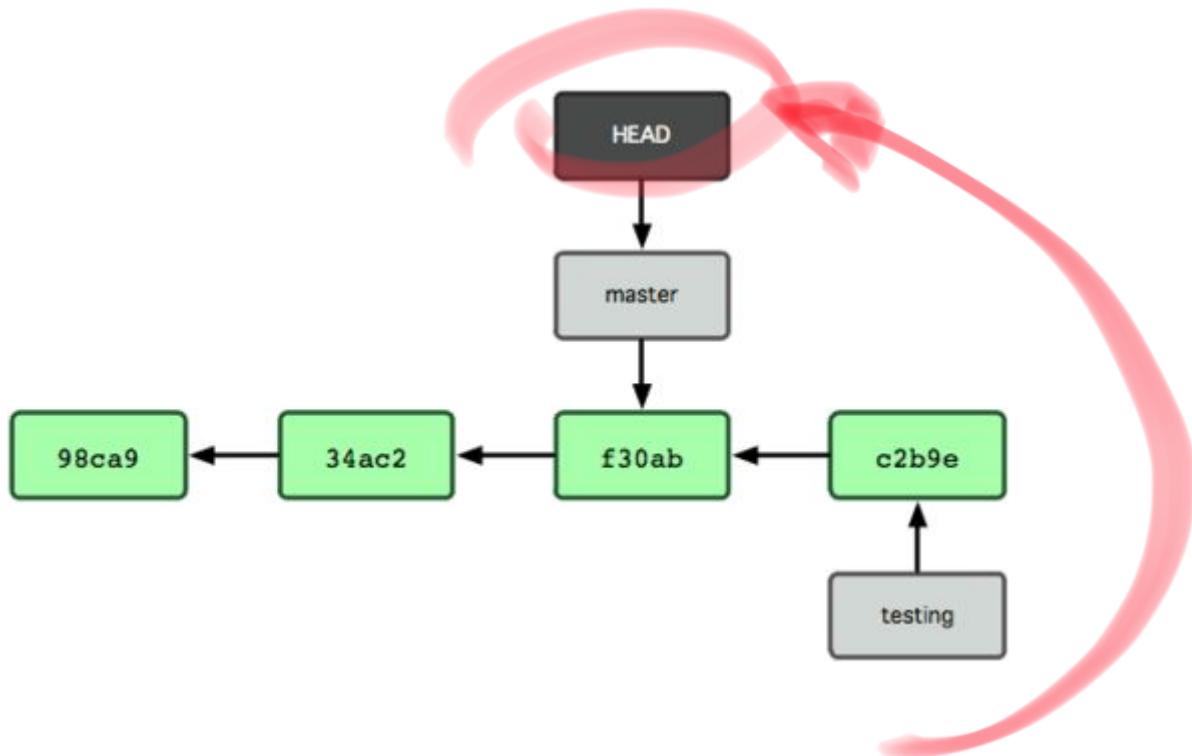
¿Cuál es el significado de todo esto? Lo veremos tras realizar otra confirmación de cambios:

```
git commit -a -m 'made a change'
```



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
```



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 divergirán 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.

ESTRATEGIAS DE UNIFICACIÓN : INTEGRANDO CAMBIOS

Trabajando con **GIT** y sus branches podemos observar fácilmente que el branch de una nueva funcionalidad siempre nos va a quedar por “encima” del branch *master*, el cual contiene nuestro código estable, aquel que ya pasó por las pruebas necesarias para

confirmar que el mismo es válido y puede estar en producción; esto es en nuestro servidor en el caso que estemos desarrollando una aplicación web por ejemplo.

Cada vez que estemos trabajando sobre alguno de estos branches de funcionalidades y hayamos podido testearlos y confirmado su aporte al proyecto, usualmente vamos a querer integrarlos a nuestro código oficial. Para esto **GIT** nos ofrece varias herramientas de integración, unificación, fusión o reubicación de trabajo.

EL COMANDO MERGE

Si solamente queremos incorporar los cambios nuevos que hayamos ya pasado a través de pruebas oportunas, podemos utilizar el comando *git merge* :

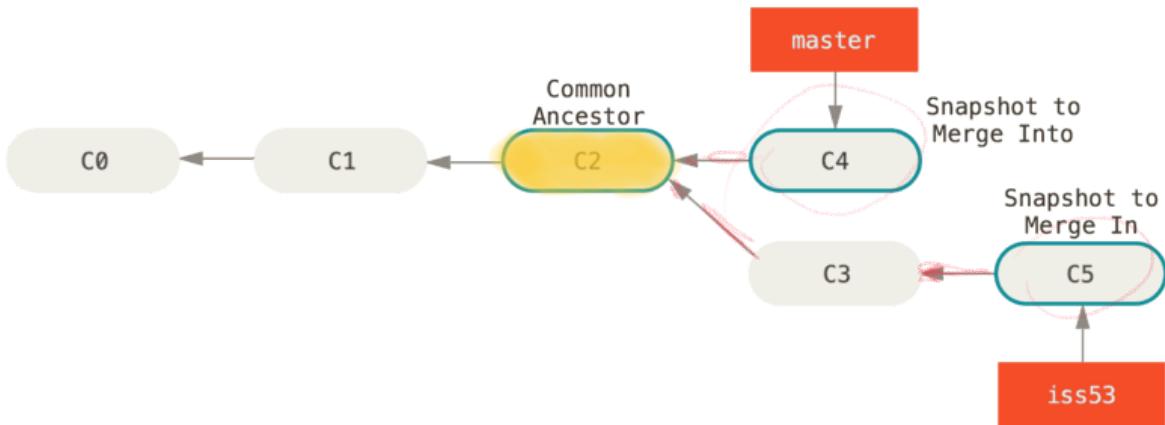
```
> git merge <branch>
```

Dependiendo cómo esté distribuido nuestro repositorio local, en muchos casos podremos notar la frase “Fast forward” aparecer en la salida del comando.

GIT ha movido el apuntador hacia adelante, ya que la confirmación apuntada en la rama donde has fusionado estaba directamente arriba respecto a la confirmación actual. Dicho de otro modo: cuando intentas fusionar una confirmación con otra confirmación accesible siguiendo directamente el historial de la primera; **GIT** simplifica las cosas avanzando el puntero, ya que no hay ningún otro trabajo divergente a fusionar. Esto es lo que se denomina “avance rápido” (“fast forward”).

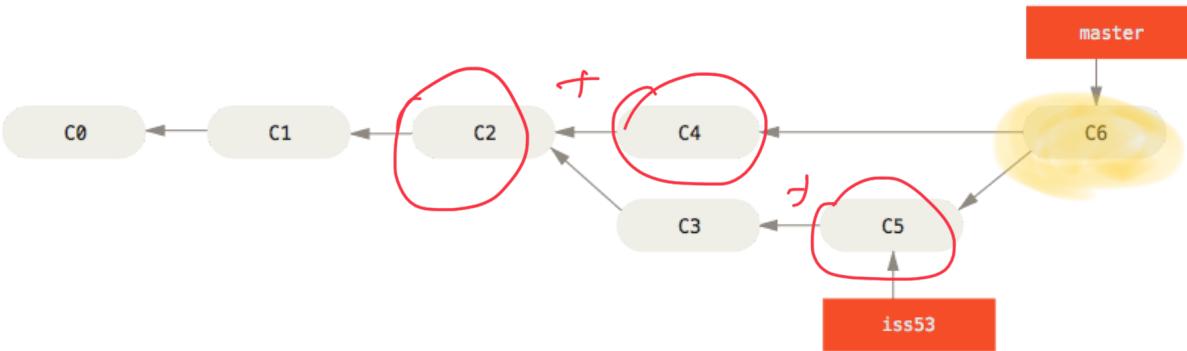
Ahora, los cambios realizados están ya en el snapshot del commit apuntado por el branch master.

Supongamos ahora que el branch que queremos fusionar con nuestro trabajo actual no es lineal, es decir, no tiene como ancestro a nuestro branch master. En este caso, los branches se verán bifurcados respecto a algún ancestro en común probablemente :



Debido a que la confirmación en la rama actual no es ancestro directo de la rama que pretendes fusionar, Git tiene cierto trabajo extra que hacer. Git realizará 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.

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 su particularidad es que tiene más de un padre.



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 pretendes fusionar, Git no será capaz de fusionarlas directamente. Por ejemplo, si en tu trabajo del problema #53 has modificado una misma porción que también ha sido modificada en el problema hotfix, verás un conflicto como este:

```
> git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git no crea automáticamente una nueva fusión confirmada (merge commit), sino que hace una pausa en el proceso, esperando a que tú resuelvas el conflicto. 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
You have unmerged paths.
  (fix conflicts and run "git commit")

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 aquello que sea conflictivo y no se haya podido resolver, se marca como "sin fusionar" (unmerged). Git añade a los archivos conflictivos unos marcadores especiales de resolución de conflictos que te guiarán cuando abras manualmente los archivos implicados y los edites para corregirlos. 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, 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 iss53 contiene el resto, lo indicado en la parte inferior del bloque. Para resolver el conflicto, has de elegir manualmente el contenido de uno o de otro lado. Por ejemplo, puedes optar por cambiar el bloque, dejándolo así:

```
<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 marcar cada archivo modificado. Marcar archivos como preparados (staged) indica a Git que sus conflictos han sido resueltos.