Git

Antonio Espín Herranz

Contenidos

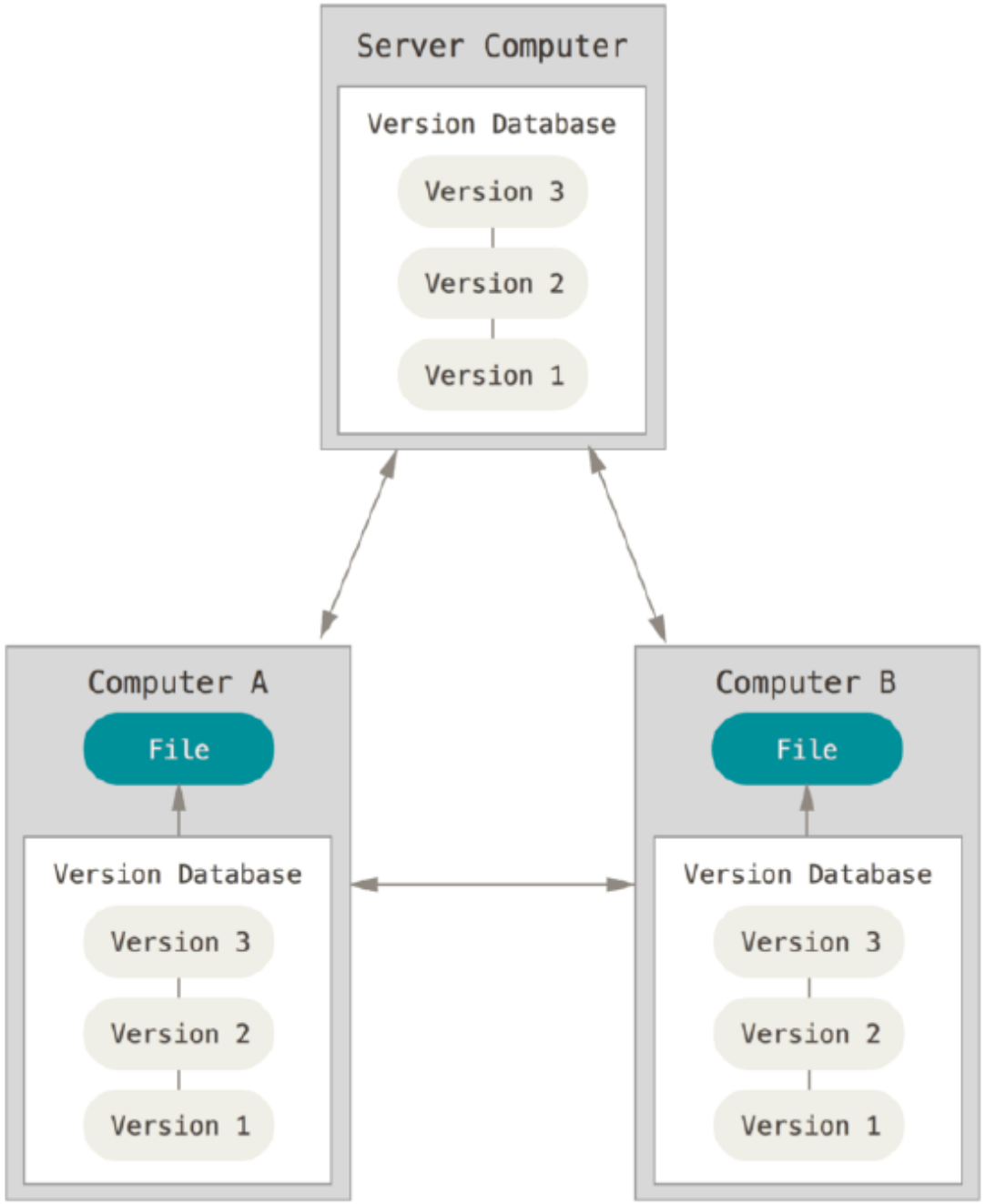
- ¿Qué es Git?
- Instalación en Ubuntu
- Configuración en Ubuntu
- Repositorios
- Trabajo con Git

¿Qué es Git?

- Git es un sistema de control de versiones.
- Un control de versiones (VCS) es un sistema que registra los cambios en un archivo o conjunto de archivos (por ejemplo, un proyecto) a lo largo del tiempo, de modo que podamos recuperar versiones específicas mas adelante.
- Los VCS pueden ser: locales, centrales y distribuidos.
- Git es un VCS distribuido.

¿Qué es Git?

- Control de versiones distribuido.
- Los clientes descargan la última versión de los archivos.
- Se replica en un repositorio local.
- Si el servidor central falla se puede restaurar el repositorio con cualquiera de los clientes.



Instalación en Ubuntu

- Para instalar Git, desde un terminal lanzar el comando:

- **sudo apt install git**
  - Pedirá la contraseña de root

- Luego comprobar la versión instalada:
  - **git --version** 2.17.1

- Puede surgir problemas al instalar:
  - Por ejemplo, al lanzar el comando, obtenemos el error:
  - **Could not get lock /var/lib/dpkg/lock**
  - Este error suele dar cuando se ha quedado algún proceso de instalación a la mitad.

Solución al error anterior

- Podemos comprobar si hay algún proceso que bloquea a apt con el comando: **ps aux | grep –i apt**

- Lanzar estos 4 comandos:
  - **sudo fuser –vki /var/lib/dpkg/lock**
  - **sudo rm –f /var/lib/dpkg/lock**
  - **sudo dpkg --configure –a**
  - **sudo apt-get autoremove**

- Una vez solucionado: **sudo apt install git**
- Comprobar si se ha instalado correctamente: **git --version**

Configuración en Ubuntu

- Git dispone del comando **git config**
- La configuración sólo se realiza la primera vez cuando se instala git y ya no es necesaria hasta que la queramos cambiar.

- Tenemos 3 niveles del mas general al más específico, siempre cuenta mas el más específico:
  - Para todos los usuarios del sistema.
  - **Opción --system**
    - Archivo: /etc/gitconfig

  - Para el usuario específico:
  - **Opción --global**
    - Archivo: ~/.gitconfig o ~/.config/git/config

  - Para el repositorio actual (necesitamos previamente un repositorio).
  - **Opción --local**
    - Archivo: .git/config

Configuración en Ubuntu

- Para añadir configuración a nivel de usuario:
  - **git config --global user.name “Nombre Apellido”**
  - **git config --global user.email un_email**

- Almacenar esta información para git es importante. Al menos, estos dos campos, se utilizarán para las confirmaciones.

- También necesitamos añadir a la configuración de git un editor de texto. Normalmente se utiliza **vim**.
  - **git config --global core.editor vim**

- Si no está instalado: **sudo apt install vim**

Configuración en Ubuntu

- Dentro de **vim** para insertar
  - **:i** Insertar texto
  - **:q** Salir sin grabar
  - **:w** Escribir el fichero.

- Comprobar las variables configuradas:
  - **git config --list**

- Para ver una variable en concreto:
  - **git config user.email**

- Para obtener ayuda:
  - **git help config**

Repositorios

- Repositorio en una **carpeta vacía**.
  - En una carpeta vacía lanzar el comando: **git init**.
  - Crea un repositorio vacío con una carpeta: **.git (esqueleto vacío).**

- Repositorio en una **carpeta con un proyecto**.
  - Dentro de la carpeta del proyecto de C++, lanzar el comando git init
  - Añadimos los ficheros .cpp y .h a git hacemos un commit
    - **git add *.cpp**
    - **git add *.h**
    - **git commit -m 'versión inicial del proyecto'**

- **Clonar un repositorio existente en una URL**:
  - **git clone https://github.com/libgit2/libgit2 [nombre_de_carpeta]**
  - Adicionalmente se puede añadir un nombre de carpeta.
  - Esta opción crea un subdirectorio para almacenar todos los ficheros descargados.

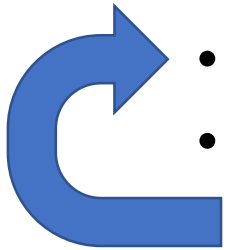
Estados de un archivo

- Una vez los archivos están en el repositorio se pueden cambiar y confirmar los cambios.

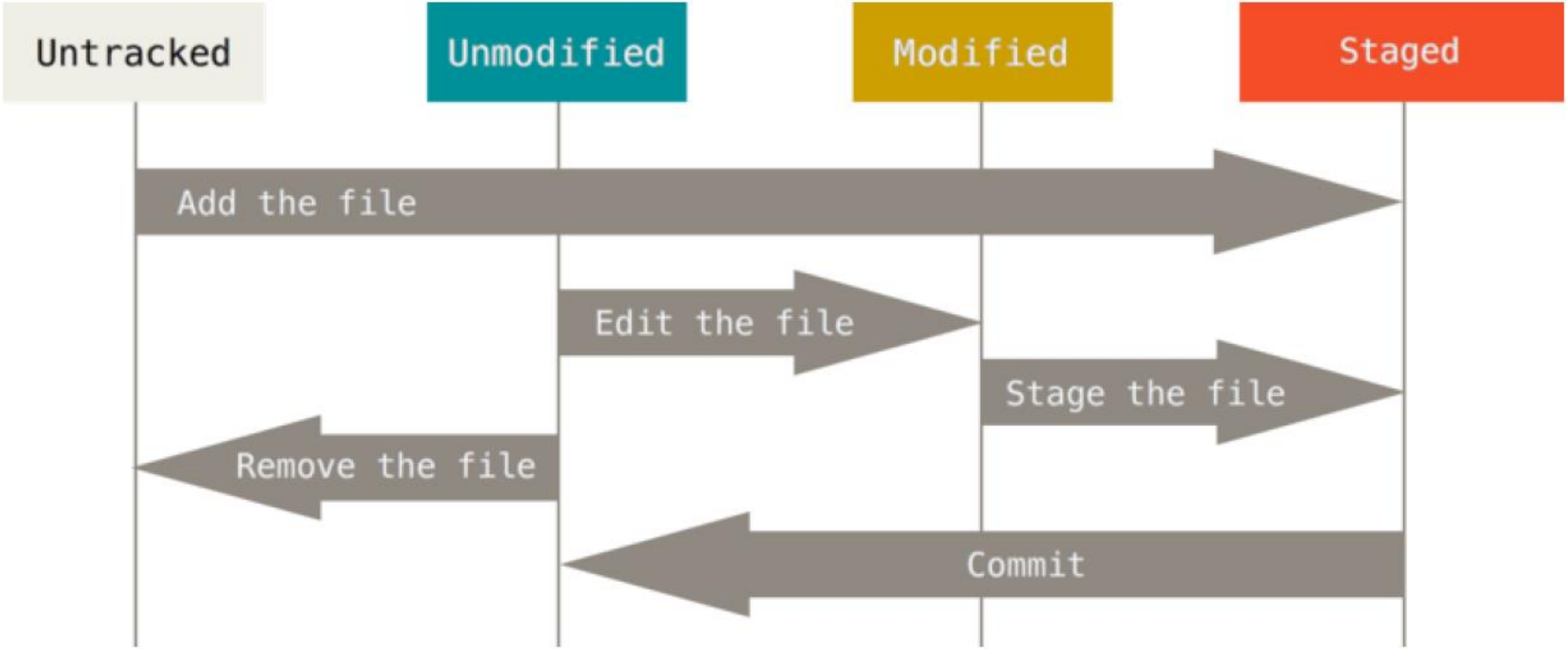
- Hay 2 tipos de archivos:
  - Rastreados:
    - Archivos sin modificar, modificados o preparados.
  - Sin rastrear:
    - Son todos los demás

- Cuando se editan archivos Git los ve como modificados, porque han cambiado desde su último commit.

- El ciclo de trabajo es:
  - Modificar
  - Preparar
  - Confirmar



Ciclo de vida



Estado de los archivos

- Ver el estado de los archivos (después de clonar el repositorio)
- **git status**
- Añadir un nuevo archivo a la carpeta:
  - echo “Prueba proyecto” > README
- Lanzar **git status** (README estará sin rastrear).
- Es un archivo que estará sin rastrear.
- Para empezar a rastrearlo: **git add README / git status**
- Si modificamos un archivo, después habrá que prepararlo.
- También se prepara con **git add** <Fichero modificado>
- El paso final para confirmar será: **git commit**
- Se confirma todo lo que estaba preparado.
- Se puede lanzar git status -s para mostrar la información mas compacta.

Estado de los archivos

- Se pueden ignorar tipos de archivos en git. (para que no aparezcan ni siquiera como sin rastrear).

- Se puede crear un archivo **.gitignore**, por ejemplo, para no mostrar los archivos **.o, .a, *~** (lo suelen utilizar varios editores).

- cat .ignore
  - *.[oa]
  - *~

Ejemplo

```
# ignora los archivos terminados en .a
*.a

# pero no lib.a, aun cuando había ignorado los archivos terminados en .a en la linea
anterior
!lib.a

# ignora unicamente el archivo TODO de la raiz, no subdir/TODO
/TODO

# ignora todos los archivos del directorio build/
build/

# ignora doc/notes.txt, pero este no doc/server/arch.txt
doc/*.txt

# ignora todos los archivos .txt el directorio doc/
doc/**/*.txt
```

Ver cambios en los ficheros

- A parte de **git status** disponemos de **git diff** que muestra de una forma más precisa y detallada todos los cambios en los archivos.

- Este comando compara lo que tienes en tu directorio de trabajo con lo que esta en el área de preparación.

- El resultado te indica los cambios que has hecho pero que aun no has preparado.

- Para ver lo está preparado y se incluirá en la próxima confirmación:
  - **git diff --staged**

Ver cambios en los ficheros

- Git diff sin parámetros, no se muestran los cambios desde la última confirmación, sólo verás los cambios que aún no están preparados.

- Si se preparan los cambios (git add …) el comando git diff (sin parámetros) no mostrará nada.

- **git diff** sirve también para que estar sin preparar.

Confirmar cambios

- El comando para confirmar es: **git commit**
- OJO, lo que hayamos modificado y no se haya preparado NO se añadirá a la confirmación.
- Cuando se confirman los cambios se abrirá el editor VIM para añadir un mensaje descriptivo de esta confirmación.
- Se puede utilizar la opción **-m**. git commit -m “mensaje ..”

Confirmar cambios

- Se puede saltar el área de preparación. Añadiendo la opción **-a** al comando git commit.

- Con esta opción git prepara automáticamente todos los archivos rastreados.

- Nos ahorramos el paso previo de git add.

Eliminar archivos

- Antes de eliminar un archivo hay que eliminarlo de los archivos rastreados (del área de preparación).
- Después hay que confirmar.
- El comando **git rm** elimina el archivo del directorio de trabajo de manera que no aparezca la próxima vez como archivo no rastreado.
- Se deben eliminar de esta forma.
- Otra opción es eliminarlo de git pero no del disco. Es decir, dejar el archivo sin rastrear: **git rm --cached <<fichero>>**
- Por ejemplo: **git rm --cached README**

Cambiar el nombre de los archivos

- Git dispone del comando: **git mv file_from file_to**

- Ejemplo: git mv README.md README

- Es equivalente a:
  - mv README.md README
  - git rm README.md
  - git add README

Historial de confirmaciones

- Cuando hayamos echo varias confirmaciones se pueden consultar mediante: **git log**

- Se puede utilizar como ejemplo:

```
git clone https://github.com/schacon/simplegit-progit
```

- Tiene varios commit realizados y se pueden consultar.

- Lista las confirmaciones en orden cronológico inverso.

Historial de confirmaciones

- El comando git log se puede utilizar con la opción -p que muestra las diferencias introducidas en cada confirmación: **git log -p**

- Se le puede pasar un número para indicar el número de confirmaciones que queremos mostrar: **git log -2**
- Muestra las dos últimas entradas del historial.

- Se pueden sacar estadísticas con: **git log --stat**

- La opción **--stat** imprime tras cada confirmación una lista de archivos modificados, indicando cuantos han sido modificados y cuantas líneas han sido añadidas y eliminadas para cada uno de ellos, y un resumen de toda esta información.

Historial de confirmaciones

- La salida del comando **git log** se puede formatear con la opción --pretty.

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

- Disponemos de una lista con los posibles parámetros que se pueden pasar.

Parámetros

Table 1. Opciones útiles de `git log --pretty=format`

Opción	Descripción de la salida
`%H`	Hash de la confirmación
`%h`	Hash de la confirmación abreviado
`%T`	Hash del árbol
`%t`	Hash del árbol abreviado
`%P`	Hashes de las confirmaciones padre
`%p`	Hashes de las confirmaciones padre abreviados
`%an`	Nombre del autor
`%ae`	Dirección de correo del autor
`%ad`	Fecha de autoría (el formato respeta la opción `--date`)
`%ar`	Fecha de autoría, relativa
`%cn`	Nombre del confirmador
`%ce`	Dirección de correo del confirmador
`%cd`	Fecha de confirmación
`%cr`	Fecha de confirmación, relativa
`%s`	Asunto

Resumen de opciones de git log

Table 2. Opciones típicas de `git log`

Opción	Descripción
`-p`	Muestra el parche introducido en cada confirmación.
`--stat`	Muestra estadísticas sobre los archivos modificados en cada confirmación.
`--shortstat`	Muestra solamente la línea de resumen de la opción `--stat`.
`--name-only`	Muestra la lista de archivos afectados.
`--name-status`	Muestra la lista de archivos afectados, indicando además si fueron añadidos, modificados o eliminados.
`--abbrev-commit`	Muestra solamente los primeros caracteres de la suma SHA-1, en vez de los 40 caracteres de que se compone.
`--relative-date`	Muestra la fecha en formato relativo (por ejemplo, “2 weeks ago” (“hace 2 semanas”)) en lugar del formato completo.
`--graph`	Muestra un gráfico ASCII con la historia de ramificaciones y uniones.
`--pretty`	Muestra las confirmaciones usando un formato alternativo. Posibles opciones son oneline, short, full, fuller y format (mediante el cual puedes especificar tu propio formato).

Opciones para limitar la salida de git log

Table 3. Opciones para limitar la salida de `git log`

Opción	Descripción
`-(n)`	Muestra solamente las últimas n confirmaciones
`--since, --after`	Muestra aquellas confirmaciones hechas después de la fecha especificada.
`--until, --before`	Muestra aquellas confirmaciones hechas antes de la fecha especificada.
`--author`	Muestra solo aquellas confirmaciones cuyo autor coincide con la cadena especificada.
`--committer`	Muestra solo aquellas confirmaciones cuyo confirmador coincide con la cadena especificada.
`-S`	Muestra solo aquellas confirmaciones que añadan o eliminen código que corresponda con la cadena especificada.

Deshacer Cosas

- Esta es una de las pocas áreas en las que Git puede perder parte de tu trabajo si cometes un error.

- El comando es: **git commit --amend**

- Por ejemplo, confirmamos y se nos había olvidado preparar los cambios de un archivo que queríamos incluir en esta confirmación:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

Trabajar con remotos

- Para poder colaborar en cualquier proyecto Git tenemos que gestionar repositorios remotos.

- Podemos tener varios repositorios remotos:
  - De solo lectura, solo escritura, o lectura y escritura.

- Habrá que realizar operaciones de enviar y traer datos de ellos para compartir nuestro trabajo.

Repositorios remotos

- Disponemos del comando: git remote
- Muestra una lista con los remotos que tenemos especificados.

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

Repositorios remotos

- Se puede utilizar la opción -v
- Comando: **git remote -v**

- Git muestra las URLs que Git ha asociado al nombre y que serán usadas al leer y escribir en ese remoto.

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

Añadir repositorios remotos

- Para añadir repositorios remotos a Git:
- Comando: **git remote add [nombre] [url]**

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb  https://github.com/paulboone/ticgit (fetch)
pb  https://github.com/paulboone/ticgit (push)
```

Añadir repositorios remotos

- Una vez añadido se puede usar el nombre pb en la línea de comandos en lugar de la URL entera.
- Por ejemplo, para traer información que no tenemos en nuestro repositorio: Se lanza el comando: **git fecth pb**

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

Traer y combinar remotos

- Ejecutar **git fetch [nombre-remoto]**
- El comando ira al proyecto remoto y se traera todos los datos que aun no tienes de dicho remoto.
- Luego de hacer esto, tendrás referencias a todas las ramas del remoto, las cuales puedes combinar e inspeccionar cuando quieras.

- Si clonas un repositorio, el comando de clonar automáticamente añade ese repositorio remoto con el nombre “origin”.

- *Por lo tanto, **git fetch origin** se trae todo el trabajo nuevo que ha sido enviado a ese servidor desde que lo clonaste (o desde la ultima vez que trajiste datos). Es importante destacar que el comando **git fetch solo trae datos a tu repositorio local - ni lo combina automáticamente con tu trabajo ni modifica el trabajo que llevas hecho**. La combinación con tu trabajo debes hacerla manualmente cuando estés listo.*

Traer y combinar remotos

- Si has configurado una rama para que rastree una rama remota, puedes usar el comando **git pull para traer y combinar automáticamente la rama remota con tu rama actual.**

- Al ejecutar **git pull** traerás datos del servidor del que clonaste originalmente y se intentara combinar automáticamente la información con el código en el que estas trabajando.

Enviar remotos

- Cuando queremos compartir un proyecto, tenemos que enviarlo al servidor.
- El comando es: **git push [nombre-remoto] [nombre-rama]**
- Por ejemplo, para enviar la rama master al servidor origin (estos nombres se establecen por defecto).
- Por ejemplo: **git push origin master**

Enviar remotos

- El comando: **git push origin master**

- Este comando solo funciona si clonaste de un servidor sobre el que tienes permisos de escritura y si nadie mas ha enviado datos por el medio.
- Si alguien mas clona el mismo repositorio que tu y envía información antes que tu, tu envío será rechazado.
- Tendrás que traerte su trabajo y combinarlo con el tuyo antes de que puedas enviar datos al servidor.

Inspeccionar un remoto

- Para ver información de un remoto:
- Comando: git remote show [nombre-remoto]

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Eliminar y Renombrar Remotos

- Para cambiar el nombre de referencia de un remoto:
- Comando: **git remote rename**

```
$ git remote rename pb paul
$ git remote
origin
paul
```

- Esto **también cambia las ramas remotas**. De pb/master a paul/master.

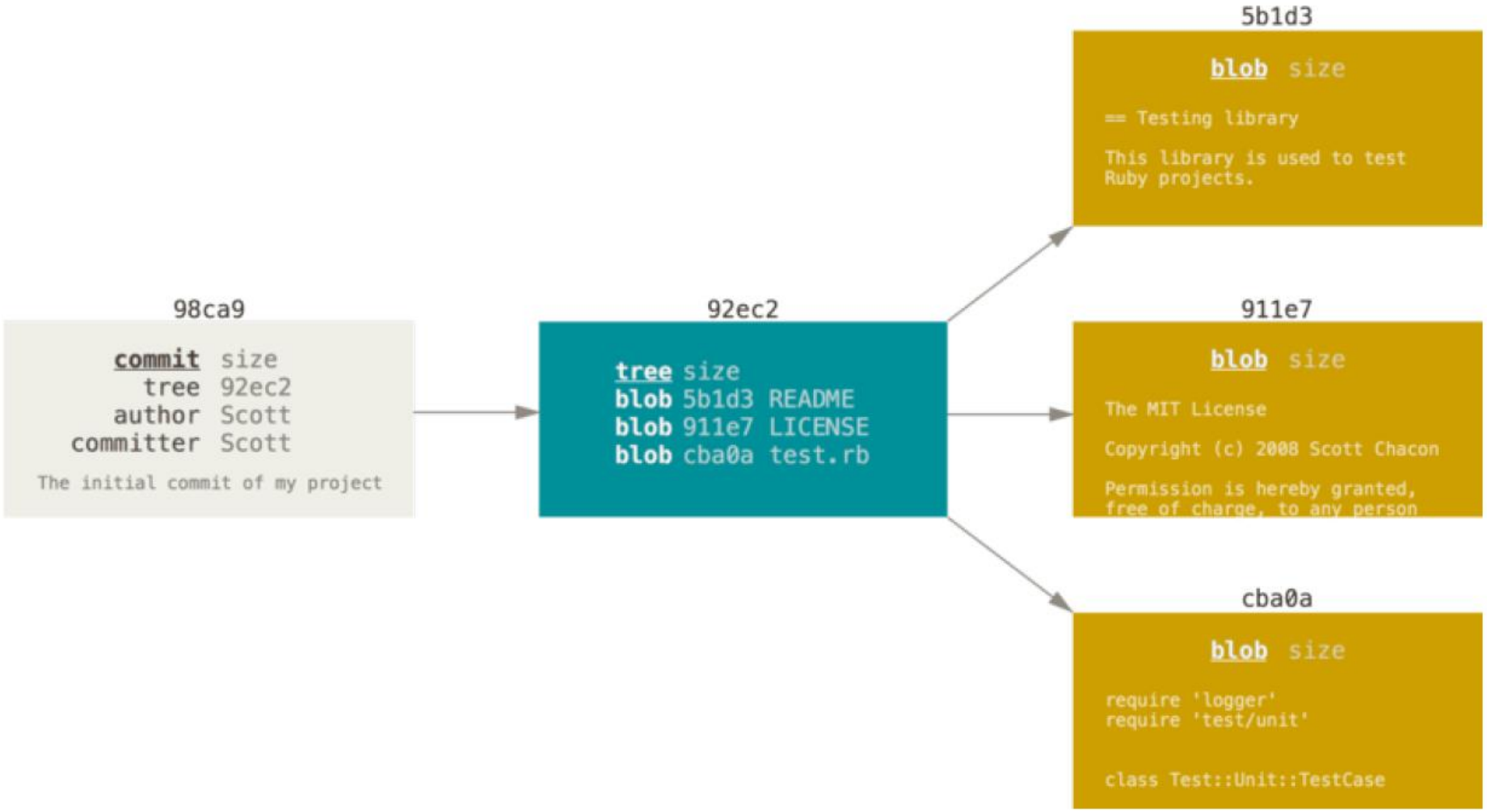
- El comando: **git remote rm** elimina un remoto.

```
$ git remote rm paul
$ git remote
origin
```

Ramificaciones en Git

- Git soporta el uso de ramas.
- La rama principal de desarrollo se llama por defecto, **master**.
- Git maneja las ramificaciones de una forma rápida y eficiente.
- Cuando hacemos un commit y guardamos una versión del proyecto no realiza copias incrementales, si no que guarda una instantánea además de una serie de metadatos.
- Si suponemos que tenemos 3 ficheros en el proyecto y una sola carpeta.
  - Git contendrá 5 objetos:
    - Un blob para cada uno de los 3 archivos.
    - Un árbol con la lista de contenidos del directorio.
    - Y una confirmación de los cambios (commit)

Ejemplo


5b1d3
blob size
== Testing library
This library is used to test
Ruby projects.
98ca9
commit size
tree 92ec2
author Scott
committer Scott
The initial commit of my project
92ec2
tree size
blob 5b1d3 README
blob 911e7 LICENSE
blob cba0a test.rb
911e7
blob size
The MIT License
Copyright (c) 2008 Scott Chacon
Permission is hereby granted,
free of charge, to any person
cba0a
blob size
require 'logger'
require 'test/unit'
class Test::Unit::TestCase

Las ramas

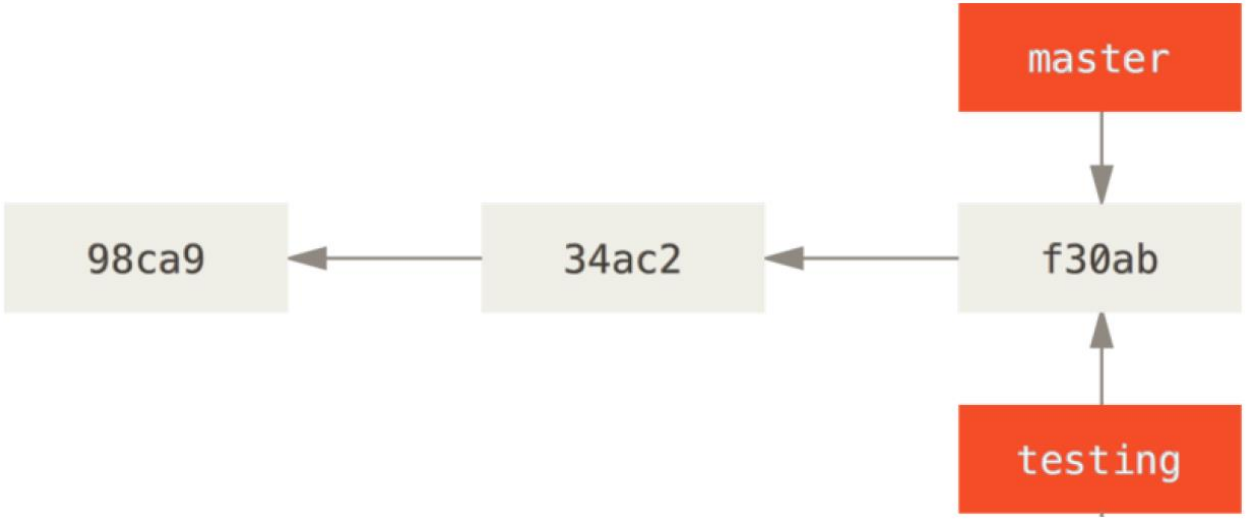
- Una rama Git es simplemente un apuntador móvil apuntando a una de esas confirmaciones (los commit).

- La rama por defecto de Git es la rama master. Con la primera confirmación de cambios que realicemos, se creara esta rama principal master apuntando a dicha confirmación. En cada confirmación de cambios que realicemos, la rama ira avanzando automáticamente.

- La rama "**master**" en Git no es una rama especial. Es como cualquier otra rama. La única razón por la cual aparece en casi todos los repositorios es porque es la que crea por defecto el comando **git init** y la gente no se molesta en cambiarle el nombre.

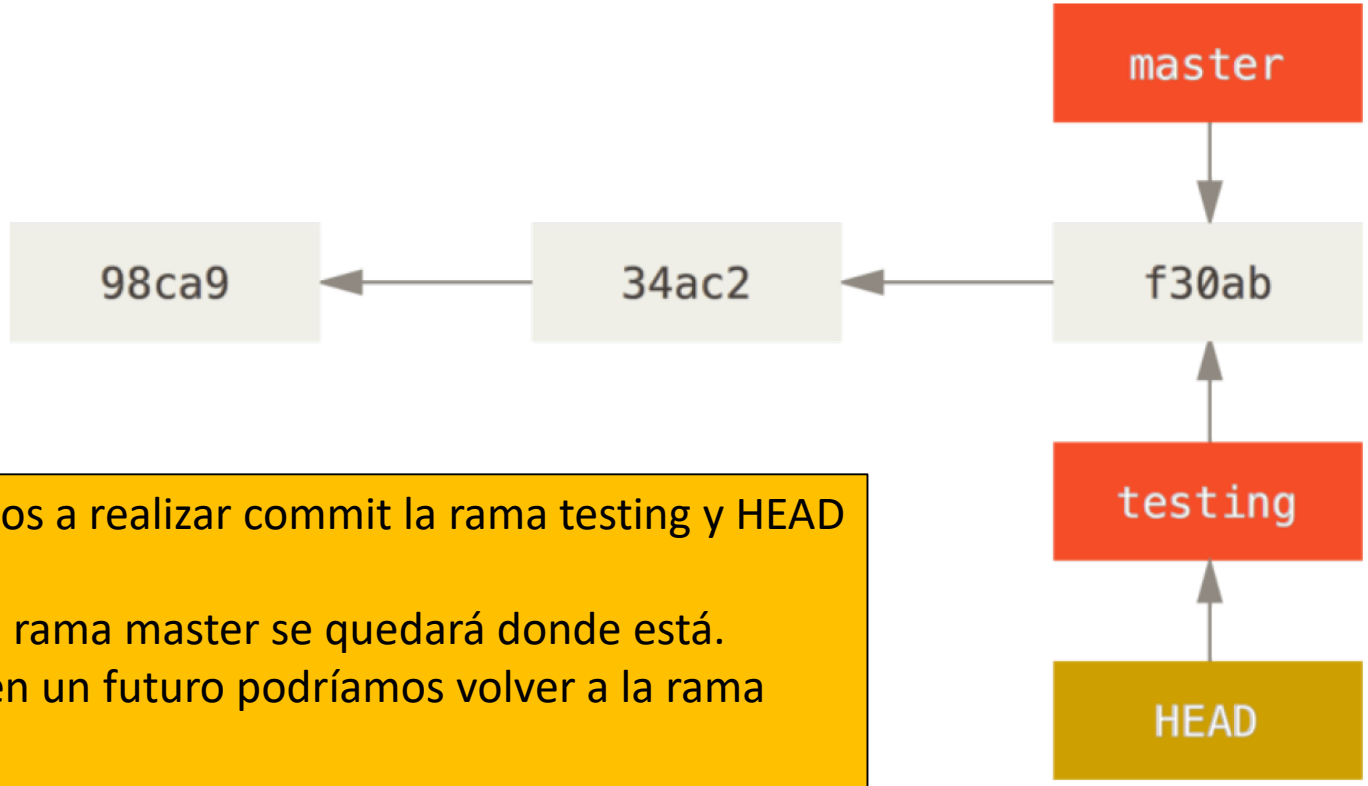
Crear una nueva rama

- Con el comando: **git branch testing**
- Crea un apuntador que apunta a la misma confirmación donde estés actualmente.
- Git utiliza un apuntado HEAD para saber en que rama nos encontramos en todo momento. Cuando cambiamos de rama el apuntador HEAD cambiará.



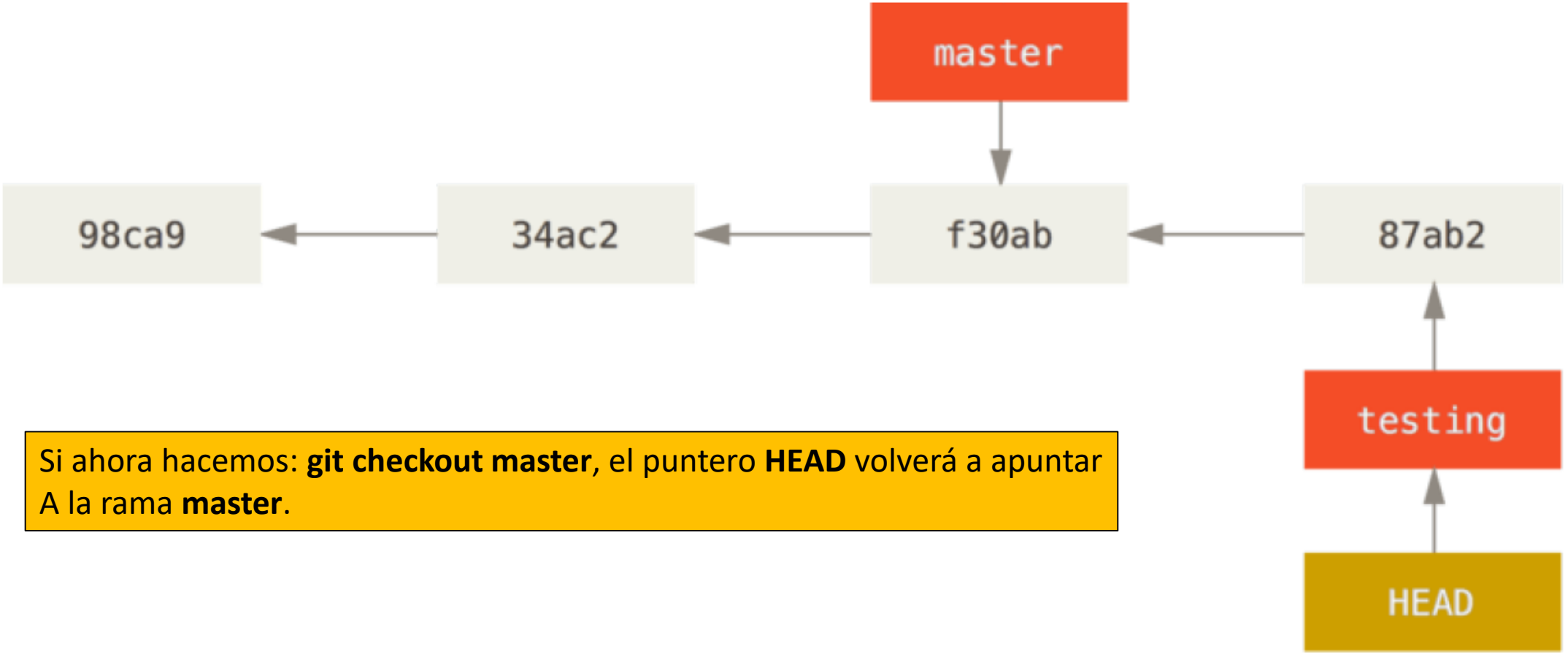
Cambiar de rama

- Comando: **git checkout testing**
- Mueve el apuntado HEAD a la rama testing.



Si ahora empezamos a realizar commit la rama testing y HEAD seguirán
Avanzando pero la rama master se quedará donde está.
De tal forma que en un futuro podríamos volver a la rama **master**.

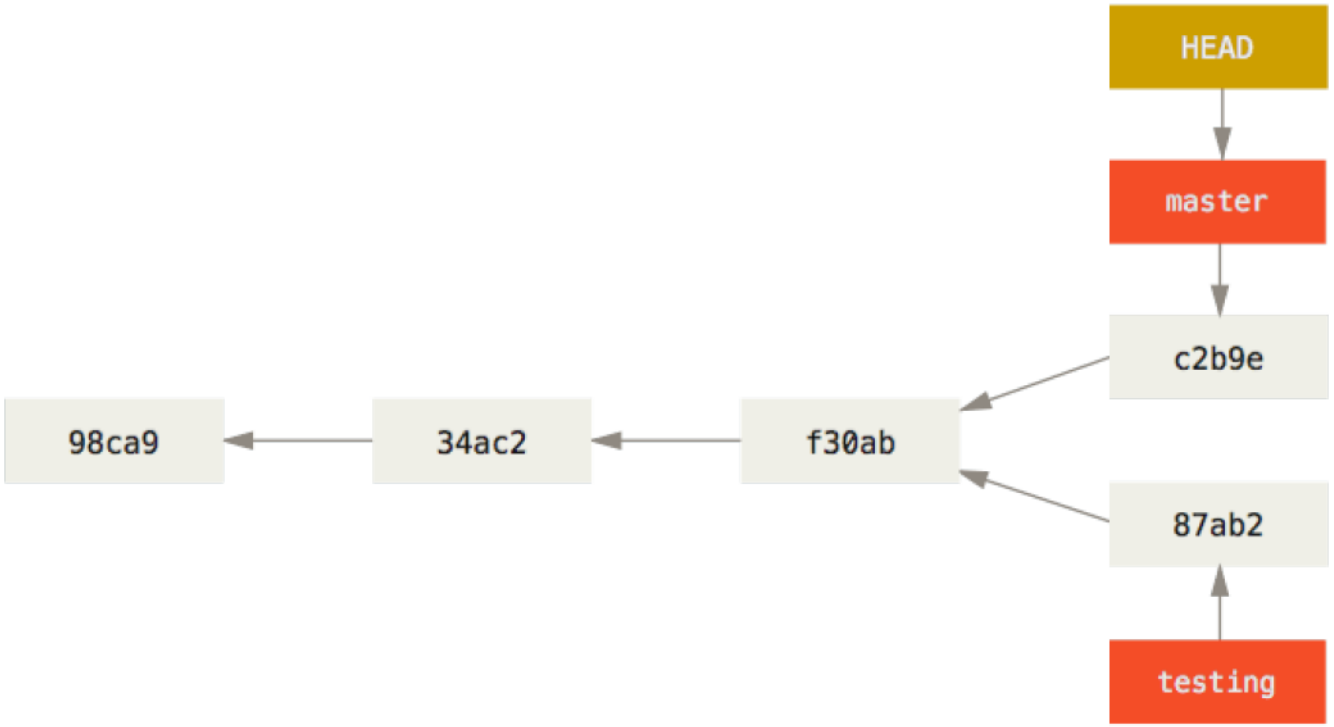
Cambiar de rama



Si ahora hacemos: **git checkout master**, el puntero **HEAD** volverá a apuntar A la rama **master**.

Cambiar de rama

- El comando git checkout realizada dos acciones:
  - Cambia de rama, es decir mueve el puntero HEAD.
  - Y revierte todos los archivos del directorio de trabajo dejándolos como estaban en la última instantánea confirmada en dicha rama master.



Consultar con git log

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Ejemplo, flujo de trabajo en la ramificación

- 1. Trabajas en un sitio web.
- 2. Creas una rama para un nuevo tema sobre el que quieres trabajar.
- 3. Realizas algo de trabajo en esa rama.

- En este momento, recibes una llamada avisándote de un problema crítico que has de resolver. Y sigues los siguientes pasos:
  - 1. Vuelves a la rama de producción original.
  - 2. Creas una nueva rama para el problema crítico y lo resuelves trabajando en ella.
  - 3. Tras las pertinentes pruebas, fusionas (merge) esa rama y la envías (push) a la rama de producción.
  - 4. Vuelves a la rama del tema en que andabas antes de la llamada y continuas tu trabajo.

Ramificación y Fusión de ramas

- Ejemplo, suponemos una rama master y abrimos una nueva rama (hotfix) para solucionar un problema: **git checkout -b hotfix**

- Hacemos modificaciones y confirmamos:
  - **git commit -a -m “mensaje …”**

- Después incorporamos los cambios a la rama master.
  - **git checkout master**
  - **git merge hotfix**

- Cuando fusionamos la rama master apunta al mismo sitio que hotfix. Y una vez hecho esto la rama hotfix se puede eliminar. Ya no la vamos a necesitar.
  - **git Branch -d hotfix**

Gestión de ramas

- El comando: **git branch** sin parámetros muestra las ramas presentes en el proyecto.
- Con un * indica cual es la rama en la que nos encontramos.

- Con el comando: **git branch -v**
  - Podemos ver los cambios de confirmación de cada rama.

- Otra opción para averiguar el estado de las ramas, es filtrarlas y mostrar sólo aquellas que han sido fusionadas (o que no lo han sido) con la rama actualmente activa. Git ofrece las opciones **--merged** y **--no-merged**.

Gestión de ramas

git branch --merged

iss53

* master

- Las ramas que no llevan el * se pueden eliminar porque su contenido ya se ha incorporado a otras ramas.

- Para mostrar las ramas que contienen trabajos sin fusionar, utilizaremos: **git branch --no-merged**, las ramas que muestre no dejaría borrarlas porque están pendientes de fusionar (**git branch -d testing) → ERROR**.
- Se puede forzar el borrado con la opción -D, pero perderemos esos cambios realizados en dicha rama: **git branch -D testing**