Documentación de GIT

Página principal de git: http://git-scm.com/
Para trabajar remotamente se puede utilizar https://github.com o https://about.gitlab.com/
Tutorial en línea de git (octocats): https://try.github.io/levels/1/challenges/1

Es un versionador que puede ser utilizado por más de 10.000 personas como la comunidad que mantiene el kernel del Linux.
Contenido del documento:

Instalando el git en Windows:

Bájalo de :

1. http://msysgit.github.com/

O

2. Esta segunda opción descarga un instaldor del git y el git-bash.

- https://github.com/msysgit/msysgit/releases
- `git`: una herramienta de control de versiones
- `bash`: un shell que ejecuta mandatos, en lugar de vez de escribir el nombre de un comando y pulsar Enter del teclado.

3. Y si queremos una interfaz gráfica podemos bajar github for Windows: https://windows.github.com/ (durante el curso no lo usaremos)

Para saber en qué rama estamos y nuestros cambios a colores: en windows hay que instalar un shell y en Windows podemos instalar el git-posh, para ello ver el siguiente enlace:

http://learnaholic.me/2012/10/12/make-powershell-and-git-suck-less-on-windows/

(el powershell, del enlace de arriba ya viene con el Windows7 en adelante, por lo que solo es necesario instalar el posh-git)

Para poner una imagen como marca de agua:

For fullscreen and transparency with Powershell (or any other command prompt) in Windows, try ConEmu: http://code.google.com/p/conem... I used to use Console2 but I like ConEmu much better. I typically run it the way I do iTerm2 on my MBP, see attached (watching "House of Cards" while working :-)

http://www.imtraum.com/blog/streamline-git-with-powershell/

El git tiene la siguiente estructura

Working Directory	Este es el directorio de trabajo, todo lo que hacemos está en esta área **Readme.md :** Archivo donde se indica de que se trata el repositorio
Staging Area	En este espacio ponemos los archivos que queremos hacer commit. Es decir los archivos que hemos hecho cambios y cuyos cambios queremos que se registren como un solo commit. A estos archivos se los llama archivos preparados para commit (confirmar). **$Git add** miArchivo : con este comando ponemos los archivos que queremos al staging área $Git add . : con punto añadimos todos los archivos cambiados al staging area
Commits Area	En esta área están todos los archivos confirmados o cuyo commit hemos realizado, es decir aquí está todo lo que queremos que se quede del trabajo realizado y lo hemos confirmado (commit) así. **$Git commit** –m "nombre o descripción del commit" Este comando pasa TODO lo que está en el staging Area a estado de commit

Es importante saber que cada commit registra el estado actual del proyecto y se puede volver a ese estado con el comando $git checkout NNNNN donde NNNNN son los primeros 5 dígitos de ese commit.

Configuración

La configuración tiene 3 niveles(aplicación: --global, usuario y proyecto):

Aplicación	~\.gitconfig, se guarda para toda la aplicación (en Linux por defecto esta en etc)
Usuario	\user\config es a nivel de usuario
Proyecto	Cuando solo modificamos la configuración de .gitconfig de nuestro proyecto
P/trabajar remotamente	$ export http-proxy=http://172.20.240.5:8080 $ export https-proxy=http://172.20.240.5:8080
p/mostrar nombre rama	En windows para ello instalamos el posh-git. $apt-get install git-sh //instala $git-sh //muestra la rama en la que estoy
p/colores	$ git config - -global color.ui true (en linux, en win hay que instalar el posh-git)

Trabajando con git

Manual oficial de git en español: http://git-scm.com/book/es/Empezando-Configurando-Git-por-primera-vez

Tu identidad y tu editor

Lo primero que deberías hacer cuando instalas 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 "John Doe"
$ git config --global user.email johndoe@example.com

$ git config --global core.editor emacs     →pone como editor a emacs
$ git config --list   //para ver configuración de nuestro repo
$ git config user.name  // para ver el nombre de usuario
```

Crear repositorio

$ git init : crea nuestro directorio actual como nuevo repositorio GIT

$ git clone <url> : crea un repositorio local del repositorio remoto indicado en la url, este comando creará un directorio con el contenido del repositorio global

.gitignore : es importante poner en este archivo que archivos o carpetas no queremos que estén en nuestros commits, y no sean seguidos

Para el ejemplo de BattleShip el .ignore tiene el siguiente contenido

```
proyecto/build/*          -> archivos compilados a java
proyecto/node_modules/*   -> módulos del node
proyecto/.idea/*          -> archivo que genera el webstorm (API para coffe pago)
*.[oa]                    -> ignorará los archivos que terminen en "o" o "a"
*~                        -> ignorará los archivos que terminen en tilde
#comentario
build/          -> ignora all files in the build/ directory
doc/*.txt       -> ignora doc/notes.txt, but not doc/server/arch.txt
doc/**/*.txt    -> ignora all .txt files in the doc/ directory
!lib.txt        -> hará track lib.txt,aunque ignoring .txt en linea arriba
```

Autocompletado

Si usas el shell Bash, Git viene con un buen script de autocompletado que puedes activar. Descárgalo directamente desde el código fuente de Git en `https://github.com/git/git/blob/master/contrib/completion/git-completion.bash`, copia este fichero en tu directorio `home` y añade esto a tu archivo `.bashrc`:

```
source ~/git-completion.bash
```

Si quieres que Git tenga automáticamente autocompletado para todos los usuarios, copia este script en el directorio `/opt/local/etc/bash_completion.d` en sistemas Mac, o en el directorio `/etc/bash_completion.d/` en sistemas Linux. Este es un directorio de scripts que Bash cargará automáticamente para proveer de autocompletado.

Si estás usando Windows con el Bash de Git, el cual es el predeterminado cuando instalas Git en Windows con msysGit, el autocompletado debería estar preconfigurado.

Presiona el tabulador cuando estés escribiendo un comando de Git, y deberían aparecer un conjunto de sugerencias para que escojas.

Alias de Git

Para no escribir el texto entero de cada uno de los comandos de Git, puedes establecer un alias para cada comando usando `git config`. Aquí hay un par de ejemplos que tal vez quieras establecer:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

Esto significa que, por ejemplo, en vez de escribir `git commit`, simplemente necesitas escribir `git ci`. A medida que uses Git, probablemente uses otros comandos de forma frecuente. En este caso no dudes en crear nuevos alias.

Esta técnica también puede ser muy útil para crear comandos que creas que deben existir. Por ejemplo, para corregir el problema de usabilidad que encontramos al quitar del área de preparación un archivo, puedes añadir tu propio alias:

```
$ git config --global alias.unstage 'reset HEAD --'
```

Esto hace los siguientes dos comandos equivalentes:

```
$ git unstage fileA
$ git reset HEAD fileA
```

Esto parece un poco más claro. También es común añadir un comando `last`, tal que así:

```
$ git config --global alias.last 'log -1 HEAD'
```

De esta forma puedes ver la última confirmación fácilmente:

```
$ git last
```

Obteniendo ayuda

```
$ git help <comando>
$ git <comando> --help
$ man git-<comando>
```

Poniendo archivos al staging Area

Para que un archivo pase al commit primero se debe poner al stagging área y luego solo se aplica el comando : $ commit –m “nombre del comit”

$git add arch	Pone el archivo al staging área.
$git add .	Pone todos los archivos modificados al Staging Area , no importa si están en el directorio actual o no.

Inspeccionando nuestro repositorio

$git status	Muestra los archivos modificados de la rama actual. Si están activados los colores mostrará los archivos modificados con rojo Y con verde los archivos en el Staging Area
$git log $git log –p -2 $git log stat `$git log --pretty=oneline`	Muestra TODOS los commits (salir con q) El –p hace que muestra las diferencias introducidas en cada confirmación y el 2 que solo muestre las dos últimas confirmaciones (commits) Muestra estadísticas de commits $git log - -pretty=format: “%h -%an,%ar:%s” $git log - -pretty=format: “%h %s” –graph $git log - - since=2.weeks o $git log - - since=”01/08/2014” o $git log - -since=”2 years 1day 3 minutes ago” Muestra en una línea nuestros commits (confirmaciones)
$git diff	Muestra diferencias entre mis archivos preparados y no preparados (mas info que el status). Si ya están preparados todos los cambios no mostrara nada. (muestra cambios no preparados)
$git diff --staged	Compara tus cambios preparados con tu última confirmación (último commit).

Confirmando cambios: commit

$git commit –m “nombre comit”	Saca una instantánea del Staging Area (archivos preparados). Cuanto más seguidos nuestros commits, mejor para nosotros, cada que hacemos commits, tenemos la posibilidad de deshacer
$git commit -a –m “nombre commit”	Pone todas las modificaciones como commit, aunque no estén Preparados

$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit - -amend
	- - amend modifica el último commit como en el ejemplo. Los tres comandos harán un única confirmación.

Borrar Archivos del directorio de trabajo, preparados y commiteados

$git rm (–f) arch	Preparó arch para la eliminación, siguiente commit se eliminará. La opción –f fuerza la eliminación, si el archivo ya estaba preparado (en el Staging Area)
$git rm –cached arch	Interrumpe su seguimiento y si estaba en Stagging Area lo saca, es útil cuando olvidamos añadir el archivo a .ignore
$git reset HEAD arch	Saca arch de la lista staged, es decir borra archivos preparados
$git checkout --arch	Vuelve nuestro arch a la versión del último commit.

$git rm -r nombreDir	prepara el directorio para borrado, siguiente commit borrará el directorio

Renombrando Archivos del directorio de trabajo

$git mv file_from file_to	Prepara el archive para renombrar, el nuevo nombre se hará efectivo en el siguiente commit.

Ejercicio 1:

1. Crea tu repositorio con el nombre de ejercicio1
2. En el directorio de ejercicio1, crea el archivo: arch1
   con la siguiente línea: primer commit branch master
3. Ejecuta el comando git status
4. Añade este archivo al stagging área. Y luego ejecuta un git status y los comandos indicados para ver el estado del repositorio.
5. Añade una segunda línea a tu archivo: “para segundo commit”. Y luego ejecuta un git status y los comandos indicados para ver el estado del repositorio.
6. Realiza un commit de tu stagging área, con el nombre de: “primer commit rama master”. Y luego ejecuta un git status y los comandos indicados para ver el estado del repositorio.
7. Añade tus nuevos cambios al stagging área y realiza el commit correspondiente y luego ejecuta un git status y los comandos indicados para ver el estado del repositorio.
8. Luego ve al primer commit
9. Vuelve al segundo commit o estado actual.

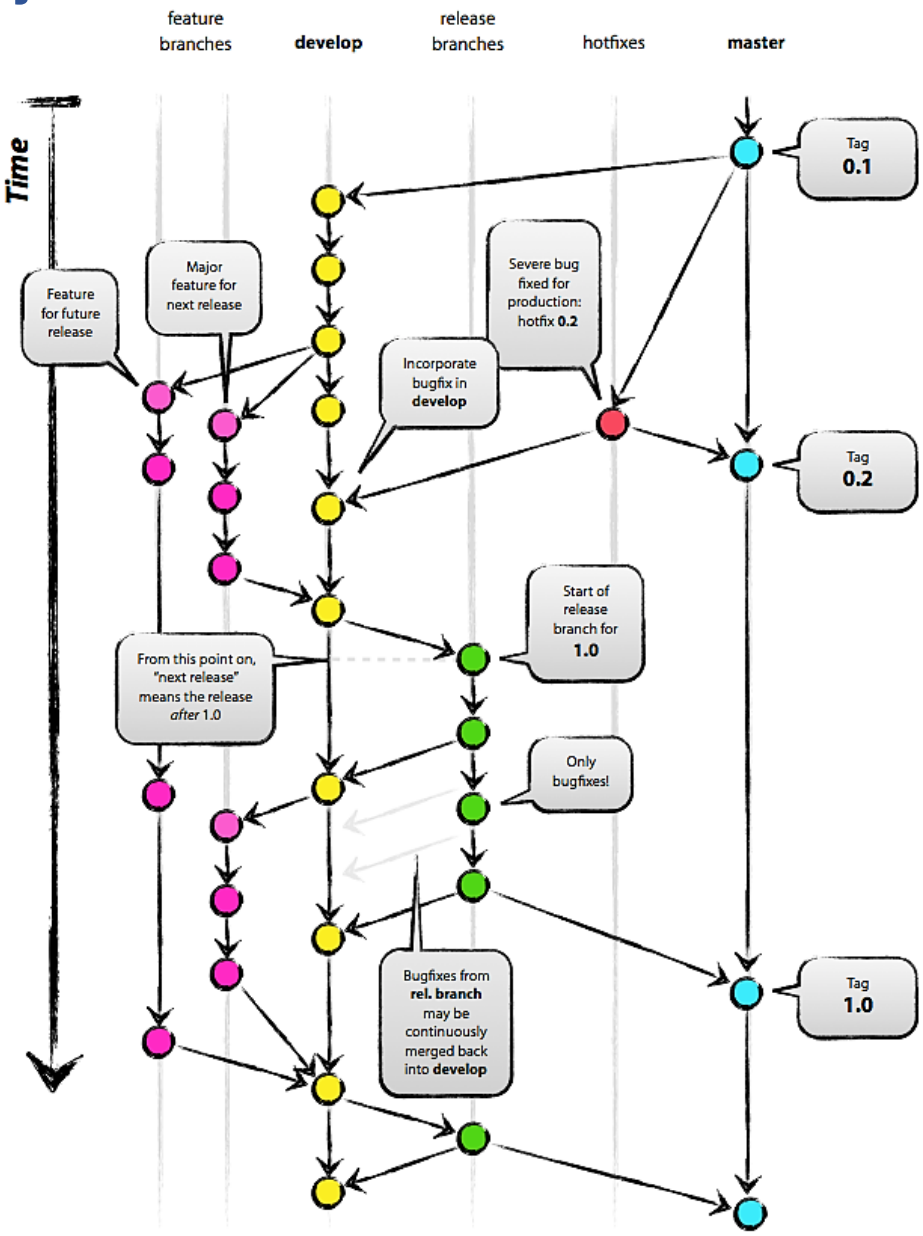
Mejores prácticas:

- Es conveniente hacer commits atómicos:
- El nombre o mensaje debe ser significativo, si para explicar lo que se hizo se necesitan más de dos líneas tal vez es que nuestro commit es muy grande.
- Cada nodo tiene todo el repositorcdio y se puede comunicar con cualquier otro repositorio.
- Git hace track del repositorio local y remoto.
- Leer el libro de Calidad en el Desarrollo de Software, Guillermo Pantaleo, sección 6.2.1 Trabajo con el repositorio, pag. 145.

Trabajando Remotamente

Para trabajar remotamente se puede utilizar https://github.com o https://about.gitlab.com/

Son repositorios distribuidos que crean repositorios locales con toda la información del remoto, mediante estos repositorios remotos podemos compartir código (haciendo commits, pull y push) sin esperar a terminar todo.
El modelo de trabajo que se aconseja para esto es el indicado en la imagen:

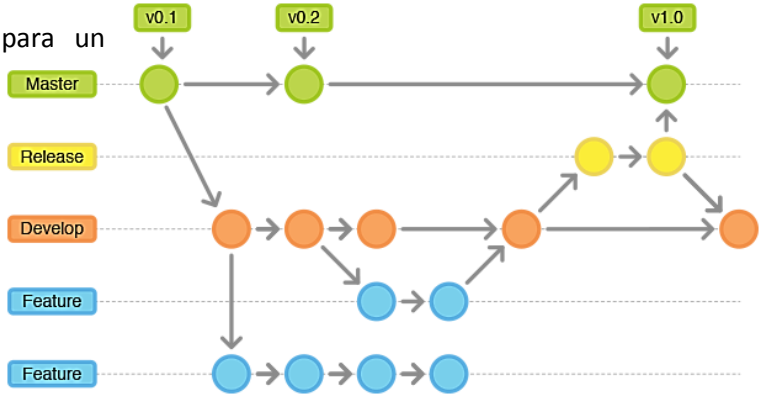


En todo momento en el repositorio del proyecto oficial siempre deben haber dos ramas la rama **master** y la rama **develop**:

Rama Master	La rama `origin/master` es la rama principal donde HEAD siempre es un estado listo de producción. Es una rama de duración infinita, pues ahí se pondrán todos los releases (versiones para producción). Esta es la rama oficial que almacena la historia de releases. Es conveniente etiquetar (tag), cada merge en master con el número de versión. Nunca debemos hacer commits en la rama master, solo merges de develop con master o ramas de release con master. Si el avance es poco significativo, tenemos reléase 0.1. si es muy grande tenemos la versión 1.0
Rama Develop	La rama `origin/develop` es la rama principal de desarrollo donde `HEAD` siempre refleja el último estado entregado de desarrollo, para el siguiente release. Algunos llaman a esta rama de integración, puesto que sirve para integrar features. Es una rama de duración infinita. En esta rama tampoco se deben hacer commits, solo merges con las ramas de features, y solo cuando una rama feature considere que terminó su trabajo.
Rama Features	Es una rama que debe partir de develop y debe volver a develop. Esta rama termina su su razón de ser cuando se hace el merge con develop (y debe destruirse). Si fue un experimento fallido solo se la borra. Las ramas features no deben interactuar con la rama master.

	Las ramas de features solo existen en los repositorios locales no en “origin”. En estas ramas cada desarrollador trabaja en su feature. Algunas buenas prácticas son: • Al empezar el día hacer un pull, puesto que alguien pudo afectar develop y esos cambios pueden afectarme. En lugar de hacer un pull y merge, se puede hacer un **rebase** • Hacer commtis diarios, locales en mi propia branch • Darle un nombre significativo a cada commit: $ git commit –m “nombre Comit” Si se “metió la pata” se puede volver atrás con $ git checkout “nombre commit” (donde nombre commit son los primeros 5 caracteres de los número que aparecen como título del commit cuando se hace $git log Recordar que si alguien hizo un merge que no funciona con develop o master todos y cada uno tienen una copia exacta de master y develop. Por lo que se puede arreglar simplemente haciendo un push de lo que se tiene y el repositorio volverá al estado del repo local que hizo el push.

Release Branches
https://www.atlassian.com/git/workflows#!workflow-gitflow



Cuando develop ya tiene suficientes features para un release o la fecha de release se está aproximando, se debe hacer un fork a la rama release de develop. Creada esta rama empieza el ciclo del siguiente reléase, por lo que ya no se pueden añadir nuevas features, solo – bug fixed, se genera la documentación y otras tareas relacionadas con el release. Una vez listo el release se hace un merge con develop, y debe ser etiquedado con con un número de versión.
Luego de esto el release debe hacer un merge con develop, que seguramente tiene otros progresos desde que se inicio el release.

Usar una rama para releases hace posible que un equipo trabaje para mejorar (pulir) la versión actual, mientras que otro equipo continúa trabajando en características para la próxima versión.
Convenciones:

- branch off: `develop`
- merge into: `master`
- naming convention: `release-* or release/*,  feature-* or feaature/*`

Trabajando con repositorios remotos

Para poder colaborar en cualquier proyecto Git, necesitas saber cómo gestionar tus repositorios remotos. 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.
Gestionar repositorios remotos implica conocer cómo añadir repositorios nuevos, eliminar aquellos que ya no son válidos, gestionar ramas remotas e indicar si están bajo seguimiento o no, y más cosas. En esta sección veremos todos estos conceptos.
Mostrando tus repositorios remotos
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.

$ git clone git://github.com/schacon/ticgit.git	Creará un repositorio local del repositorio remoto indicado en la url.
$git remote $git remote –v	Dentro el repo mostrará origin que es el nombre predeterminado que la da Git al servidor clonado. La opción –v mostrará las url y si puedes hacer push o fetch

$git remote add [nomRemoto] [url]	Da un nombre al repositorio remote especificado en la url Y con ese nombre se puede referenciar en los comandos git siguientes

$git fetch nomRemoto
$git fetch origin
`$git checkout -b rama origin/rama`
 	
`[r] git checkout --track origin/r`
	*Recupera los datos del proyecto remoto que no tengas todavía. Y las ramas solo como punteros, referencias, pero no puedes modificar.
	`*Crear rama en nuestro repositorio local, basada en la del repositorio remoto origin/rama, y ahí si podemos modificar. Además al igual que en origin y develop, rama hará seguimiento de orgin/rama, es decir se puede ejecutar pull y push.`
	*Preparar otras ramas de seguimiento, ahora se puede hacer pull y push directamente de r.

$git push origin master	Envía rama master al servidor origin (origin es el nombre que se da a nuestro repositorio remoto). Este comando funciona si has clonado un repositorio en el que tienes permiso de escritura
$ git remote show origin	Tienes información de un repositorio remoto en particular, en el ejemplo origin
$ git remote rename pb paul	Renombra el repositorio remote pb
$ git remote rm paul	Ellimina el repositorio remoto paul

Creando etiquetas

Como muchos VCSs, 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). En esta sección aprenderás cómo listar las etiquetas disponibles, crear nuevas etiquetas y qué tipos diferentes de etiquetas hay.

Git usa dos tipos principales de etiquetas: ligeras y anotadas. Una etiqueta ligera es muy parecida a una rama que no cambia —un puntero a una confirmación específica—. Sin embargo, las etiquetas anotadas son almacenadas como objetos completos en la base de datos de Git. Tienen suma de comprobación; contienen el nombre del etiquetador, correo electrónico y fecha; tienen mensaje de etiquetado; y pueden estar firmadas y verificadas con GNU Privacy Guard (GPG). Generalmente se recomienda crear etiquetas anotadas para disponer de toda esta información; pero si por alguna razón quieres una etiqueta temporal y no quieres almacenar el resto de información, también tiene disponibles las etiquetas ligeras.

git tag git tag –l ‘v1.4.2.*’	Lista etiquetas Lista etiquetas con el patrón especificado
git tag –a v1.4 –m ‘my versión 1.4’	Pone v1.4 como etiqueta anotada (-a) con el mensaje ´my version´, si no ponemos mensajes nos mostrará el editor de texto
Git show v1.4	Muestra los datos de la etiquetada v1.4, autor, fecha y mensaje
`Git tag -s v1.5 -m 'my signed 1.5 tag'`	Crea la etiqueta v1.5 como una etiqueta firmada, GPG siempre que se tenga una clave privada
`$git tag v1.4-lw`	Crea una etiqueta ligera llamada v1.4-lw
`git tag -a v1.2 -m 'version 1.2' 9fceb02`	Pone una etiqueta una confirmación (commit) pasada: 9fceb02
`git push origin v1.5` `git push origin --tags`	Pone la etiqueta al repositorio remoto (origin) Pone todas las etiquetas de nuestro repositorio local al remoto

Trabajando con ramas

Para más información de cómo el git trabaja con ramas ver: http://git-scm.com/book/es/Ramificaciones-en-Git
Para ver más detalladamente el modelo de ramas se recomienda ver:
http://nvie.com/posts/a-successful-git-branching-model/

Comandos para trabajar con ramas

`[d]git branch ramaNew`
`[d]git checkout ramNew`
[d]git checkout -b nombre_Rama
	Creas ramaNew
	Pasas a trabajar en ramaNew
	Crea una rama y se posiciona en la rama creada.

[r]git push origin nombre_Rama	Sube los commits en una rama
[r]git pull origin nombre_Rama	Baja y hace merge del contenido (rama) repositorio remoto con el local

[d]git pull origin develop [d]git pull --rebase origin develop	Baja y hace merge del contenido (develop) repositorio remoto con el local (d) Equivale a git fetch y git merge FETCH_HEAD. En lugar de hacer un merge, hace un rebase

[rama] `$git rebase master`
`Ejem:`
`[m]git checkout r1`
`[r1]git rebase master`
`[r1]git checkout master`
`[m]git merge r1`
	Pone todos los commits uno por uno en master y si hay conflictos, se los va resolviendo secuencialmente.
	
	1.Va a r1 y luego aplica en master todos los commits de r1 secuencialemente
	2. en rama master hace un merge, que en realidad luego del rebase es un fast forward (actualiza el puntero de master que apunte al HEAD de r1).

git branch -d nombre_Rama	Eliminar una rama, hay que estar fuera de la rama
git branch -D nombre_Rama	Eliminar una rama, forzando borrado cuando rama no ha hecho todavía merge y se pierde todo lo de la rama
git push origin :rama	Borra rama de repo remoto llamado origin
[d] git merge –no-ff rama	Hacer merge con rama
[d]git push origin develop	Actualiza el repositorio remoto con el local (fetch y merge)

Recomendaciones:

1. Cada día antes de hacer cambio ver si ha habido cambios en develop
2. Si ha habido cambios en develop, bajar los cambios a develop
3. Si hubo cambios en rama hacer merge con develop
4. En rama, Al finalizar el dia hacer push de lo que hice, a mi rama
5. Al finalizar mi feature hacer 2 y3, para luego hacer 4 y
    - En develop hacer merge con la feature
    - Hacer push de develop

Ejercicio 2:

En grupos de 3 o 4 personas, lean cuidadosamente las recomendaciones y asegúrense de mantener el branching-model expuesto:

1. Creen un repositorio en github.com ejercicio2
2. Poner la rama develop como rama por defecto
3. Cada uno clone el repo en su máquina
4. Cree una rama a partir de develop con el nombre de feature/nombreOperacion
5. Una vez en la nueva rama, en el lenguaje que elegimos, cada integrante en un archivo llamado aritmética realice una funciones de operaciones aritméticas
6. Terminada la función, realicen el commit correspondiente en su rama local
7. Antes de subir al repositorio la rama, lean las recomendaciones suban al repositorio la rama
8. Realicen el respectivo merge en su repositorio local, de acuerdo a las recomendaciones y suban develop al repositorio compartido

Anexo1- Chapter 3: Ramificaciones en Git

Página principal de git: http://git-scm.com/
Página donde esta este documento completo: http://git-scm.com/book/es/Ramificaciones-en-Git

Cualquier sistema de control de versiones moderno tiene algún mecanismo para soportar distintos ramales. Cuando hablamos de ramificaciones, significa que tú has tomado la rama principal de desarrollo (master) y a partir de ahí has continuado trabajando sin seguir la rama principal de desarrollo. En muchas sistemas de control de versiones este proceso es costoso, pues a menudo requiere crear una nueva copia del código, lo cual puede tomar mucho tiempo cuando se trata de proyectos grandes.

Algunas personas resaltan que uno de los puntos más fuertes de Git es su sistema de ramificaciones y lo cierto es que esto le hace resaltar sobre los otros sistemas de control de versiones. ¿Porqué esto es tan importante? La forma en la que Git maneja las ramificaciones es increíblemente rápida, haciendo así de las operaciones de

ramificación algo casi instantáneo, al igual que el avance o el retroceso entre distintas ramas, lo cual también es tremendamente rápido. A diferencia de otros sistemas de control de versiones, Git promueve un ciclo de desarrollo donde las ramas se crean y se unen ramas entre sí, incluso varias veces en el mismo día. Entender y manejar esta opción te proporciona una poderosa y exclusiva herramienta que puede, literalmente, cambiar la forma en la que desarrollas.

3.1.Ramificaciones en Git - ¿Qué es una rama?

¿Qué es una rama?

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 solo diferencias), sino que los 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).

Para ilustrar esto, vamos a suponer, por ejemplo, que tienes una carpeta con tres archivos, que preparas (stage) todos ellos y los confirmas (commit). Al preparar los archivos, Git realiza una suma de control de cada uno de ellos (un resumen SHA-1, tal y como se mencionaba en el capítulo 1), almacena una copia de cada uno en el repositorio (estas copias se denominan "blobs"), y guarda cada suma de control en el área de preparación (staging area):

```
$ git add README test.rb LICENSE
```

```
$ git commit -m 'initial commit of my project'
```

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 guarda en el repositorio Git una copia de cada uno de los archivos contenidos en ella/s. Después, Git crea un objeto de confirmación con los metadatos pertinentes y un apuntador al nodo correspondiente del árbol de proyecto. Esto permitirá poder regenerar posteriormente dicha instantánea cuando sea necesario.

En este momento, el repositorio de Git contendrá cinco objetos: un "blob" para cada uno de los tres archivos, un árbol con la lista de contenidos de la carpeta (más sus respectivas relaciones con los "blobs"), y una confirmación de cambios (commit) apuntando a la raiz de ese árbol y conteniendo el resto de metadatos pertinentes. Conceptualmente, el contenido del repositorio Git será algo parecido a la Figura 3-1

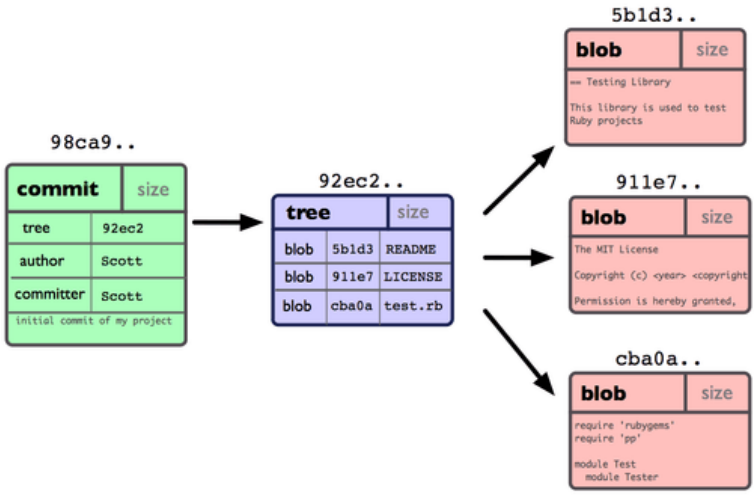


Figura 3-1. Datos en el repositorio tras una confirmación sencilla.

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, el registro ha de ser algo parecido a la Figura 3-2.

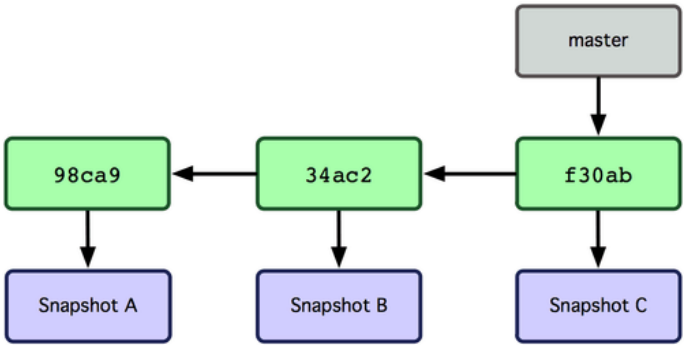


Figura 3-2. Datos en el repositorio tras una serie de confirmaciones:

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.

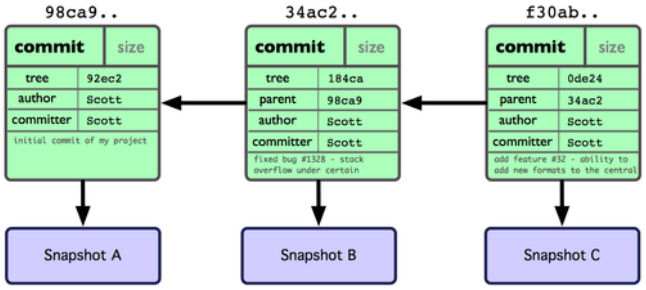


Figura 3-3. Apuntadores en el registro de confirmaciones de una rama:

¿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
```

Esto creará un nuevo apuntador apuntando a la misma confirmación donde estés actualmente (ver Figura 3-4).

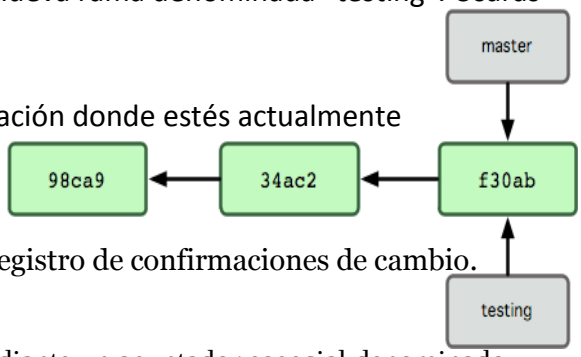


Figura 3-4. Apuntadores de varias ramas en el registro de confirmaciones de cambio.

Y, ¿cómo sabe Git en qué rama estás en este momento? Pues..., 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.

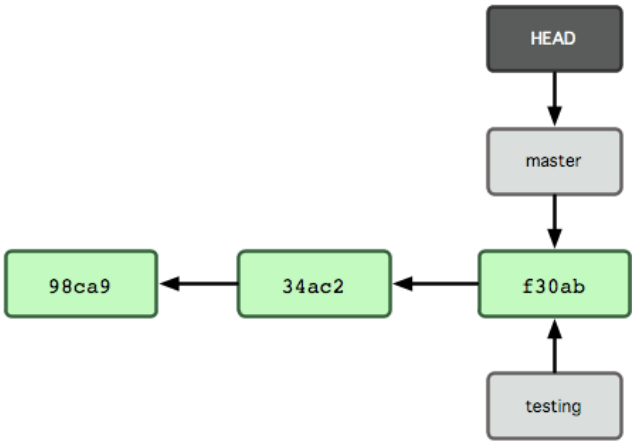


Figura 3-5. Apuntador HEAD a la rama donde estás actualmente:

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`

Esto mueve el apuntador HEAD a la rama `testing` (ver Figura 3-6).

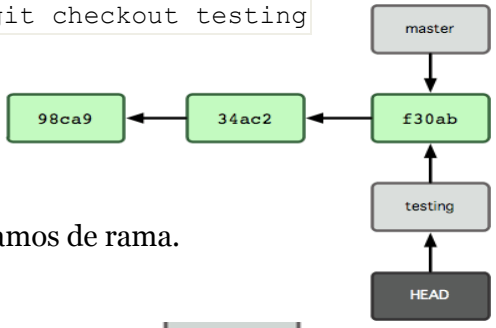
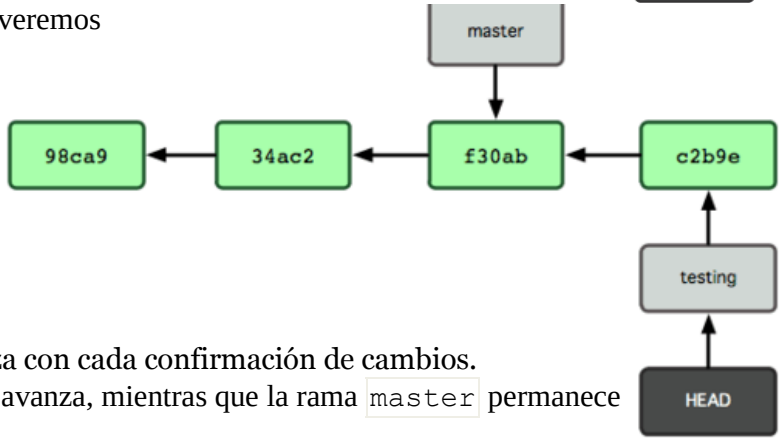


Figura 3-6. Apuntador HEAD apuntando a otra rama cuando saltamos de rama.

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

```
$ vim test.rb
$ git commit -a -m 'made a change'
```



La Figura 3-7 ilustra el resultado:

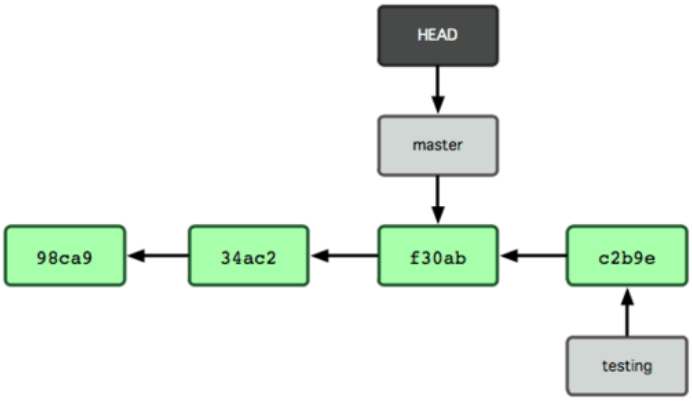
Figura 3-7. La rama apuntada por HEAD avanza con cada confirmación de cambios.

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
```



La Figura 3-8 muestra el resultado:

Figura 3-8. HEAD apunta a otra rama cuando hacemos un checkout.

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. Haz algunos cambios más y confirmalos:

```
$ vim test.rb
```

```
$ git commit -a -m 'made other changes'
```

Ahora el registro de tu proyecto diverge (ver Figura 3-9). Has creado una rama y saltado a ella, has trabajado sobre ella; has vuelto a la rama original, y has trabajado también sobre ella. Los cambios realizados en ambas sesiones de trabajo están aislados en ramas independientes: puedes saltar libremente de una a otra según estimes oportuno. Y todo ello simplemente con dos comandos: `git branch` y `git checkout`.

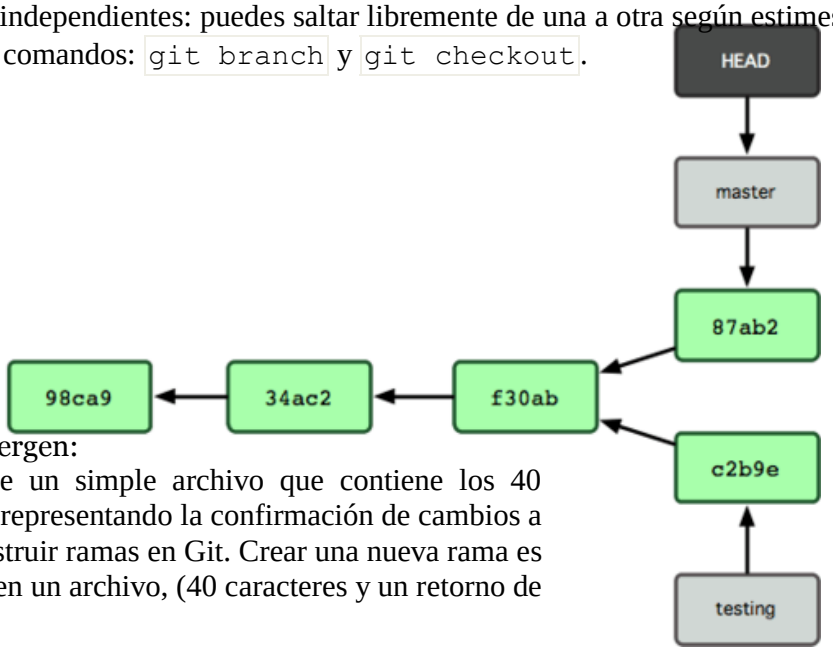


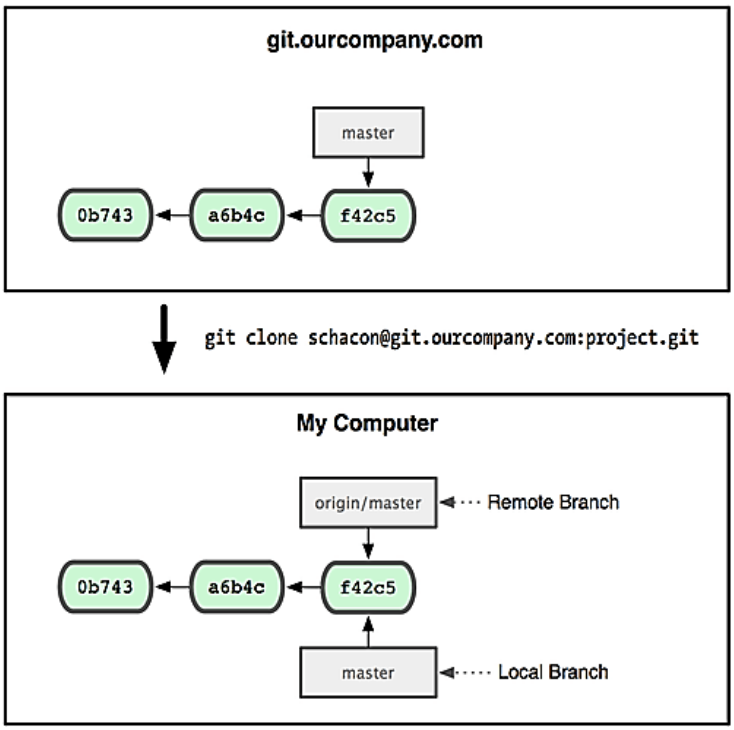
Figura 3-9. Los registros de las ramas divergen:

Debido a que una rama Git es realmente un simple archivo que contiene los 40 caracteres de una suma de control SHA-1, (representando la confirmación de cambios a la que apunta), no cuesta nada el crear y destruir ramas en Git. Crear una nueva rama es tan rápido y simple como escribir 41 bytes en un archivo, (40 caracteres y un retorno de carro).

Esto contrasta fuertemente con los métodos de ramificación usados por otros sistemas de control de versiones. En los que crear una nueva rama supone el copiar todos los archivos del proyecto a una nueva carpeta adicional. Lo que puede llevar segundos o incluso minutos, dependiendo del tamaño del proyecto. Mientras que en Git el proceso es siempre instantáneo. Y, además, debido a que se almacenan también los nodos padre para cada confirmación, el encontrar las bases adecuadas para realizar una fusión entre ramas es un proceso automático y generalmente sencillo de realizar. Animando así a los desarrolladores a utilizar ramificaciones frecuentemente.

3.5 Ramificaciones en Git - Ramas Remotas

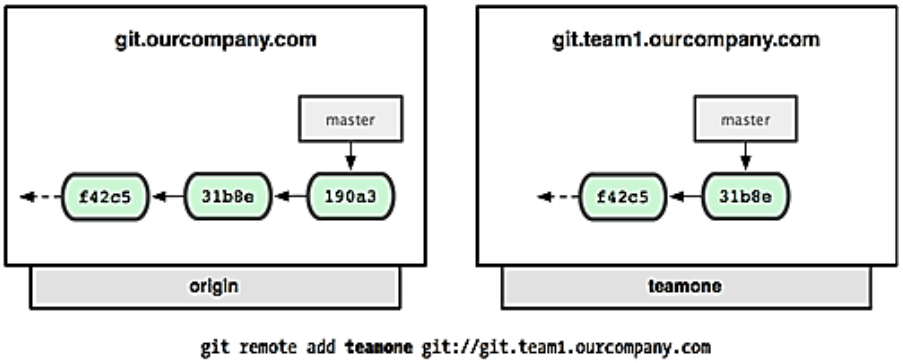
Ramas Remotas



Las ramas remotas son referencias al estado de ramas en tus repositorios remotos. Son ramas locales que no puedes mover; se mueven automáticamente cuando estableces comunicaciones en la red. Las ramas remotas funcionan como marcadores, para recordarte en qué estado se encontraban tus repositorios remotos la última vez que conectaste con ellos.

Suelen referenciarse como `(remoto)/(rama)`. Por ejemplo, si quieres saber cómo estaba la rama `master` en el remoto `origin`. Puedes revisar la rama `origin/master`. O si estás trabajando en un problema con un compañero y este envia (push) una rama `iss53`, tu tendrás tu propia rama de trabajo local `iss53`; pero la rama en el servidor apuntará a la última confirmación (commit) en la rama `origin/iss53`.

Esto puede ser un tanto confuso, pero intentemos aclararlo con un ejemplo. 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.



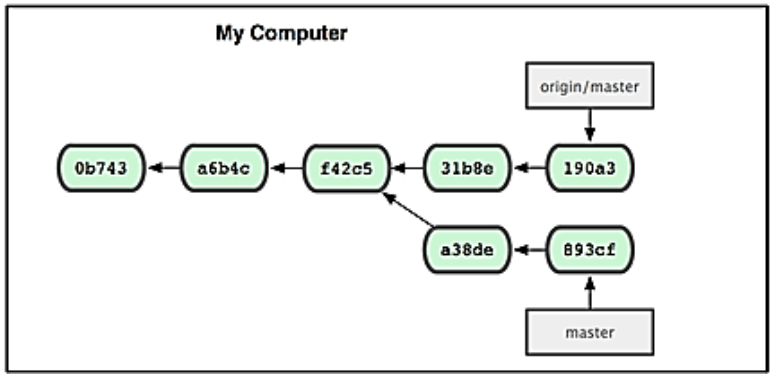
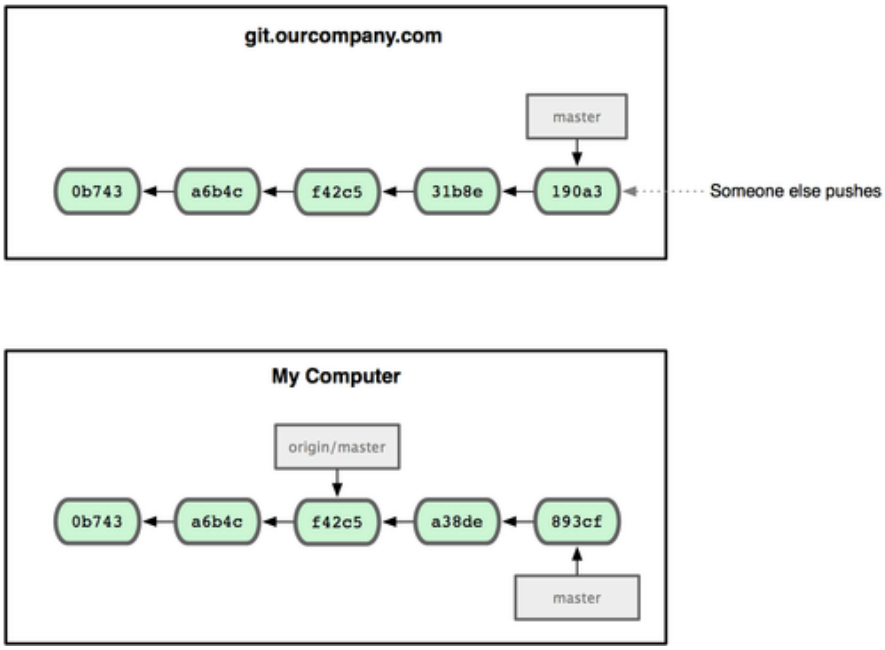


Figura 3-22. Un clón Git te proporciona tu propia rama `master` y otra ama `origin/master` apuntando a la rama `master` original.

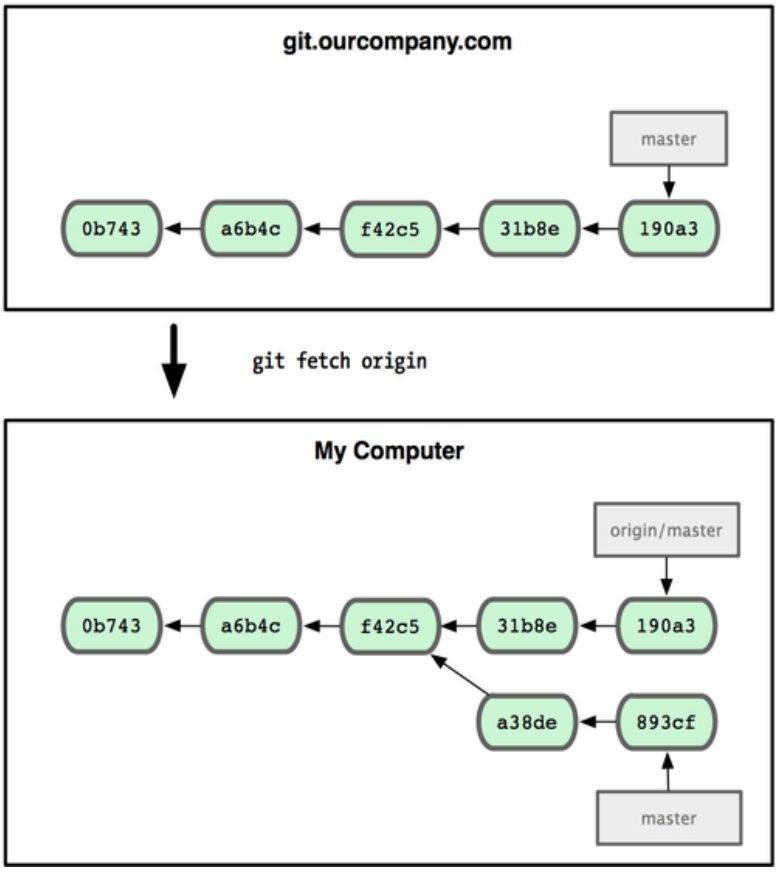
Si haces algún trabajo en tu rama `master` local, y al mismo tiempo, alguna otra persona lleva (push) su trabajo al servidor `git.ourcompany.com`, actualizando la rama `master` de allí, te encontrarás con que ambos registros avanzan de forma diferente. Además, mientras no tengas contacto con el servidor, tu apuntador a tu rama `origin/master` no se moverá (ver Figura 3/23).

Figura 3-23. Trabajando localmente y que otra persona esté llevando (push) algo al servidor remoto, hace que cada registro avance de forma distinta.



Para sincronizarte, puedes utilizar el comando `git fetch origin`. Este comando localiza en qué servidor está el origen (en este caso `git.ourcompany.com`), recupera cualquier dato presente allí que tu no tengas, y actualiza tu base de datos local, moviendo tu rama `origin/master` para que apunte a esta nueva y más reciente posición (ver Figura 3-24).

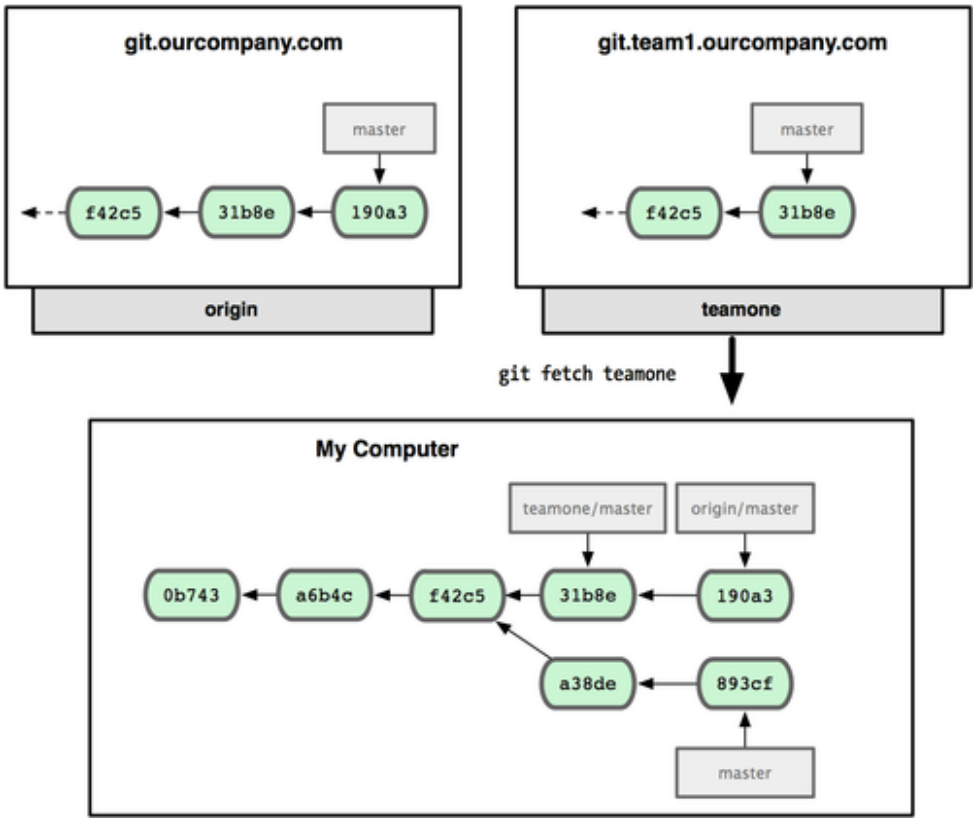
Figura 3-24. El comando `git fetch` actualiza tus referencias remotas.



Para ilustrar mejor el caso de tener múltiples servidores y cómo van las ramas remotas para esos proyectos remotos. Supongamos que tienes otro servidor Git; utilizado solamente para desarrollo, por uno de tus equipos sprint. Un servidor en `git.team1.ourcompany.com`. Puedes incluirlo como una nueva referencia remota a tu proyecto actual, mediante el comando `git remote add`, tal y como vimos en el capítulo 2. Puedes denominar `teamone` a este

remoto, poniendo este nombre abreviado para la URL (ver Figura 3-25)

Figura 3-25. Añadiendo otro servidor como remoto.



Ahora, puedes usar el comando `git fetch teamone` para recuperar todo el contenido del servidor que tu no tenias. Debido a que dicho servidor es un subconjunto de los datos del servidor `origin` que tienes actualmente, Git no recupera (fetch) ningún dato; simplemente prepara una rama remota llamada `teamone/master` para apuntar a la confirmación (commit) que `teamone` tiene en su rama `master`.

Figura 3-26. Obtienes una referencia local a la posición en la rama `master` de `teamone`.

Publicando

Cuando quieres compartir una rama con el resto del mundo, has de llevarla (push) a un remoto donde tengas permisos de escritura. Tus ramas locales no se sincronizan automáticamente con los remotos en los que escribes. Sino que tienes que llevar (push) expresamente, cada vez, al remoto las ramas que desees compartir. De esta forma, puedes usar ramas privadas para el trabajo que no deseas compartir. Llevando a un remoto tan solo aquellas partes que deseas aportar a los demás.

Si tienes una rama llamada `serverfix`, con la que vas a trabajar en colaboración; puedes llevarla al remoto de la misma forma que llevaste tu primera rama. Con el comando :

`git push (remoto) (ramaLocal)[: ramaRemotaOpcional]`

```
$ git push origin serverfix
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      serverfix -> serverfix
```

Esto es un poco como un atajo. Git expande automáticamente el nombre de rama `serverfix` a `refs/heads/serverfix:refs/heads/serverfix`, que significa: "coge mi rama local `serverfix` y actualiza con ella la rama `serverfix` del remoto". Volveremos más tarde sobre el tema de `refs/heads/`, viéndolo en detalle en el capítulo 9; aunque puedes ignorarlo por ahora. También puedes hacer **`git push origin serverfix:serverfix`, que hace lo mismo**; es decir: "coge mi `serverfix` y hazlo el `serverfix` remoto". Puedes utilizar este último formato para llevar una rama local a una rama remota con otro nombre distinto. Si no quieres que se llame `serverfix` en el remoto, puedes lanzar, por ejemplo, `git push origin serverfix:awesomebranch`; para llevar tu rama `serverfix` local a la rama `awesomebranch` en el proyecto remoto.

La próxima vez que tus colaboradores recuperen desde el servidor, obtendrán una referencia a donde la versión de `serverfix` en el servidor esté bajo la rama remota `origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 20, done.
```

```
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

Es importante destacar que cuando recuperas (fetch) nuevas ramas remotas, no obtienes automáticamente una copia editable local de las mismas. En otras palabras, en este caso, no tienes una nueva rama `serverfix`. Sino que únicamente tienes un puntero no editable a `origin/serverfix`.

Para integrar (merge) esto en tu actual rama de trabajo, puedes usar el comando `git merge origin/serverfix`. Y si quieres tener tu propia rama `serverfix`, donde puedas trabajar, puedes crearla directamente basaádote en rama remota:

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"Switched to a new branch "serverfix"
```

Esto sí te da una rama local donde puedes trabajar, comenzando donde `origin/serverfix` estaba en ese momento.

Haciendo seguimiento a las ramas

Activando (checkout) una rama local a partir de una rama remota, se crea automáticamente lo que podríamos denominar "una rama de seguimiento" (tracking branch). Las ramas de seguimiento son ramas locales que tienen una relación directa con alguna rama remota. Si estás en una rama de seguimiento y tecleas el comando `git push`, Git sabe automáticamente a qué servidor y a qué rama ha de llevar los contenidos. Igualmente, tecleando `git pull` mientras estamos en una de esas ramas, recupera (fetch) todas las referencias remotas y las consolida (merge) automáticamente en la correspondiente rama remota.

Cuando clonas un repositorio, este suele crear automáticamente una rama `master` que hace seguimiento de `origin/master`. Y es por eso que `git push` y `git pull` trabajan directamente, sin necesidad de más argumentos. Sin embargo, puedes preparar otras ramas de seguimiento si deseas tener unas que no hagan seguimiento de ramas en `origin` y que no sigan a la rama `master`. El ejemplo más simple, es el que acabas de ver al lanzar el comando `git checkout -b [rama] [nombreremoto]/[rama]`. Si tienes la versión 1.6.2 de Git, o superior, puedes utilizar también el parámetro `--track`:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"Switched to a new branch "serverfix"
```

Para preparar una rama local con un nombre distinto a la del remoto, puedes utilizar:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "sf"
```

Así, tu rama local `sf` va a llevar (push) y traer (pull) hacia o desde `origin/serverfix`.

Borrando ramas remotas

Imagina que ya has terminado con una rama remota. Es decir, tanto tu como tus colaboradores habeis completado una determinada funcionalidad y la habeis incorporado (merge) a la rama `master` en el remoto (o donde quiera que tengais la rama de código estable). Puedes borrar la rama remota utilizando la un tanto confusa sintaxis: `git push [nombreremoto] :[rama]`. Por ejemplo, si quieres borrar la rama `serverfix` del servidor, puedes utilizar:

```
$ git push origin :serverfix
To git@github.com:schacon/simplegit.git
 - [deleted]         serverfix
```

Y... ¡Boom!. La rama en el servidor ha desaparecido. Puedes grabarte a fuego esta página, porque necesitarás ese comando y, lo más probable es que hayas olvidado su sintaxis. Una manera de recordar este comando es dándonos cuenta de que proviene de la sintaxis `git push [nombreremoto]`

`[ramalocal]:[ramaremota]`. Si omites la parte `[ramalocal]`, lo que estás diciendo es: "no cojas nada de mi lado y haz con ello [ramaremota]".

3.6 Ramificaciones en Git - Reorganizando el trabajo realizado

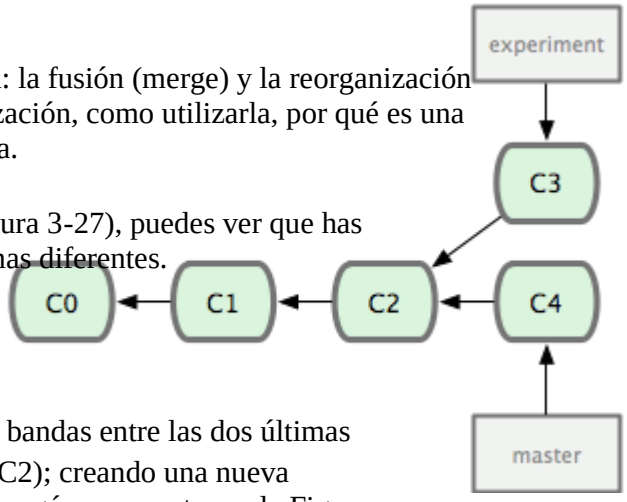
Reorganizando el trabajo realizado

En Git tenemos dos formas de integrar cambios de una rama en otra: la fusión (merge) y la reorganización (rebase). En esta sección vas a aprender en qué consiste la reorganización, como utilizarla, por qué es una herramienta sorprendente y en qué casos no es conveniente utilizarla.

Reorganización básica

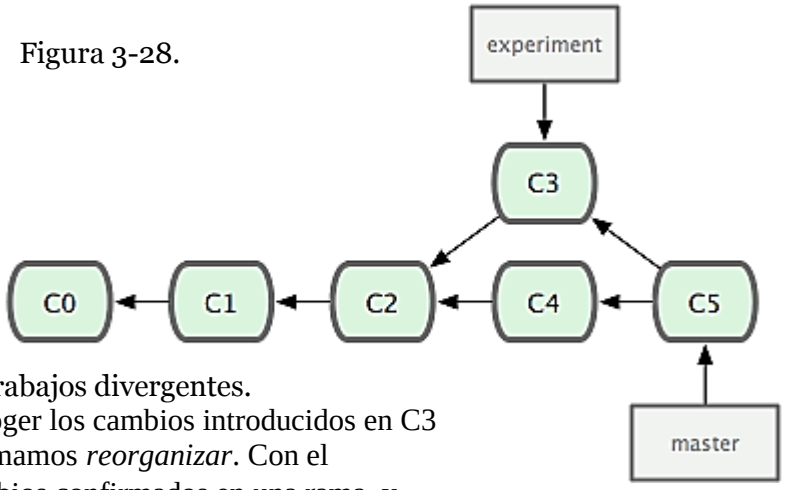
Volviendo al ejemplo anterior, en la sección sobre fusiones (ver Figura 3-27), puedes ver que has separado tu trabajo y realizado confirmaciones (commit) en dos ramas diferentes.

Figura 3-27. El registro de confirmaciones inicial.



La manera más sencilla de integrar ramas, tal y como hemos visto, es el comando `git merge`. Realiza una fusión a tres bandas entre las dos últimas instantáneas de cada rama (C3 y C4) y el ancestro común a ambas (C2); creando una nueva instantánea (snapshot) y la correspondiente confirmación (commit), según se muestra en la Figura 3-28.

Figura 3-28.



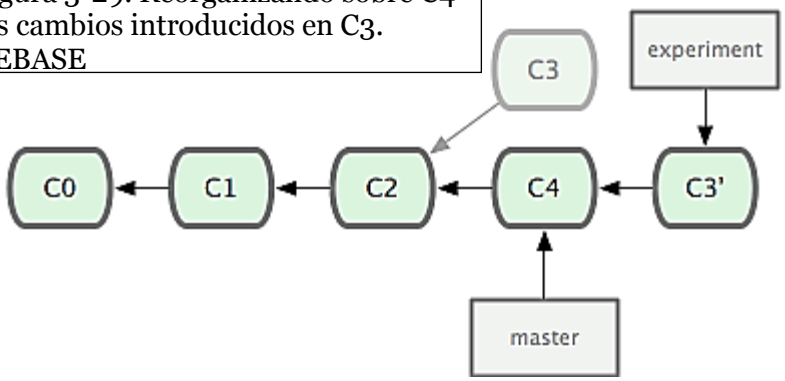
Fusionando una rama para integrar el registro de trabajos divergentes.

Aunque también hay otra forma de hacerlo: puedes coger los cambios introducidos en C3 y reaplicarlos encima de C4. Esto es lo que en Git llamamos *reorganizar*. Con el comando `git rebase`, puedes coger todos los cambios confirmados en una rama, y reaplicarlos sobre otra.

Por ejemplo, puedes lanzar los comandos:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay
your work on top of it...
Applying: added staged command
```

Figura 3-29. Reorganizando sobre C4 los cambios introducidos en C3. REBASE



Haciendo que Git: vaya al ancestro común de ambas ramas (donde estás actualmente y de donde quieres reorganizar), saque las diferencias introducidas por cada confirmación en la rama donde estás, guarde esas diferencias en archivos temporales, reinicie (reset) la rama actual hasta llevarla a la misma confirmación en la rama de donde quieres reorganizar, y, finalmente, vuelva a aplicar ordenadamente los cambios. El proceso se muestra en la Figura 3-29.

En este momento, puedes volver a la rama `master` y hacer una fusión con avance rápido (fast-forward merge). (ver Figura 3-30)

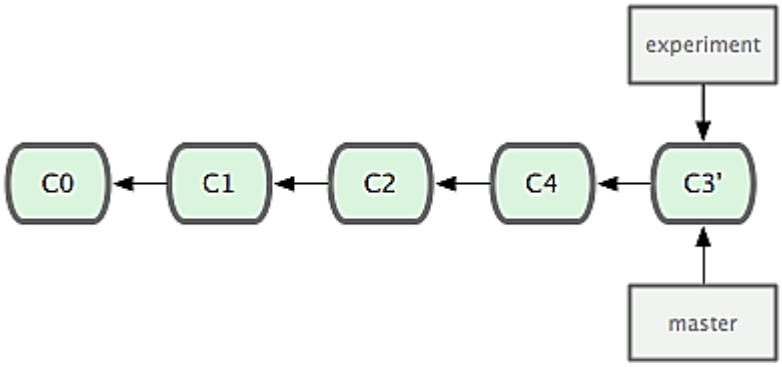


Figura 3-30. Avance rápido de la rama `master`.

Así, la instantánea apuntada por C3' aquí es exactamente la misma apuntada por C5 en el ejemplo de la fusión. No hay ninguna diferencia en el resultado final de la integración, pero el haberla hecho reorganizando nos deja un registro más claro. Si examinas el registro de una rama reorganizada, este aparece siempre como un registro lineal: como si todo el trabajo se hubiera realizado en series, aunque realmente se haya hecho en paralelo.
Habitualmente, optarás por esta vía cuando quieras estar seguro de que tus confirmaciones de cambio (commits) se pueden aplicar limpiamente sobre una rama remota; posiblemente, en un proyecto donde estés intentando colaborar, pero lleves tu el mantenimiento. En casos como esos, puedes trabajar sobre una rama y luego reorganizar lo realizado en la rama `origin/master` cuando lo tengas todo listo para enviarlo al proyecto principal. De esta forma, la persona que mantiene el proyecto no necesitará hacer ninguna integración con tu trabajo; le bastará con un avance rápido o una incorporación limpia.
Cabe destacar que la instantánea (snapshot) apuntada por la confirmación (commit) final, tanto si es producto de una reorganización (rebase) como si lo es de una fusión (merge), es exactamente la misma instantánea. Lo único diferente es el registro. La reorganización vuelve a aplicar cambios de una rama de trabajo sobre otra rama, en el mismo orden en que fueron introducidos en la primera. Mientras que la fusión combina entre sí los dos puntos finales de ambas ramas.