XUNTA DE GALICIA
CONSELLERÍA DE CULTURA, EDUCACIÓN
E ORDENACIÓN UNIVERSITARIA

ENTORNOS DE DESARROLLO

Jose Rey Cid

¿Qué es el 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.

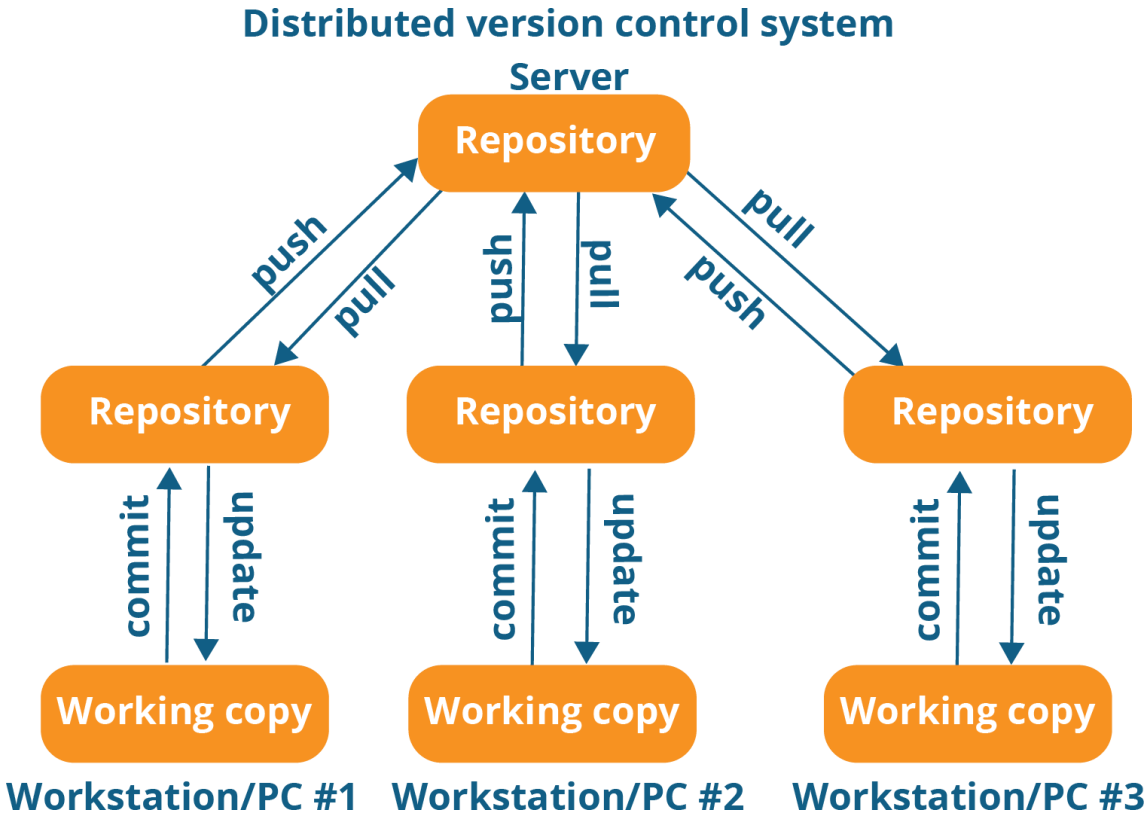
Te permite revertir archivos a un estado anterior, revertir el proyecto entero a un estado anterior, comparar cambios a lo largo del tiempo, ver quién modificó por última vez algo que puede estar causando un problema, quién introdujo un error y cuándo, y mucho más. Usar un VCS también significa generalmente que si fastidias o pierdes archivos, puedes recuperarlos fácilmente. Además, obtienes todos estos beneficios a un coste muy bajo.

¿Qué es GIT?

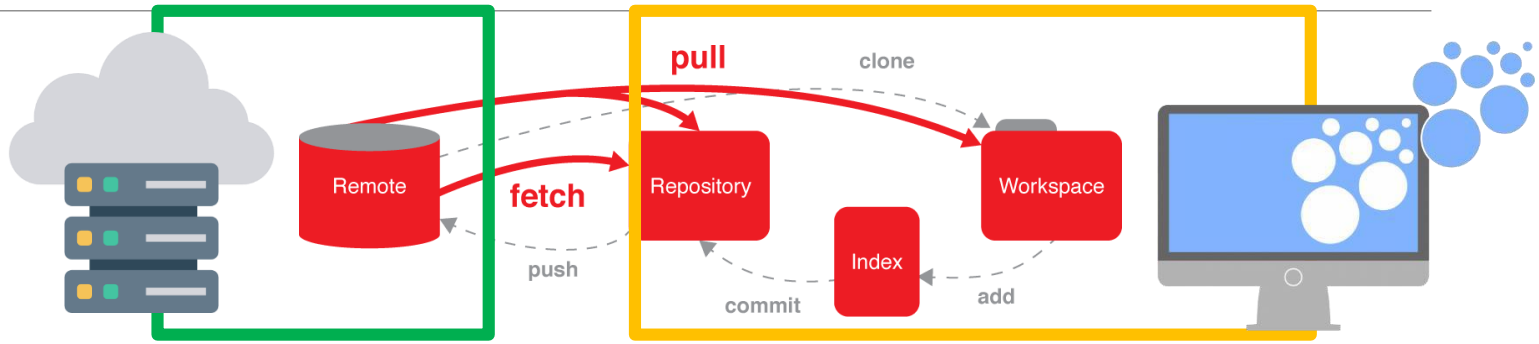
Es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.

Su propósito es llevar registro de los cambios en archivos de computadora y coordinar el trabajo que varias personas realizan sobre archivos compartidos.

Operaciones



Repositorio local y remoto



La arquitectura de Git cuenta con dos repositorios:

Repositorio remoto: Es un servidor de acceso compartido con otros usuarios donde cada uno sube los cambios del proyecto.

Repositorio local: En nuestro equipo contamos con nuestro espacio de trabajo donde nosotros trabajamos. Además contamos con un repositorio local en el que subir los nuestros cambios. De esta manera podemos volver a una versión anterior con la que hayamos trabajado.

Cuando nosotros subimos cambios al repositorio en remoto primero se subes los cambios de nuestro espacio trabajo al repositorio local y posteriormente al repositorio remoto. Del mismo modo para obtener cambios de otros usuarios primero se descarga del repositorio remoto al local y finalmente a nuestro espacio de trabajo.

1. Inicialización de un proyecto

1. `$ git init`
2. `$ git clone`

1.1 Git init.

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

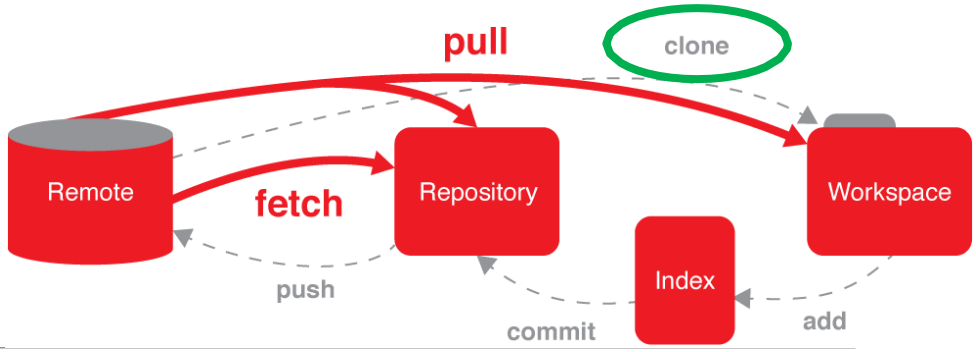
```
$ git init
```

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

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

```
$ git add *.c
$ git add README
$ git commit -m 'versión inicial del proyecto'
```

1.2 Git clone



Si deseas **obtener una copia de un repositorio Git existente** —por ejemplo, un proyecto en el que te gustaría contribuir— el comando que necesitas es git clone. Si estás familiarizado con otros sistemas de control de versiones como Subversion, verás que el comando es clone y no checkout.

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

```
$ git clone git://github.com/schacon/grit.git
```

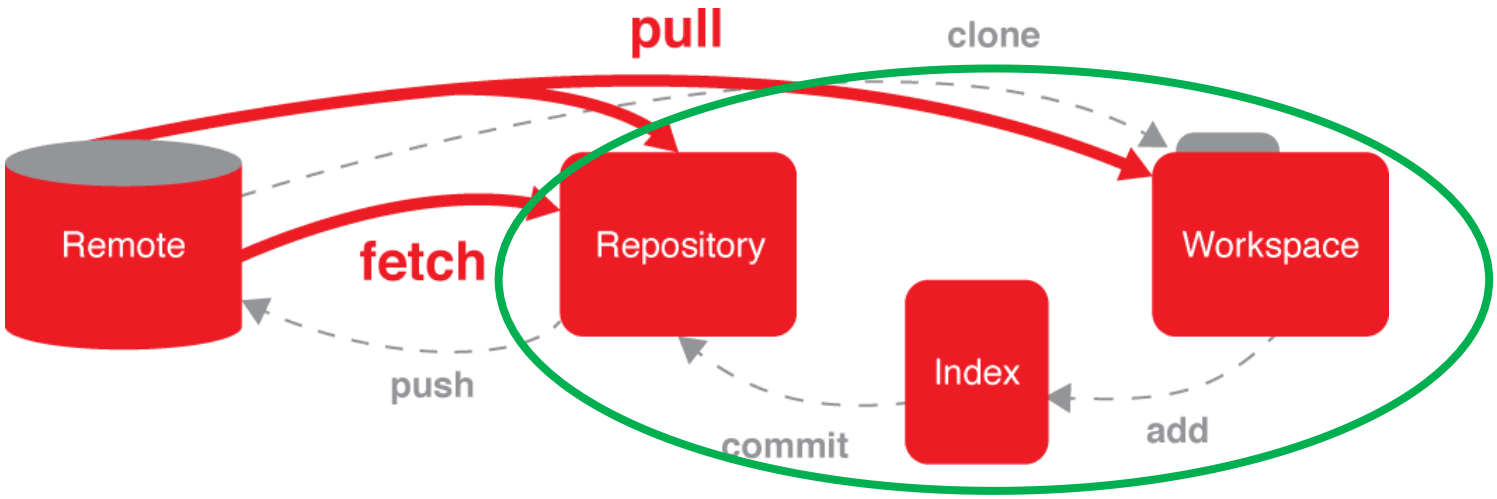
Esto crea un directorio llamado "grit", inicializa un directorio .git en su interior, descarga toda la información de ese repositorio, y saca una copia de trabajo de la última versión. Si quieres clonar el repositorio a un directorio con otro nombre que no sea grit, puedes especificarlo con la siguiente opción de línea de comandos:

```
$ git clone git://github.com/schacon/grit.git mygrit
```

2. Repositorio local

2.1 Estado de los ficheros
2.2 `$ git status`
2.3 `$ git add`
2.4 `$ git commit`
2.5 `$ git merge`

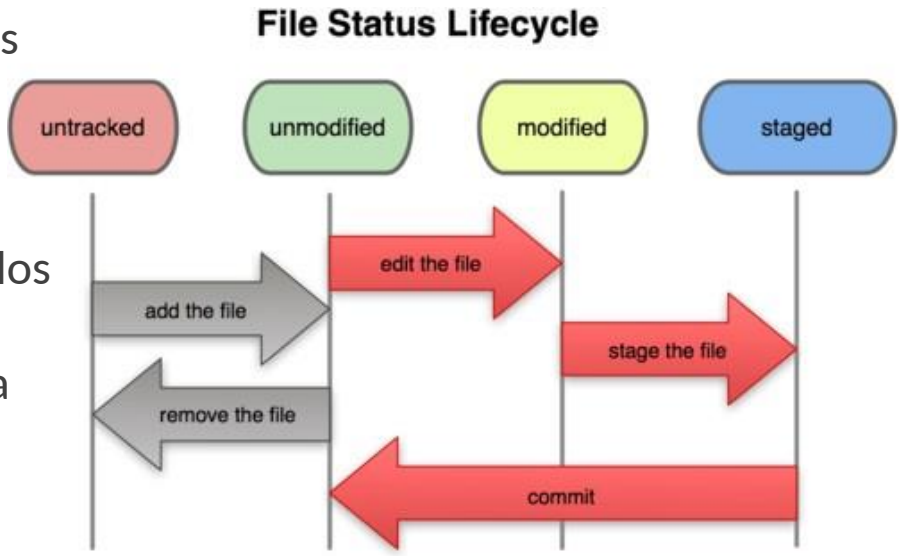
https://www.youtube.com/watch?v=3a2x1iJFJWc



2.1 Estado de los ficheros

Cada archivo de tu directorio de trabajo puede estar en uno de estos dos estados:

- bajo seguimiento (tracked): son aquellos que existían en la última instantánea; pueden estar sin modificaciones, modificados, o preparados
- sin seguimiento (untracked): son todos los demás —cualquier archivo de tu directorio que no estuviese en tu última instantánea ni está en tu área de preparación



La primera vez que clonas un repositorio, todos tus archivos estarán bajo seguimiento y sin modificaciones, ya que los acabas de copiar y no has modificado nada.

2.1 Git status

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

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

Esto significa no tienes archivos bajo seguimiento y modificados. Git tampoco ve ningún archivo que no esté bajo seguimiento, o estaría listado ahí. Por último, el comando te dice en qué rama estás.

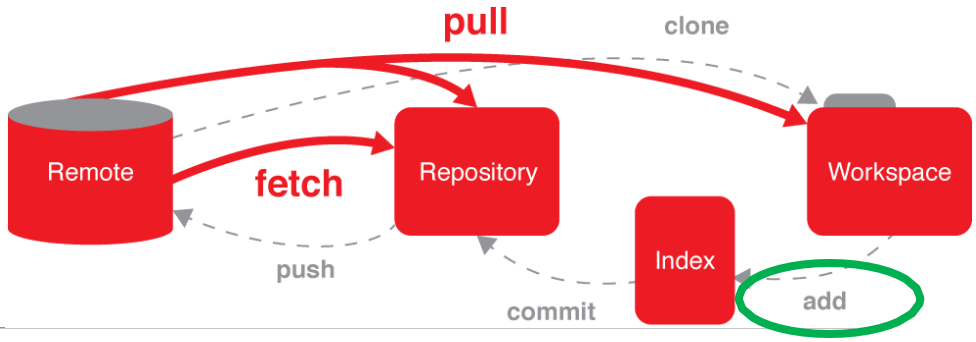
Digamos que añades un nuevo archivo a tu proyecto, un sencillo archivo README. Si el archivo no existía y ejecutas git status, verás tus archivos sin seguimiento así:

2.2 Git status

Digamos que añades un nuevo archivo a tu proyecto, un sencillo archivo README. Si el archivo no existía y ejecutas git status, verás tus archivos sin seguimiento así:

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

2.3 Git add

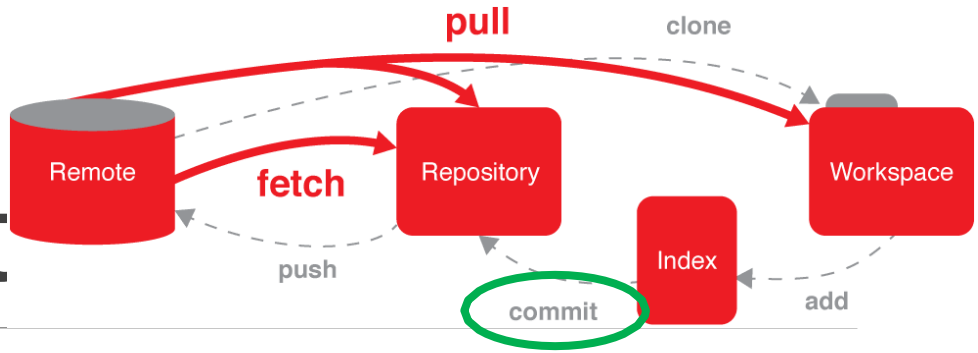



```
$ git add *.c
$ git add README
$ git commit –m 'versión inicial del proyecto'
```

Este comando nos añade archivos al index de nuestro repositorio, de manera que convertimos un archivo untracked en un archivo modified.

De esta manera al hacer el commit se guará en nuestro repositorio local, para que posteriormente en un push se suba al repositorio remoto.

2.4 Git commit

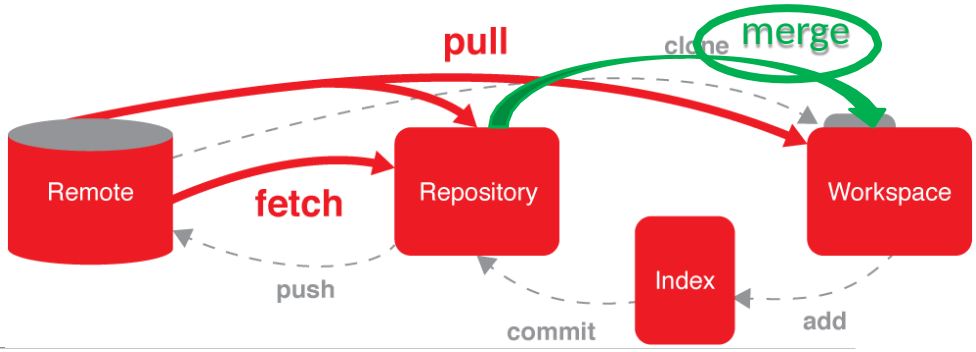



```
$ git commit –m 'versión inicial del proyecto'
```

Guarda los cambios de los ficheros indexados de nuestro espacio de trabajo en nuestro repositorio.

Añadimos las opción –m para introducir un mensaje a ese commit.

2.5 Git merge




```
$ git merge
```

Sirve para incorporar los cambios de un repositorio a otro.

En este contexto se puede utilizar a nivel repositorio local para incorporar los cambios de nuestro repositorio local a nuestro espacio de trabajo. Por ejemplo después de realizar un `$ git fetch`, incorporados los cambios del repositorio remoto al nuestro local los podemos mezclar con nuestro espacio de trabajo.

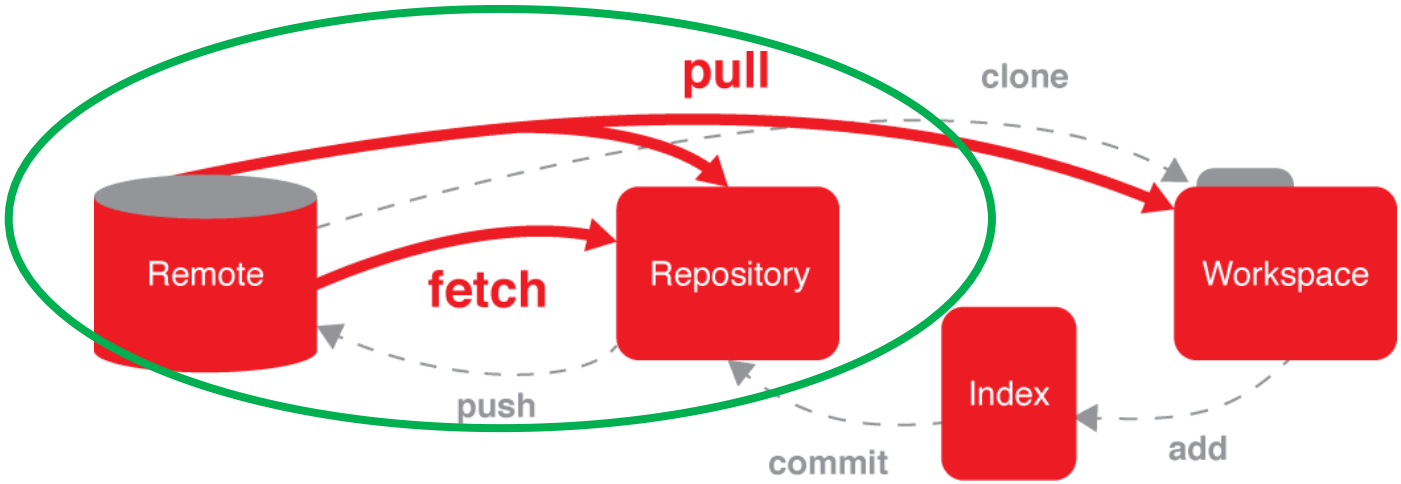
3. Repositorio remoto.

3.1 `$ git remote`

3.2 `$ git fetch`

3.3 `$ git pull`

3.4 `$ git push`



3. Repositorio remoto (ii).

Los repositorios remotos son versiones de tu proyecto que se encuentran alojados en Internet o en algún punto de la red. Puedes tener varios, cada uno de los cuales puede ser de sólo lectura, o de lectura/escritura, según los permisos que tengas.

Colaborar con otros implica gestionar estos repositorios remotos, y mandar (push) y recibir (pull) datos de ellos cuando necesites compartir cosas.

3.1 Git remote

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

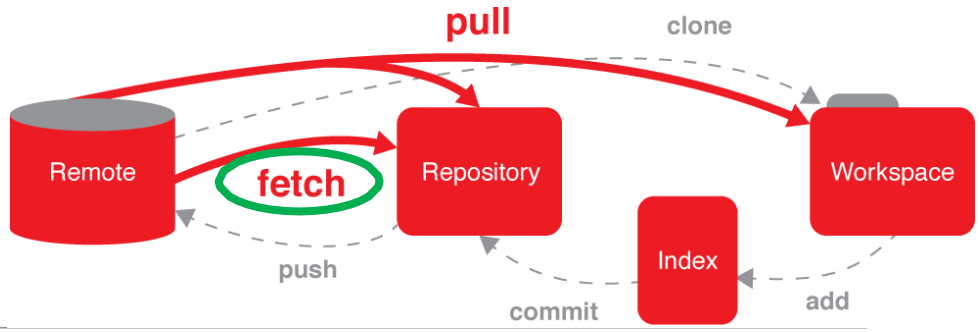
```
$ git remote -v bakkdoor
git://github.com/bakkdoor/grit.git cho45
git://github.com/cho45/grit.git defunkt
git://github.com/defunkt/grit.git koke
git://github.com/koke/grit.git origin
git@github.com:mojombo/grit.git
```

3.1 Git remote (ii)

Para añadir un nuevo repositorio Git remoto, asignándole un nombre con el que referenciarlo fácilmente, ejecuta `git remote add` [nombre]

```
$ git remote origin
$ git remote add pb
      git://github.com/paulboone/ticgit.git
$ git remote -v
      origin git://github.com/schacon/ticgit.git pb
      git://github.com/paulboone/ticgit.git
```

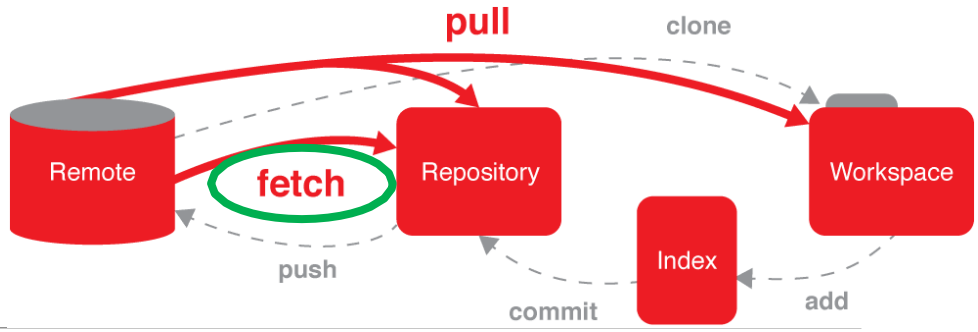
3.2 Git fetch



Ahora puedes usar la cadena "pb" en la línea de comandos, en lugar de toda la URL. Por ejemplo, si quieres recuperar toda la información de Paul que todavía no tienes en tu repositorio, puedes ejecutar

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
* [new branch] master -> pb/master
* [new branch] ticgit -> pb/ticgit
```

3.2 Git fetch (ii)

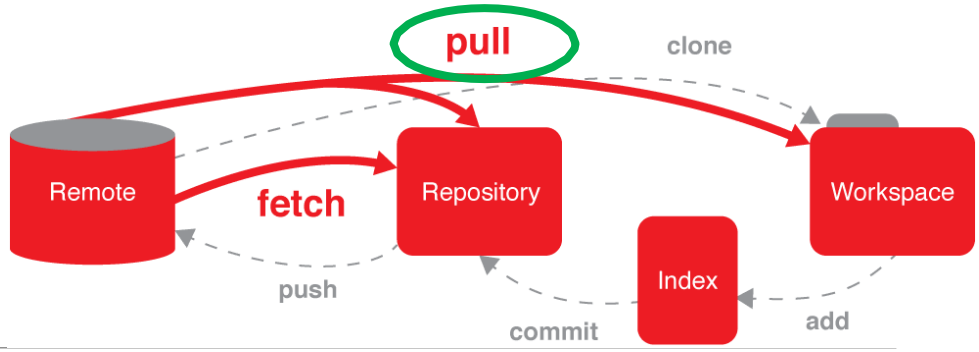



```
$ git fetch [remote-name]
```

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

git fetch origin recupera toda la información enviada a ese servidor desde que lo clonaste (o desde la última vez que ejecutaste fetch). Es importante tener en cuenta que el comando fetch sólo recupera la información y la pone en tu repositorio local

3.3 Git pull

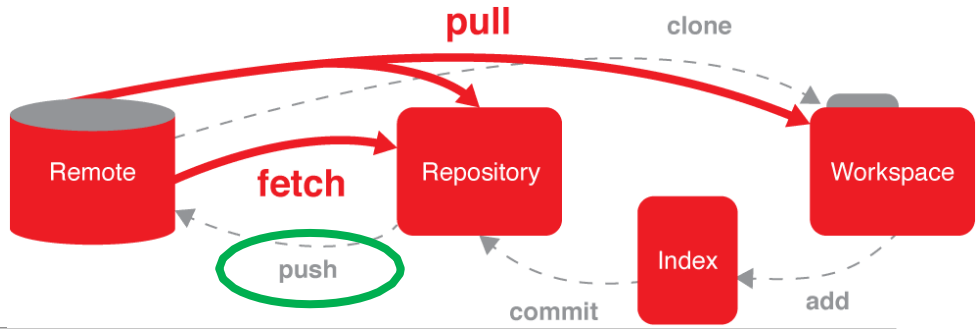



```
$ git pull
```

Incorpora los cambios del repostorio en nuestro espacio de trabajo directamente.

Es lo mismo que realizar un `$ git fetch  + $ git merge`

3.4 Git push



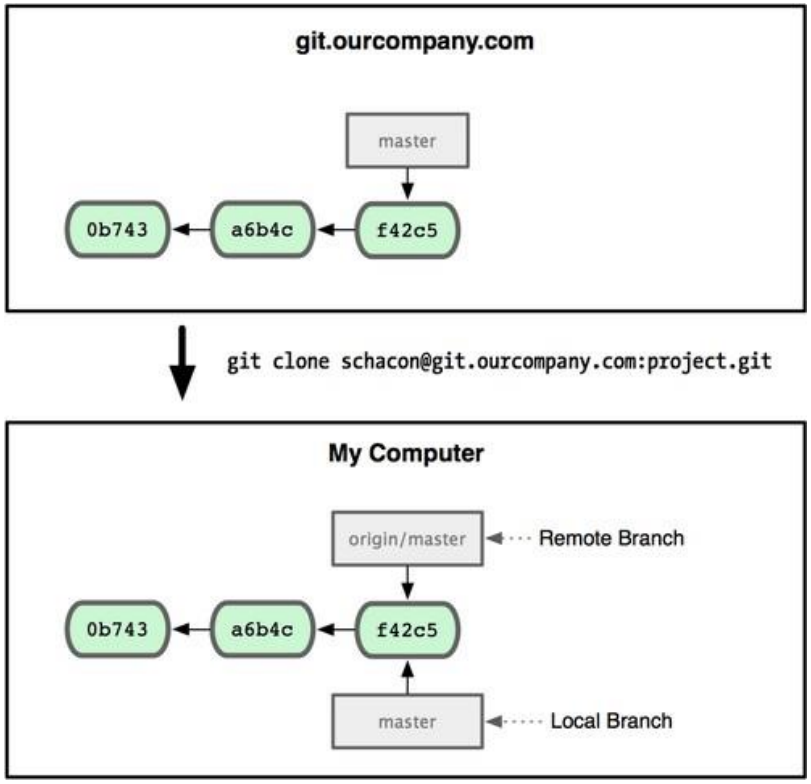
Cuando tu proyecto se encuentra en un estado que quieres compartir, tienes que enviarlo a un repositorio remoto. El comando que te permite hacer esto es sencillo: git push [nombre-remoto][nombre-rama]. Si quieres enviar tu rama maestra (master) a tu servidor origen (origin), ejecutarías esto para enviar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

```
$ git push origin master
```

4 Apuntadores

Supongamos que tienes un sevidor Git en tu red, en git.ourcompany.com.

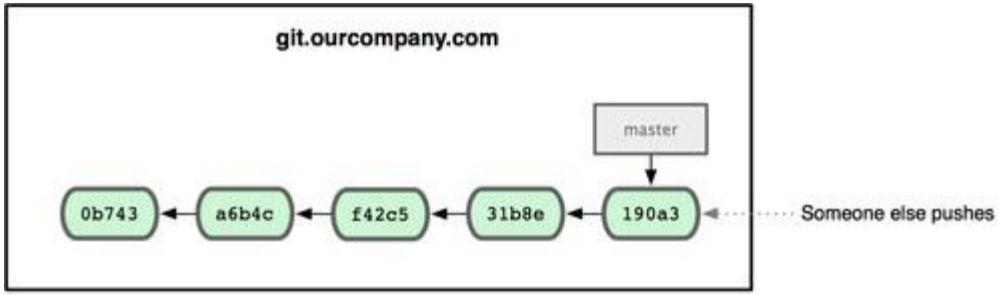
Si haces un clón desde ahí, Git automáticamente lo denominará `origin`, traerá (`pull`) sus datos, creará un apuntador hacia donde esté en ese momento su rama master, denominará la copia local `origin/master`; y será inamovible para ti. Git te proporcionará también tu propia rama master, apuntando al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

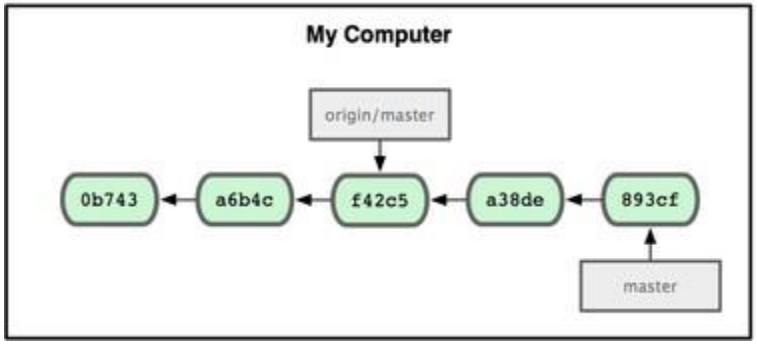


Apuntadores (ii)

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

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





Creando etiquetas

Git tiene la habilidad de etiquetar (tag) puntos específicos en la historia como importantes. Generalmente la gente usa esta funcionalidad para marcar puntos donde se ha lanzado alguna versión (v1.0, y así sucesivamente).

Listar las etiquetas disponibles en Git es sencillo:

```
$ git tag
v0.1
v1.3
```

Este comando lista las etiquetas en orden alfabético; el orden en el que aparecen no es realmente importante.

Creando etiquetas (II)

Crear una etiqueta anotada en Git es simple. La forma más fácil es especificar -a al ejecutar el comando tag:

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
'
```

Creando etiquetas (III)

Puedes ver los datos de la etiqueta junto con la confirmación que fue etiquetada usando el comando git show:

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 14:45:11 2009 -0800
my version 1.4
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800
    Merge branch 'experiment
```

Creando etiquetas (IV)

Por defecto, el comando git push no transfiere etiquetas a servidores remotos. Tienes que enviarlas explicitamente a un servidor compartido después de haberlas creado. Este proceso es igual a compartir ramas remotas —puedes ejecutar git push origin [tagname].

```
$ git push origin v1.5 Counting objects: 50, done. Compressing objects: 100% (38/38), done. Writing objects: 100% (44/44), 4.56 KiB, done. Total 44 (delta 18), reused 8 (delta 1) To git@github.com:schacon/simplegit.git * [new tag] v1.5 -> v1.5
```

Ramas

https://git-scm.com/book/es/v1/Ramificaciones-en-Git

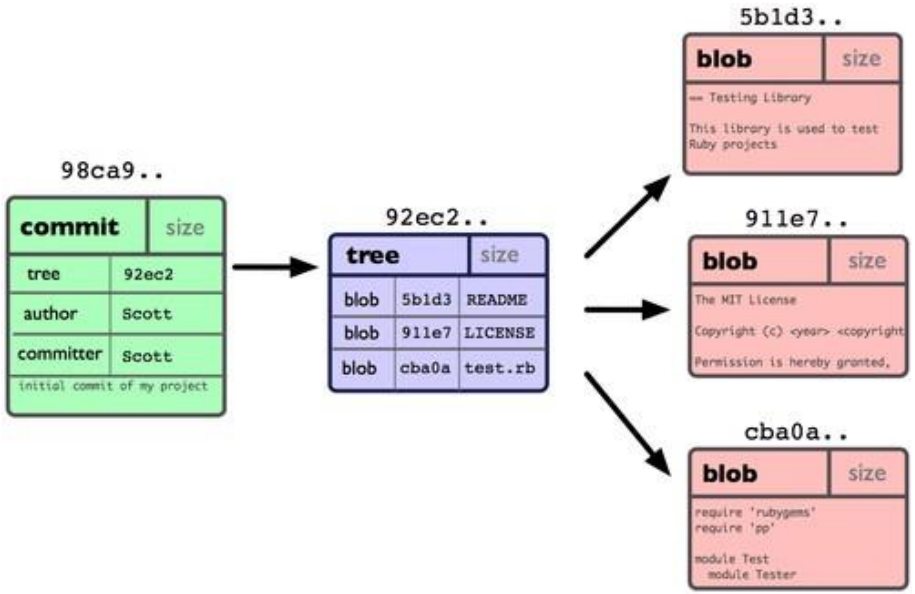
Para entender realmente cómo ramifica Git, previamente hemos de examinar la forma en que almacena sus datos. Git almacena como una serie de instantáneas (copias puntuales de los archivos completos, tal y como se encuentran en ese momento).

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 padre en los casos de confirmación normal, y múltiples padres en los casos de estar confirmando una fusión (merge) de dos o mas ramas).

Ramas (II)

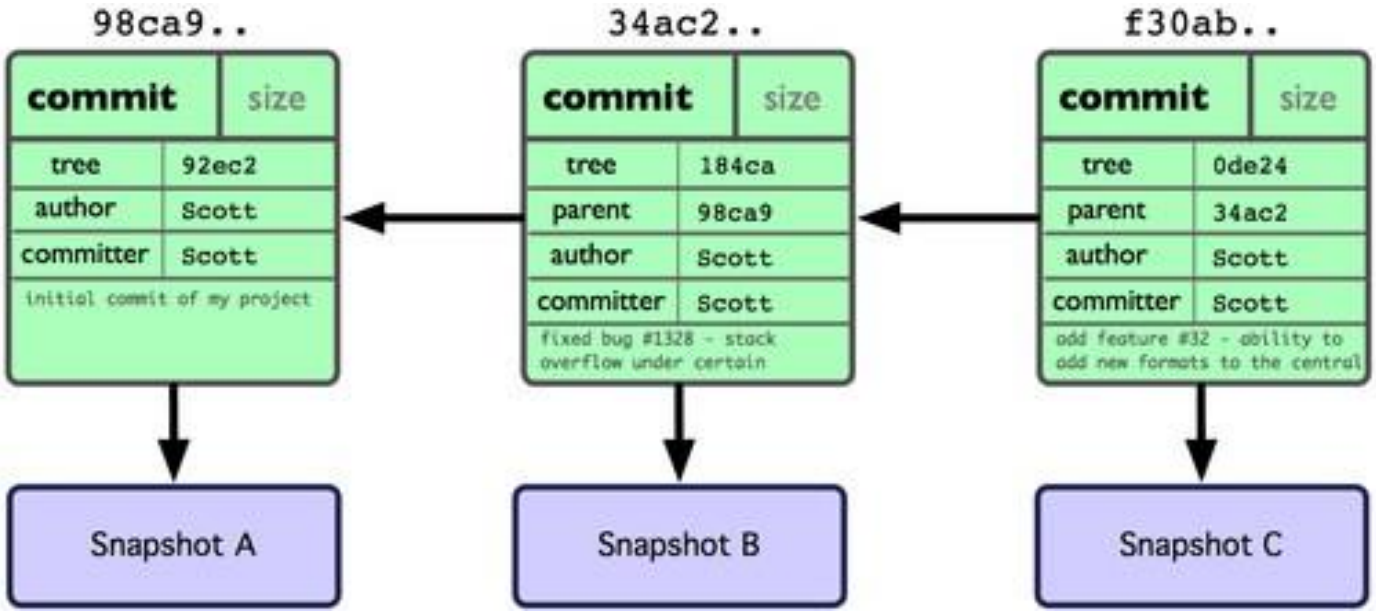
Cuando creas una confirmación con el comando git commit, Git realiza sumas de control de cada subcarpeta (en el ejemplo, solamente tenemos la carpeta principal del proyecto), y las guarda como objetos árbol en el repositorio Git. Después, Git crea un objeto de confirmación con los metadatos pertinentes y un apuntador al objeto árbol raiz del proyecto. Esto permitirá poder regenerar posteriormente dicha instantánea cuando sea necesario.

Se generan una commit con la información subida, con un archivo tree que controlan todos los blos's subidos y los blob's que son los archivos subidos.



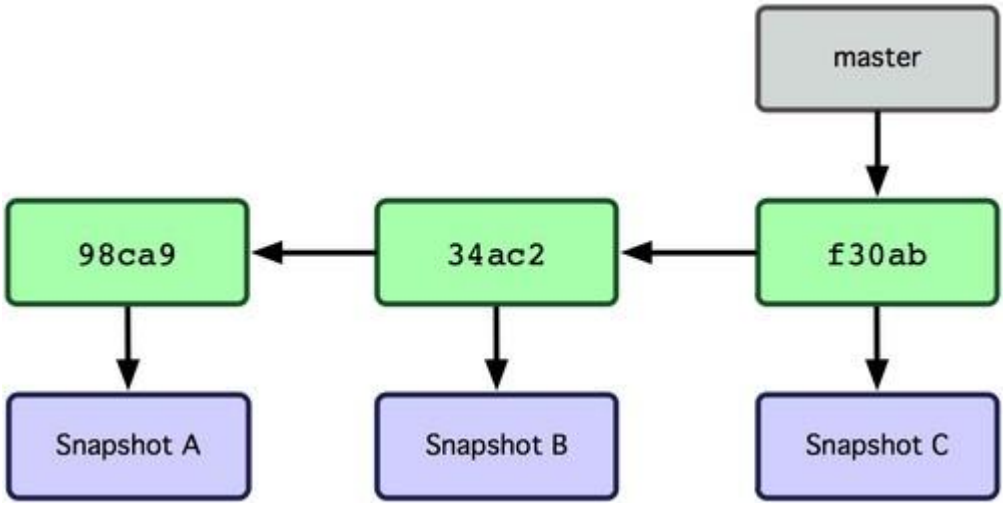
Ramas (III)

Si haces más cambios y vuelves a confirmar, la siguiente confirmación guardará un apuntador a esta su confirmación precedente. Tras un par de confirmaciones más:



Ramas (IV)

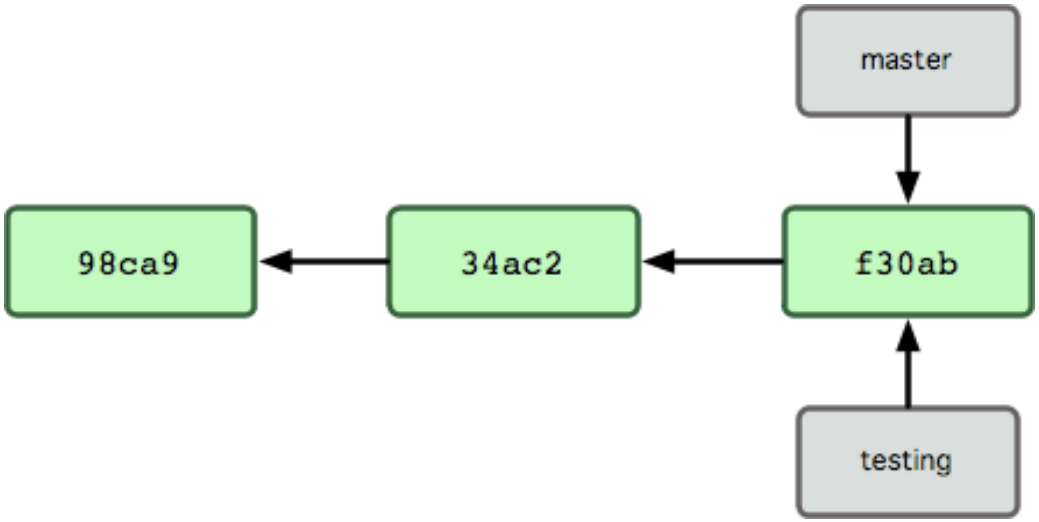
Una rama Git es simplemente un apuntador móvil apuntando a una de esas confirmaciones. La rama por defecto de Git es la rama master. Con la primera confirmación de cambios que realicemos, se creará esta rama principal master apuntando a dicha confirmación. En cada confirmación de cambios que realicemos,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



Ramas (V)

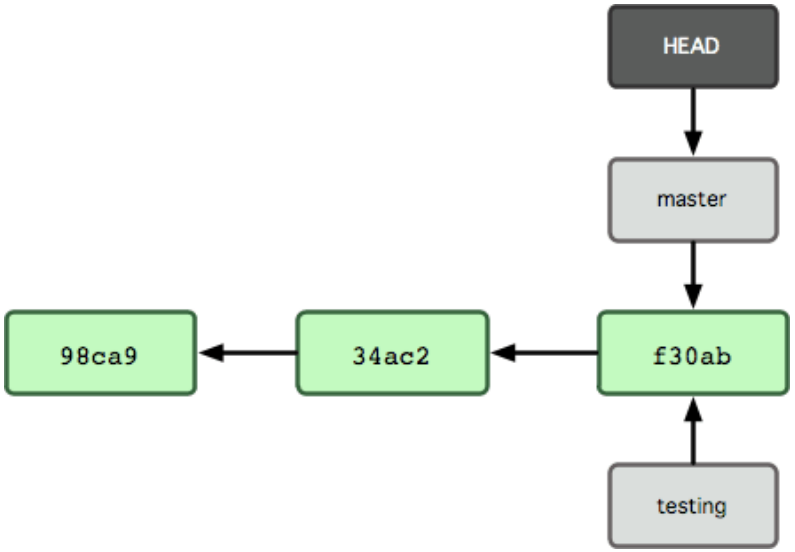
¿Qué sucede cuando creas una nueva rama? Bueno..., simplemente se crea un nuevo apuntador para que lo puedas mover libremente. Por ejemplo, si quieres crear una nueva rama denominada "testing". Usarás el comando git branch:

```
$ git branch testing
```



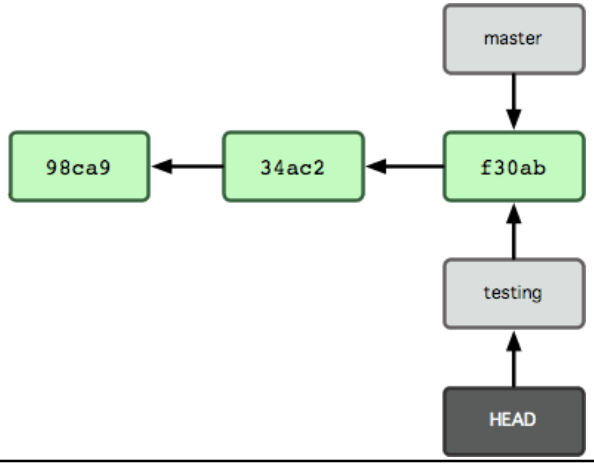
Ramas (VI)

Y, ¿cómo sabe Git en qué rama estás en este momento? Pues…, mediante un apuntador especial denominado HEAD.



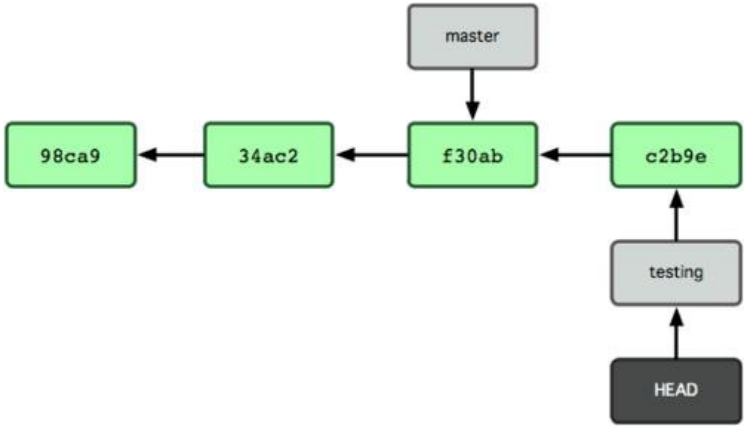
Ramas (VII)

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



¿Que pasa ahora si subimos ahora un nuevo cambio?

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



Ramas (VIII)

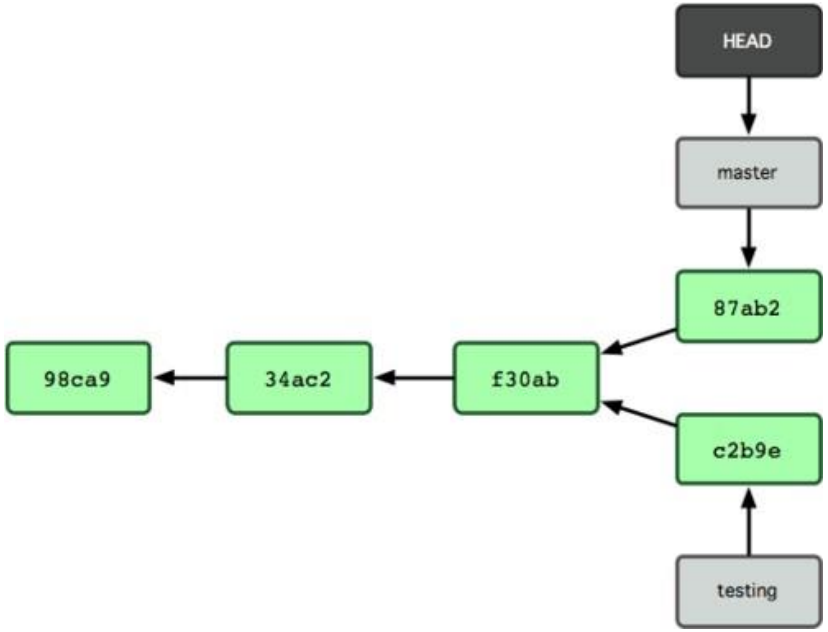
```
$ git checkout master
```

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.

```
$ git commit -a

-m 'made other changes'
```

Subimos nuevos archivos y nos queda:



.gitignore

El **.gitignore** es un fichero que se coloca en la raíz del proyecto en el se indica que fichero no serán subidos al repositorio. Esto es interesante porque siempre hay fichero que no deben ser subidos al repositorio, por ejemplo:

- **.jar** , **.war** : Es el resultado de compilar un proyecto java normal o web.
- **.class:** Son el resultado de compilación de una clase en java.
- **/target/:** directorio sobre el que se descargan la librerías en Maven.

- Los motivos por los que no se suben estos ficheros son:
  - Son específicos de cada equipo ordenador, por lo tanto cada usuario generará los suyos propios.
  - Ralentizan las subidas y bajadas de cambios.
  - Ocupan el repositorio de tamaño inservible. Un fichero .war pesan mínimo 100 mb.
  - Los fichero compilables son distintos para cada máquina de no ser excluidos en el git ignore supondría que cada vez que un usuario suba cambios al repositorio estos sean subidos al repositorio también de manera que otro usuario también los descargaría.

Ejemplo de .gitignore

.gitignore Java

```
1  ##############################
2  ## Java
3  ##############################
4  .mtj.tmp/
5  *.class
6  *.jar
7  *.war
8  *.ear
9  *.nar
10 hs_err_pid*
11
12 ##############################
13 ## Maven
14 ##############################
15 target/
16 pom.xml.tag
17 pom.xml.releaseBackup
18 pom.xml.versionsBackup
19 pom.xml.next
20 release.properties
21 dependency-reduced-pom.xml
22 buildNumber.properties
23 .mvn/timing.properties
24 .mvn/wrapper/maven-wrapper.jar
```

Ejemplo completo de gitignore GitHub