

GIT

CLASE 4 : DESHACIENDO CAMBIOS

REVERSIÓN DE CAMBIOS

Nos va a pasar muchas veces trabajando con **GIT** que simplemente realizamos trabajo que no queremos conservar, bien porque ya no es implementable o porque terminó siendo un **bug**; cómo podemos hacer para revertir cambios que están actualmente en nuestro historial del repositorio?

EL COMANDO REVERT

Siempre tenemos que tener en cuenta, antes de siquiera revertir nada, si los cambios que queremos eliminar ya fueron publicados al repositorio remoto o no. Revertir cambios que otras personas pudieron potencialmente haber descargado ya nos podrá traer problemas a futuro ya que nuestro historial de cambios, es decir el árbol de grafos de hash de commits, seguramente es diferente a la del remoto o mismo a la de mis propios compañeros. En algún momento del trabajo alguien no va a poder o actualizar su propio trabajo o/ni subir cambios al remoto, ya que no se estaría tratando técnicamente del mismo repositorio ya, el grafo entero cambió.

Si este es el caso, vamos a tener a mano el comando `git revert`. Este comando nos permite elegir uno o un conjunto de commits, revertir los cambios que ellos implementan.

`git revert <commit>`



Como podemos ver en el gráfico anterior, tenemos un commit conflictivo en nuestro branch, el **abcd12ef** y queremos revertir sus cambios. Podemos simplemente correr el siguiente comando :

```
> git revert abcd12ef
```

Y veremos que el commit en cuestión no desaparece sino que nuestro historial de commits avanza, generando uno nuevo en donde se refleja nuestro trabajo sin los cambios que pasaron en el commit que seleccionamos.

Tengamos en cuenta que siempre vamos a ser propensos a posibles conflictos que se generen por haber deshecho un cambio previo en alguno de nuestros archivos.

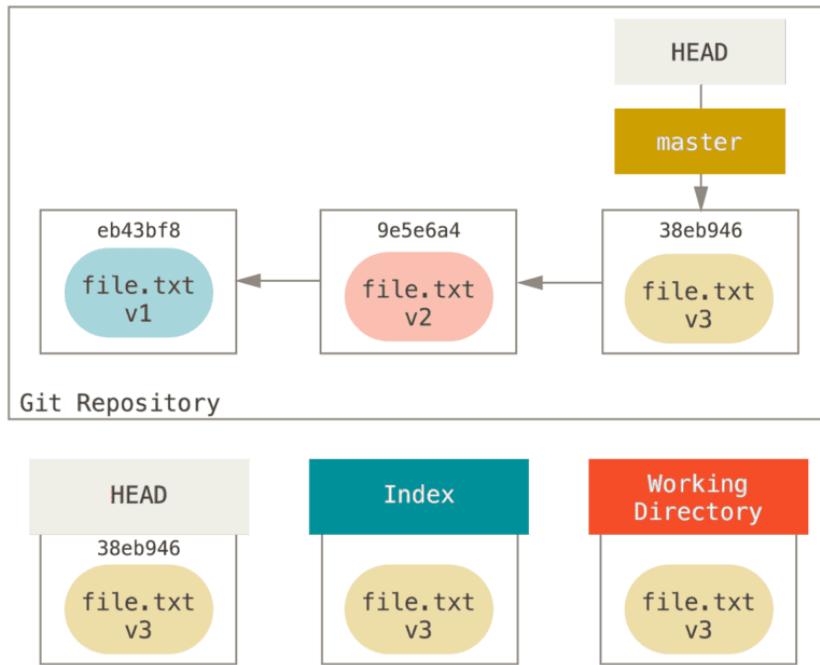
EL COMANDO RESET

Si por otro lado, todo el trabajo que estuvimos haciendo no fue aún sincronizado con el repositorio remoto todavía tenemos la posibilidad de ordenar nuestro historial de cambios y borrar aquellos commits que estuvieron de más.

Para esto podemos usar el comando git reset :

```
> git reset [tipo] <commit>
```

Para entender este comando con más detalle vamos a seguir un ejemplo más claro. Supongamos que estamos trabajando en un repositorio donde tenemos un archivo llamado file.txt al cual ya le hicimos varios cambios :

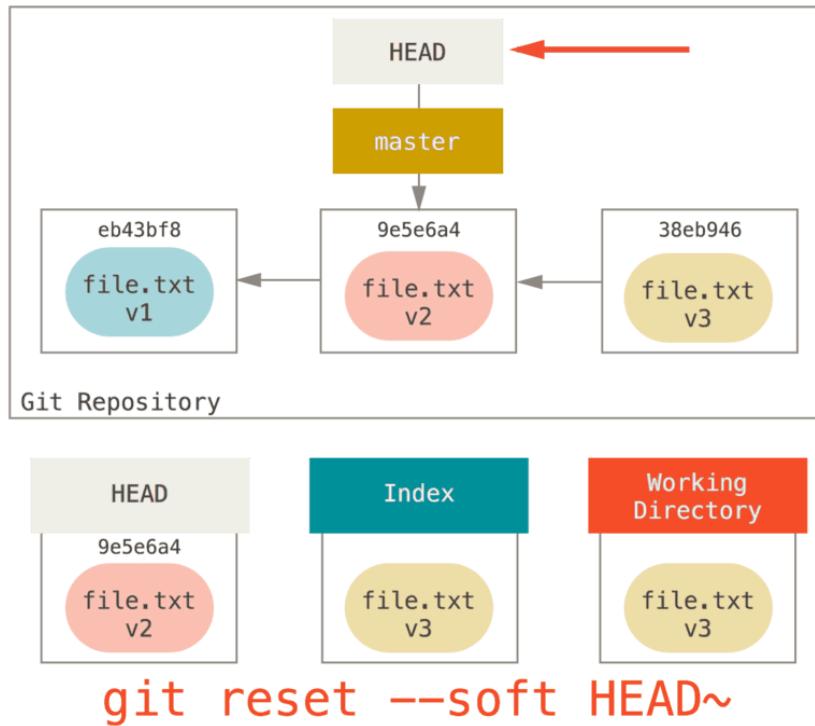


Como podemos ver en el gráfico, nuestro HEAD está actualmente ubicado en el commit 38eb946 viendo la versión 3 de nuestro archivo en el branch master. Por detrás nuestro hay dos commits más, es decir, los dos de las versiones anteriores del mismo archivo.

Supongamos entonces que lo que necesitamos es deshacer el último commit, donde nos encontramos ahora mismo, para volver a una versión anterior del archivo. Podríamos ejecutar el siguiente comando :

```
> git reset --soft 9e5e6a4
```

Primero que todo, el comando de reset va a intentar mover nuestro putero HEAD hasta ese commit que acabamos de especificar. Cómo es esto distinto a hacer un checkout y movernos libremente a ese commit también? Simplemente porque el checkout solo mueve el HEAD permitiéndonos ver nuestro Working Directory en ese momento del tiempo mientras que el reset no solo se encarga de esto sino también de editar el puntero del branch donde estemos trabajando y modificarlo para que tome el mismo cambio :



Lo más probable es que si ejecutamos un `git status` en este momento, podamos todavía observar el cambio que se introdujo en el commit del cual vinimos pero formando parte del Stage Area, tal y como se verían si se hubieran hecho recién.

Podemos notar también dos cosas : la primera es que el puntero del branch donde estábamos trabajando también se movió con nosotros y la segunda es que utilizamos el flag `--soft` para ejecutar el comando.

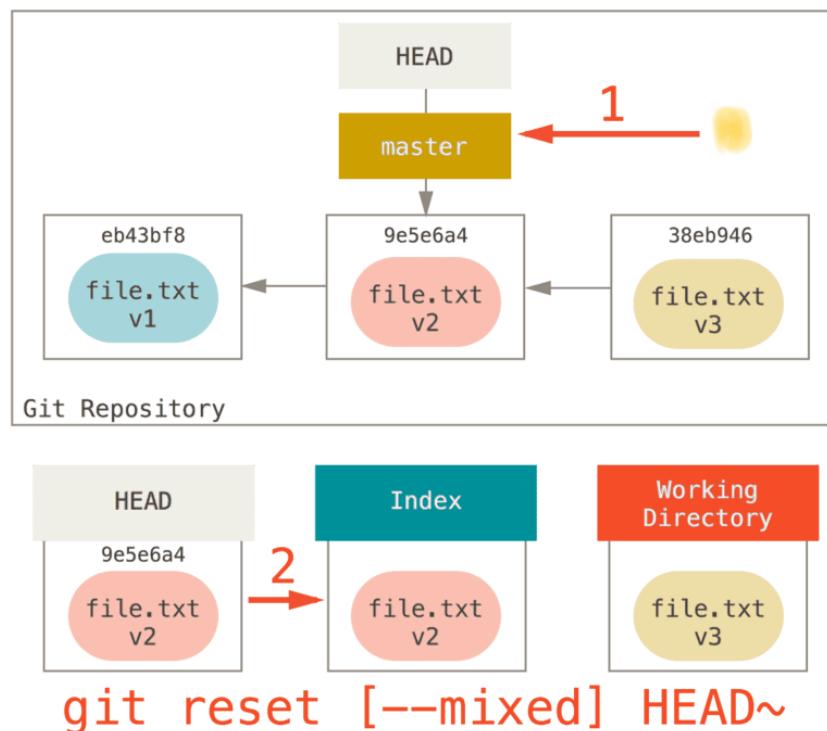
TIPOS DE RESET

Existen tres tipos de reset :

1. `--soft`
2. `--mixed`
3. `--hard`

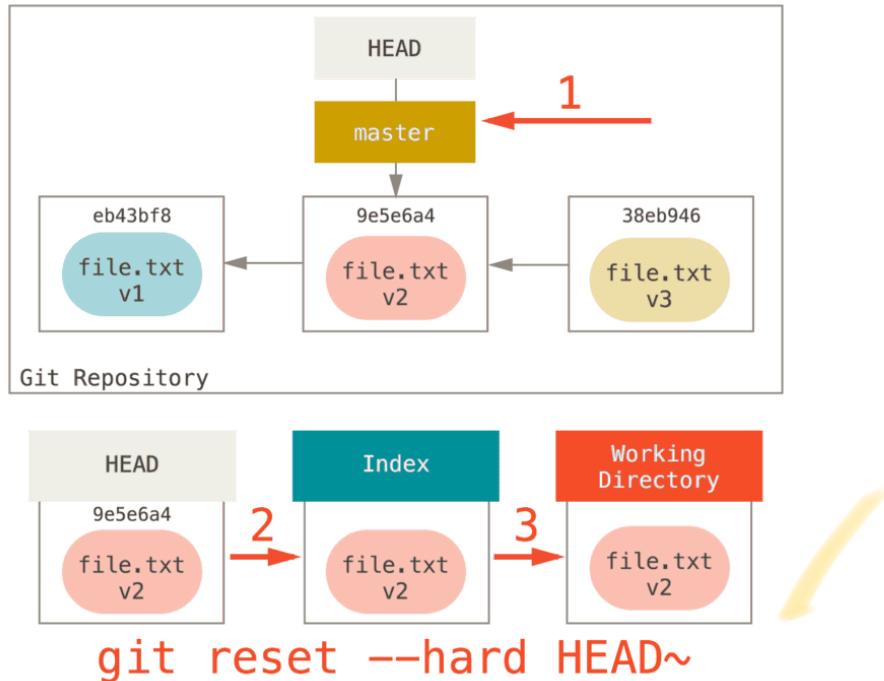
El reset soft es aquel en donde los cambios que hemos intentado resetear aún se mantienen visibles pero ya no forman parte de ningún commit en nuestro branch. Por otro lado, estos cambios se van a ver en nuestros archivos y sobre todo se van a ver reflejados en el Stage Area cuando ejecutemos un `git status` y veamos que esos cambios que intentamos deshacer forman parte del próximo commit, en el caso en que hagamos uno en este mismo momento. Podemos hacer referencia del último gráfico para entender mejor el mecanismo.

El reset mixed es aquel en donde los cambios también se siguen manteniendo visibles en nuestros archivos, es decir que no fueron totalmente deshechos pero si consultamos el status de nuestro repositorio podríamos observar que los mismos no forman parte del Stage Area, es decir que se van a ver como si se hubieran hecho recién pero como si todavía no se hubiera corrido el comando git add :



Por último tenemos el reset hard. Este es bastante particular ya que es el más “peligroso” de todos los tipos. El reset va a comenzar por lo mismo que los demás, corriendo nuestro HEAD al commit establecido pero todos los cambios que se hayan hecho durante ese tiempo no solo no van a formar parte del Stage Area sino que no vamos a verlos tampoco en nuestros archivos actuales, es decir nuestro Working Area.

Podríamos decir que en este caso “perdimos” un commit porque ya no vamos a tener ese libre acceso a los cambios en caso de arrepentirse a tiempo como con los otros dos tipos.



DEPURACIÓN DE REPOSITORIOS

Además de GIT ya ser de por sí una herramienta que nos permite tener un control absoluto de las versiones por las que pasan nuestros archivos, la misma nos da varios otros comandos con los cuales podemos realizar una depuración (debug) un poco más exhaustiva de nuestro código.

ANOTACIÓN DE ARCHIVOS

Supongamos que hemos sido capaces de identificar a través de nuestro código un error (bug) y lo que necesitamos ahora es deducir en qué momento se introdujo dicho cambio, la anotación de archivos o **FILE ANNOTATION** va a ser nuestra mejor amiga. Esta herramienta extra se puede acceder con el comando :

```
> git blame
```

La misma nos permite saber el último commit en el que fue introducida una línea específica de un archivo. Pongamos como ejemplo las primeras líneas de un archivo llamado `server.js` :

```

const express = require('express');
const app = express();
const fetch = require('node-fetch');
const LocalStorage = require('node-localstorage').LocalStorage;
const localStorage = new LocalStorage('./.data');
const fs = require('fs');
const OctokitApp = require('@octokit/app');
const request = require('@octokit/request');

class Server {

  constructor() {
    this.basicStr = 'basic';
    this.oAuthStr = 'OAuth';
    this.serverStr = 'Server-to-Server';
    this.searchStr = 'Search';
    this.userStr = 'User';
    this.state = {
      authType: '', // || 'OAuth' || 'Server-to-Server'
      authTarget: this.searchStr, // || 'User'
      clientId: process.env.GH_CLIENT_ID,
      oAuthToken: localStorage.getItem('oauth'),
      oAuthState: String(Math.random() * 1000000),
      rateLimitRemaining: '',
      rateLimitTotal: '',
      rateResetDate: '',
      serverToken: ''
    };
    this.startup();
    this.api();
  }

}

```

Si contamos desde arriba hacia abajo, vamos a prestar especial atención a la línea 10 en adelante. De esta manera podemos realizar el siguiente comando :

```

> git blame -L 10,20 server.js
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 10) class Server {
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 11)
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 12)   constructor() {
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 13)     this.basicStr = 'basic';
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 14)     this.oAuthStr = 'OAuth';
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 15)     this.serverStr =
'Server-to-Server';

```

```
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 16)      this.searchStr = 'Search';
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 17)      this.userStr = 'User';
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 18)      this.state = {
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 19)          authType: '',
// || 'OAuth' || 'Server-to-Server'
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 20)      authTarget: this.searchStr,
// || 'User'
```

Lo que estamos haciendo es ejecutar el comando git blame con varios parámetros adicionales, en particular **-L 10,20** lo cual le dice al comando que lo único que necesitamos revisar del archivo son las líneas de código que van desde la línea 10 a la 20.

Si analizamos el output que nos da la línea de comandos, podemos observar que se nos está brindando el hash del commit que introdujo cada línea escrita en el archivo, junto con el autor y la fecha y hora de la misma.

En este caso, todas las líneas fueron escritas por una misma persona pero imaginemos el caso en donde se hubiera introducido un cambio en el archivo en la línea 17, con lo cual podemos volver a realizar nuevamente la anotación de archivos para obtener el siguiente output :

```
> git blame -L 10,20 server.js
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 10) class Server {
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 11)
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 12)     constructor() {
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 13)         this.basicStr = 'basic';
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 14)         this.oAuthStr = 'OAuth';
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 15)         this.serverStr =
'Server-to-Server';
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 16)         this.searchStr = 'Search';
7737d6b0 (Horacio Gutierrez 2020-05-31 19:01:17 -0300 17)     this.test = true;
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 18)         this.userStr = 'User';
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 19)         this.state = {
8b638d1e (John Bohannon 2019-06-04 09:12:11 -0400 20)             authType: '',
// || 'OAuth' || 'Server-to-Server'
```

Como podemos observar, se presenta el nombre de un autor distinto al resto en la línea 17 junto con el hash del commit en donde el mismo realizó dicho cambio.

BÚSQUEDA

GIT nos permite realizar varios tipos de búsqueda a través de nuestros repositorios; dos de las más populares son la posibilidad de buscar strings entre nuestros archivos o el hecho de poder buscar strings en nuestro log de commits.

La búsqueda de strings en nuestros archivos la podemos realizar utilizando el siguiente comando :

```
> git grep
```

Con el podemos hacer una búsqueda común como si estuviéramos trabajando en cualquier editor de código. Continuando con el ejemplo anterior del archivo **server.js**, vamos a buscar la línea de código que sabíamos que era la conflictiva en nuestro programa :

```
> git grep this.test  
server.js:    this.test = true;
```

Si prestamos atención al output de la línea de comandos, la misma nos está informando que el string “this.test” se encuentra presente en el archivo server.js. Si estuviera en muchas más instancias, podríamos observar un listado de varios archivos. Podemos utilizar varios parámetros para hacer nuestras búsquedas un poco más interesantes como por ejemplo :

```
> git grep -n this.test  
server.js:17:    this.test = true;
```

Con esta nueva búsqueda podemos observar no solo el nombre del archivo en donde se encuentra nuestro string sino la línea en donde está presente. También podemos realizar algo muy interesante y es :

```
> git grep -c this.test  
server.js:1
```

Con esto podemos buscar cuantas veces se encuentra repetida la línea conflictiva en todos los archivos de nuestro repositorio de forma tal que podamos arreglar el error en todos lados.

Capáz no estamos buscando en dónde un término existe, sino mas bien cuándo existió o fue introducido. Para realizar esto, debemos recordar el comando de **git log**, el cual se vuelve más poderoso si empezamos a trabajar con él utilizando sus parámetros, como por ejemplo :

```
> git log -S this.test  
commit 7737d6b01946e9dafb689de011bfac9da694db2f  
Author: Horacio Gutierrez <horacio.estevez@gmail.com>  
Date:   Sun May 31 19:01:17 2020 -0300  
  
        add new change
```

Si observamos, estamos utilizando el parámetro **-S** con el cual podemos agregar nuestro término a buscar y el mismo nos está retornando un listado de todos los commits en donde dicho cambio o string fue introducido.

REBASE INTERACTIVO

Existe otra manera de poder deshacer cambios y no solo eso, también de poder ordenar, simplificar, juntar y editar nuestros commits y es con el comando de **git rebase**. Si pensamos en la utilidad de dicho comando, seguro recordemos que nos permitía ubicar todo un branch para que su branch de origen sea otro, distinto desde el cual se originó.

Gracias a la herramienta interactiva del rebase podemos entrar en una interfaz distinta la cual nos va a habilitar nuevas opciones para aquellos commits que queremos hacerles rebase :

```
> git rebase -i <commit>
```

Nótese que estamos usando el flag **-i** para poder entrar en esta interfaz interactiva del rebase :

```

pick d2f7add Change 1
pick 390275c Change 2 is the point where you want to replay
pick 48f3b48 Change 3

# Rebase 5d255a1..48f3b48 onto 5d255a1 (3 command(s))
#
# Commands:
#   p, pick = use commit
#   r, reword = use commit, but edit the commit message
#   e, edit = use commit, but stop for amending
#   s, squash = use commit, but meld into previous commit
#   f, fixup = like "squash", but discard this commit's log message
#   x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
<Desktop/rebase/.git/rebase-merge/git-rebase-todo [unix] (00:25 05/09/2015)1,1 ATI>
<Desktop/rebase/.git/rebase-merge/git-rebase-todo [converted] 21L, 681C>

```

Una vez que estemos usando este comando vamos a poder ver todos los commits que había entre nuestro HEAD y el que hayamos elegido pero, a diferencia del git log, en este caso los tenemos ordenados en el orden opuesto, es decir desde el más antiguo(arriba) al más reciente(abajo de todo).

También vamos a ver una serie de comandos listados más abajo :

- **p , pick** : El comando de pick nos permite mover commits de lugar
- **r , reword** : El comando reword nos permite entrar en modo editor y poder volver a escribir el mensaje de ese commit en particular
- **e , edit** : El comando edit nos permite pausar el rebase en ese commit en particular, dándonos tiempo para realizar más cambios que nos hayamos olvidado, guardarlos, confirmar el commit y continuar con el resto de commits del rebase
- **s , squash** : El comando squash nos permite fusionar el commit al cual le estamos aplicando este comando con el anterior. Por lógica podemos ponerle a todos squash menos al primero que aparezca en la lista, dado que no va a tener con quien fusionarse
- **f , fixup** : El comando fixup realiza una operación similar al squash pero este descarta todos los mensajes de los commits que vamos a fusionar

Todo lo que tenemos que hacer es cambiar los prefijos de aquellos commits que queramos editar con alguno de los comandos anteriores y salirnos de la interfaz de VIM :



```

MINGW64:/c/Users/rmackay9/Documents/GitHub/r9-ardupilot
pick d754913 AP_Param_Helper: HAL_F4Light parameters divided into common and board specific
pick 1224ddc AP_HAL_F4Light: fixed some support scripts
s ba0cec9 AP_HAL_F4Light: small fix (NFC)
pick b371d24 AP_HAL_F4Light: more comments translated, added support to reboot into DFU mode even in bootloader version
pick a96378e AP_HAL_F4Light: removed some commented-out code
pick 99ed57f AP_HAL_F4Light: added readme to USB driver
r 86e2e82 Tools: fixed bootloader binary - revo405_b1
pick 8ae4047 AC_Avoidance: NFC small renames and comment improvements
pick 35a4748 Copter: follow mode renames and comment improvements
pick df63268 Tools: add name to Git_Success.txt

# Rebase 5c0c3a0..df63268 onto 5c0c3a0 (10 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
-
~/Documents/GitHub/r9-ardupilot/.git/rebase-merge/git-rebase-todo[+] [unix] (14:19 06/03/2018) 7,2 All

```

Cuando confirmamos los cambios y salgamos, el rebase va a continuar y va a ir aplicando los comandos uno por uno, parando entre cada uno revisando que no se generen conflictos, en cuyo caso todo el rebase se va a pausar, dándonos la oportunidad de solucionar lo que haya salido mal y continuar con el proceso.

REFLOG

El log de referencias o **reflog** es un lugar en donde quedan guardadas las puntas de branches y otras referencias en nuestro repositorio local. El reflog cubre todas las acciones recientes como cambios de branch, resets, commits, etc.

A medida que vamos trabajando, GIT va guardando cada movimiento que realiza nuestro HEAD. Cada vez que realizamos una acción, el reflog es actualizado. Siguiendo con el ejemplo anterior del archivo **server.js**, así se encuentra nuestro reflog ahora mismo :

```

> git reflog
8d86da1 (HEAD -> master) HEAD@{0}: commit: Add copy change
7737d6b HEAD@{1}: commit: add new change
297762a (origin/master, origin/HEAD) HEAD@{2}: clone: from
https://github.com/github/platform-samples

```

Como podemos observar, de momento es un simple listado de los últimos dos commits que se realizaron, lo cual también podemos obtener con un simple **git log**, pero si prestamos atención a la primera línea, la misma nos muestra el momento en el que hicimos el clon de nuestro repositorio en la máquina local.

RESTAURANDO TRABAJO

Vamos a proceder a realizar un experimento. Imaginemos que sin querer hacemos un **git reset --hard** sobre nuestros archivos, lo cual pondría nuestro HEAD en un estado anterior de trabajo y haría que perdamos los commits en donde estaba el trabajo realizado. Utilizando el **reflog**, podríamos entonces enterarnos el momento en el que se hizo ese cambio :

```
> git reset --hard HEAD~2
HEAD is now at 297762a Merge pull request #326 from gamventures/master

> git log --oneline -4
297762a (HEAD -> master, origin/master, origin/HEAD) Merge pull request #326 from
gamventures/master
2065097 Update Readme. file
4956d3f Merge pull request #321 from jdweiner526/jdweiner526-docfix-1
fe91e2f Merge pull request #323 from github/hollywood/update-username-text

> git reflog
297762a (HEAD -> master, origin/master, origin/HEAD) HEAD@{0}: reset: moving to HEAD~2
8d86da1 HEAD@{1}: commit: Add copy change
7737d6b HEAD@{2}: commit: add new change
297762a (HEAD -> master, origin/master, origin/HEAD) HEAD@{3}: clone: from
https://github.com/github/platform-samples
```

De esta manera no solo podemos ver los cambios por los que fue atravesando el HEAD sino que también podemos utilizar el comando **git checkout** para movernos a cualquier momento del pasado de dicho HEAD :

```
> git checkout HEAD@{1}
Note: switching to 'HEAD@{1}'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in
this
```

```
state without impacting any branches by switching back to a branch.
```

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -c with the switch command. Example:

```
git switch -c <new-branch-name>
```

Or undo this operation with:

```
git switch -
```

Turn off this advice by setting config variable advice.detachedHead to false

```
HEAD is now at 8d86da1 Add copy change
```

Nótese que estamos utilizando una nueva forma de referencia para movernos a través del repositorio, distinta a la que veníamos usando en clases anteriores como el hash de los commits o el nombre de los branches. En este caso estamos usando la referencia **HEAD@{1}** la cual se muestra en el listado de nuestro reflog. Así nos estamos moviendo a un momento en el pasado, a través del cual nuestro HEAD transitó.

Tengamos en cuenta que vamos a estar parados en un commit que aún no fue borrado por nuestro repositorio local pero el cual no pertenece a ninguna parte de nuestro árbol real de commits, con lo cual nos sale el mensaje de **detached HEAD** y en nuestro log no vamos a ver ningún branch :

```
> git log --oneline -4
8d86da1 (HEAD) Add copy change
7737d6b add new change
297762a (origin/master, origin/HEAD, master) Merge pull request #326 from
gamventures/master
2065097 Update Readme. file
```

Para solucionar este inconveniente simplemente creamos un branch en donde estemos parados en ese momento y de esa manera el trabajo no solo se restaura sino que esa porción de árbol vuelve a pertenecer a nuestro historial :

```
> git checkout -b restauracion
Switched to a new branch 'restauracion'
```

```
> git log --oneline -4
8d86da1 (HEAD -> restauracion) Add copy change
7737d6b add new change
297762a (origin/master, origin/HEAD, master) Merge pull request #326 from
gamventures/master
2065097 Update Readme. file
```

FLUJOS DE TRABAJO

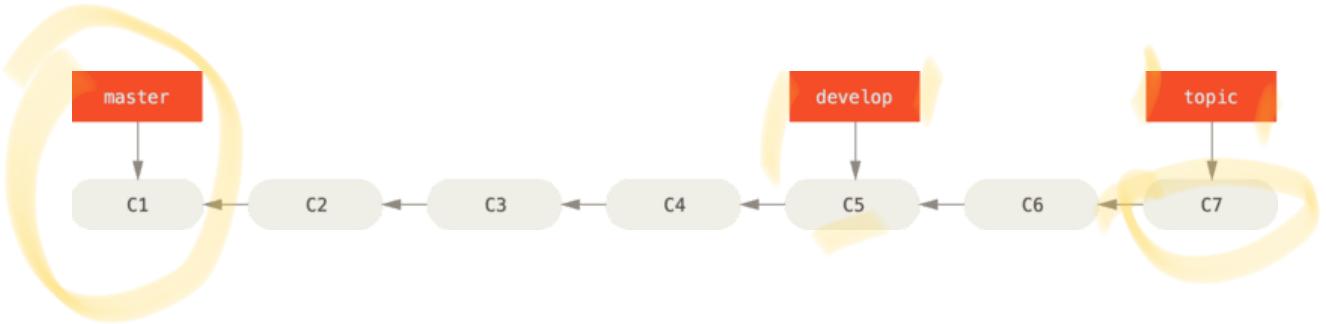
A medida que avancemos en el uso de GIT y le demos mayor presencia en cada uno de nuestros proyectos, va a surgir la necesidad de tener flujos de trabajo ya sea que estemos trabajando solos o en equipo con distintas personas; especialmente en este último caso nos sirve para unificar conceptos y poder mitigar posibles errores de integración que pasarían si cada desarrollador realiza cambios de la manera que quiere en nuestro código. Vamos a presentar a continuación algunos posibles casos comunes de flujos de trabajo.

LONG-RUNNING BRANCH

Debido a que Git utiliza un three-way merge, generalmente es fácil hacerlo de un branch a otro varias veces durante un período prolongado. Esto significa que podemos tener varios branches que siempre estén abiertos y que utilizamos para diferentes etapas de su ciclo de desarrollo; podemos fusionarlas regularmente unas con otras.

Muchos desarrolladores tienen un flujo de trabajo que abarca este enfoque, como tener solo código que sea completamente estable en su rama master, posiblemente solo código que ha sido o será lanzado. Tienen otra rama paralela llamada desarrollo (dev) desde la que trabajan o usan para probar la estabilidad: no siempre es estable, pero siempre que llega a un estado estable, se puede fusionar en master. Se utiliza para atraer ramas temáticas (ramas de corta duración) cuando están listas, para asegurarse de que pasan todas las pruebas y no introducen errores.

En realidad, estamos hablando de punteros que avanzan en la línea de commits. Los branches estables están más abajo en la línea de tu historial de commits, y los branches más avanzadas están más arriba en el historial.

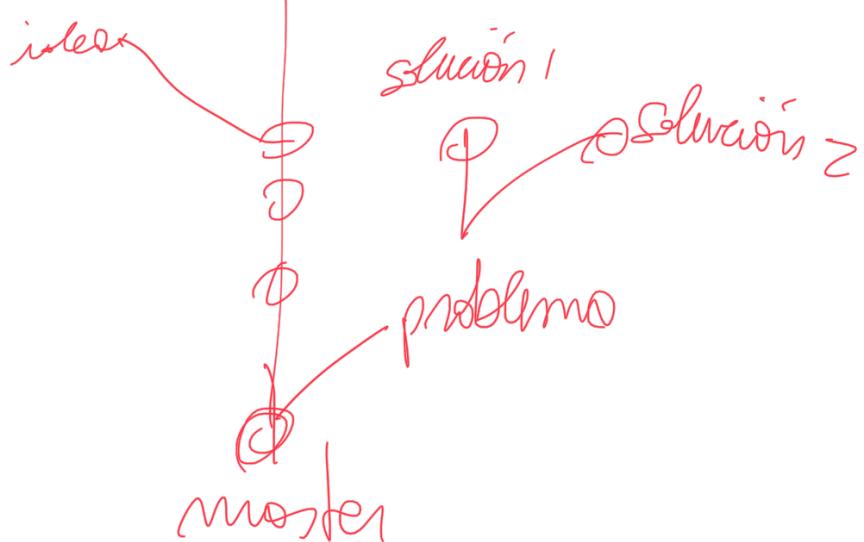


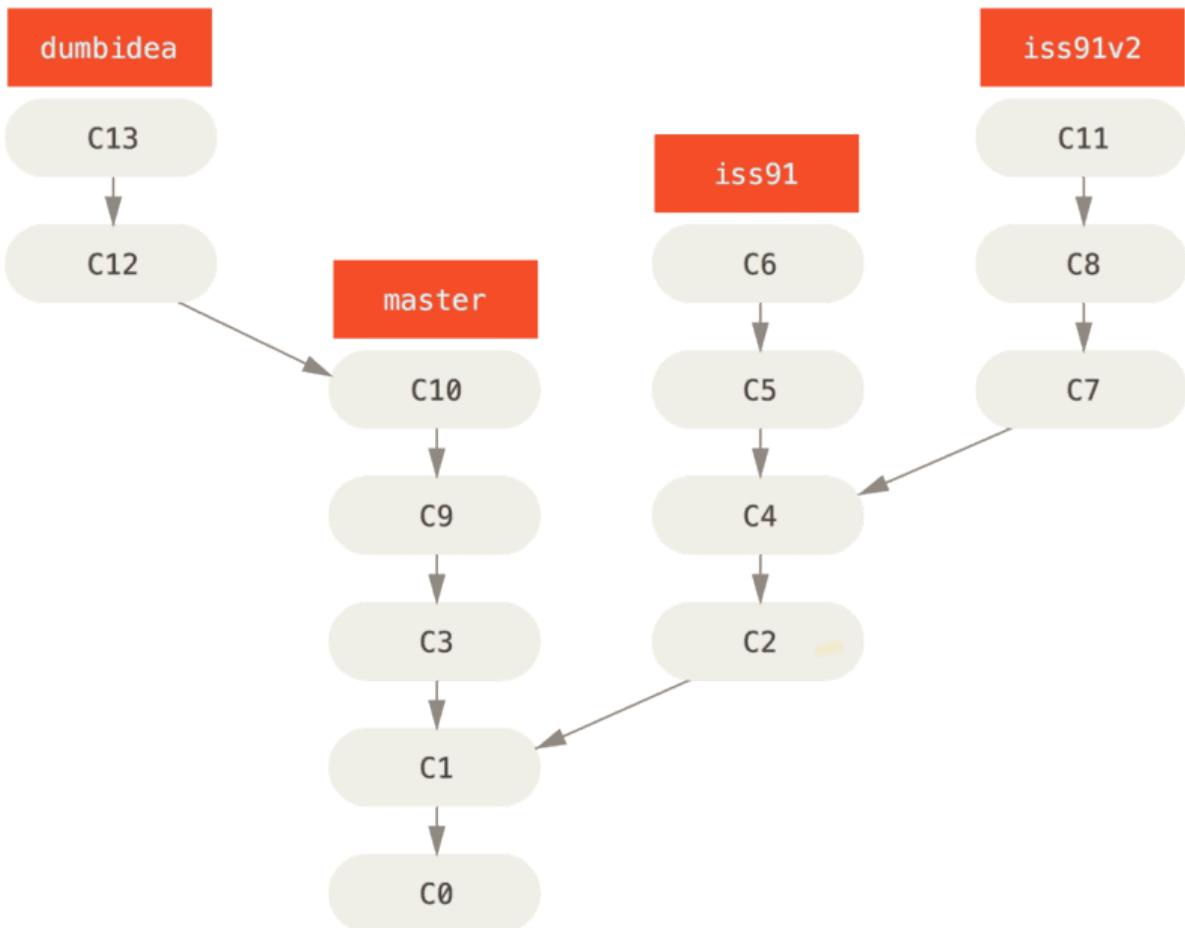
TOPIC BRANCH

Los branches temáticos son útiles en proyectos de cualquier tamaño. Un branch temático es una rama de corta duración que se crea y usa para una característica particular o trabajo relacionado. Esto es algo que probablemente nunca haya hecho antes con un VCS porque generalmente es demasiado costoso crear y fusionar. Pero en Git es común crear, trabajar, fusionar y eliminar branches varias veces al día.

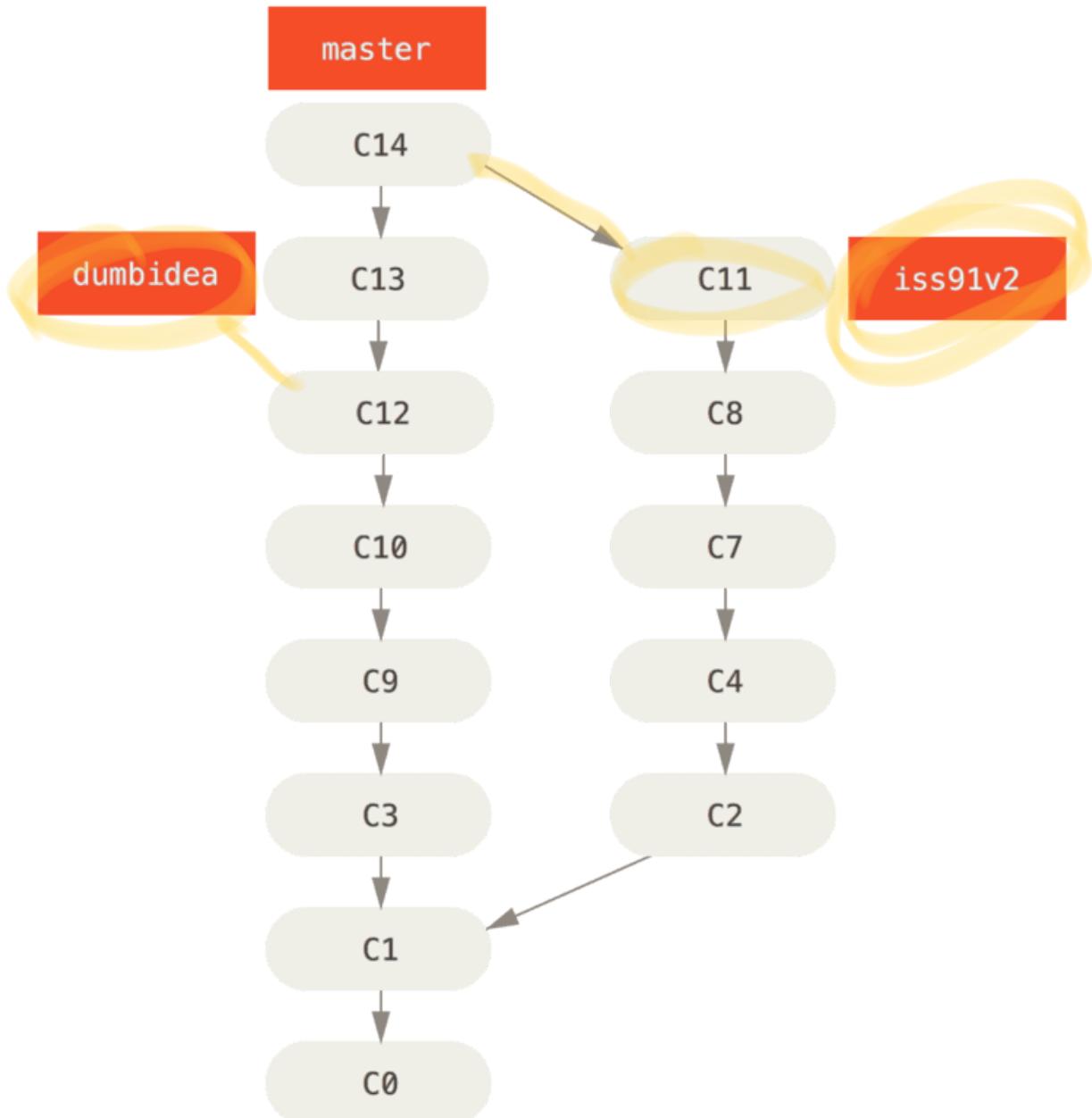
Esta técnica le permite cambiar de contexto rápida y completamente, ya que su trabajo está separado en silos donde todos los cambios en esa rama tienen que ver con ese tema, es más fácil ver lo que sucedió durante la revisión del código y tal. Puede guardar los cambios allí durante minutos, días o meses y combinarlos cuando estén listos, independientemente del orden en que se crearon o trabajaron.

Considere un ejemplo de hacer algo de trabajo (en master), ramificarse por un problema ([iss91](#)), trabajar un poco, ramificarse de la segunda rama para intentar otra forma de manejar lo mismo ([iss91v2](#)), volver a su master branch y trabajar allí por un tiempo, y luego ramificarse allí para hacer un trabajo que no estás seguro es una buena idea ([dumbidea branch](#)). Su historial de commits se verá más o menos así:





Ahora, supongamos que decide que le gusta más la segunda solución a su problema (**iss91v2**); y le mostraste el branch **dumbidea** a tus compañeros de trabajo, y resulta ser genial. Puede borrar la rama **iss91** original (perdiendo commits **C5** y **C6**) y hacer merge de las otras dos. Su historial se verá así:

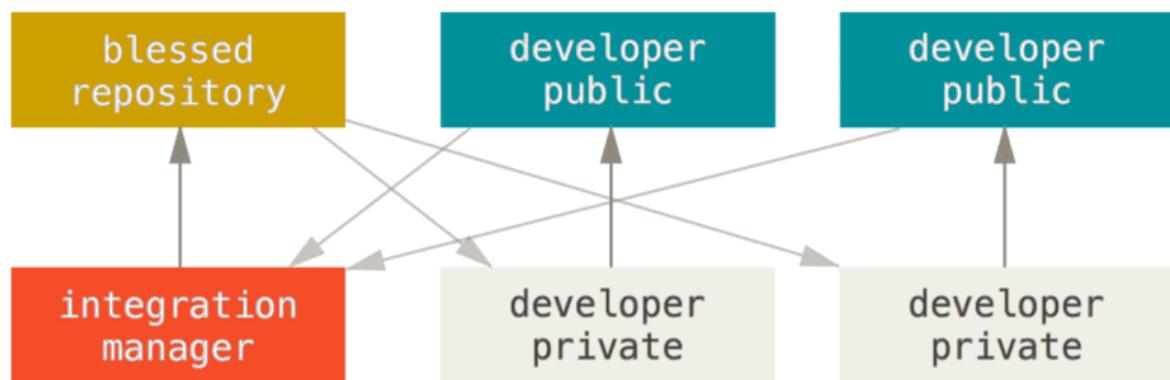


ADMINISTRADOR DE INTEGRACIÓN

Debido a que Git nos permite tener múltiples repositorios remotos, es posible tener un flujo de trabajo donde cada desarrollador tenga acceso de escritura a su propio repositorio público y acceso de lectura a todos los demás. Este escenario a menudo incluye un repositorio canónico que representa el proyecto "oficial". Para contribuir a ese proyecto, debe crear su propio clon público del proyecto e impulsar sus cambios. Luego, puede enviar una solicitud al responsable del proyecto principal para que realice sus cambios. El responsable de mantenimiento puede agregar su repositorio como un control remoto,

probar sus cambios localmente, fusionarlos en su rama y volver a su repositorio. El proceso funciona de la siguiente manera (consulte Flujo de trabajo del administrador de integración):

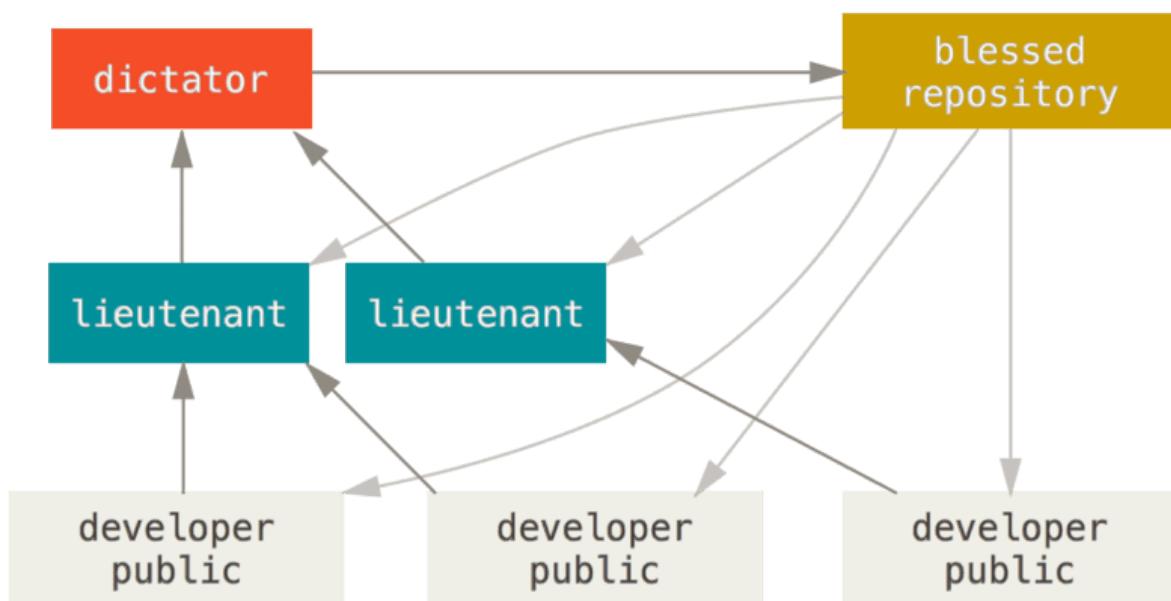
1. El responsable del proyecto hace push a su repositorio público.
2. Un contribuidor clona ese repositorio y realiza cambios.
3. El contribuyente hace push a su propia copia pública.
4. El contribuyente envía un correo electrónico al responsable de mantenimiento pidiéndole que haga pull de los cambios.
5. El mantenedor agrega el repositorio del contribuyente como remoto y se fusiona localmente.
6. El mantenedor empuja los cambios combinados al repositorio principal.



DICTADOR Y TENIENTES

Esta es una variante de un flujo de trabajo de repositorio múltiple. Generalmente es utilizado por grandes proyectos con cientos de colaboradores; Un ejemplo famoso es el kernel de Linux. Varios gerentes de integración están a cargo de ciertas partes del repositorio; se llaman tenientes. Todos los tenientes tienen un administrador de integración conocido como el dictador benevolente. El dictador benevolente hace push desde su directorio a un repositorio de referencia del que todos los colaboradores deben hacer pull. El proceso funciona así :

1. Los desarrolladores trabajan en su rama temática y hacen rebase de su trabajo por encima de master. La rama master es la del repositorio de referencia al que hace push el dictador.
2. Los tenientes fusionan las ramas temáticas de los desarrolladores en su rama master.
3. El dictador fusiona las ramas maestras de los tenientes en la rama master del dictador.
4. Finalmente, el dictador hace push de esa rama master al repositorio de referencia para que los otros desarrolladores puedan reajustarse en ella.



GITFLOW

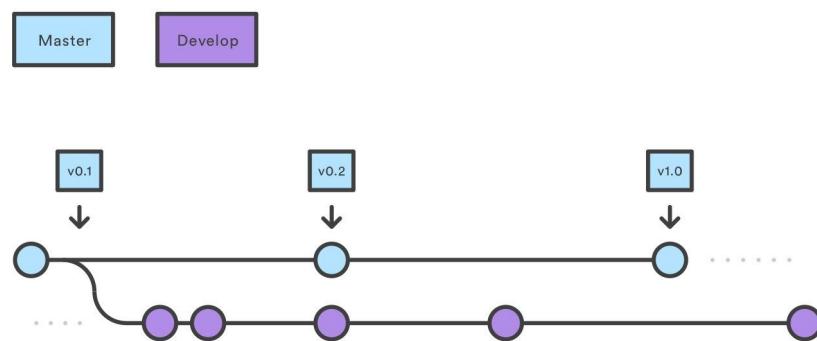
Gitflow Workflow es un diseño de flujo de trabajo de GIT que se publicó por primera vez y se hizo popular por Vincent Driessen en nvie. El flujo de trabajo de Gitflow define un modelo de ramificación estricto diseñado en torno al lanzamiento del proyecto. Esto proporciona un marco robusto para gestionar proyectos más grandes.

Gitflow es ideal para proyectos que tienen un ciclo de lanzamiento programado. Este flujo de trabajo no agrega nuevos conceptos o comandos más allá de lo que se requiere para el Flujo de trabajo de la rama de funcionalidades. En cambio, asigna roles muy específicos a

diferentes branches y define cómo y cuándo deben interactuar. Además de los branches de características, utiliza branches individuales para preparar, mantener y grabar lanzamientos.

Por supuesto, también puede aprovechar todos los beneficios del flujo de trabajo de branches de funcionalidades : pull requests, experimentos aislados y una colaboración más eficiente.

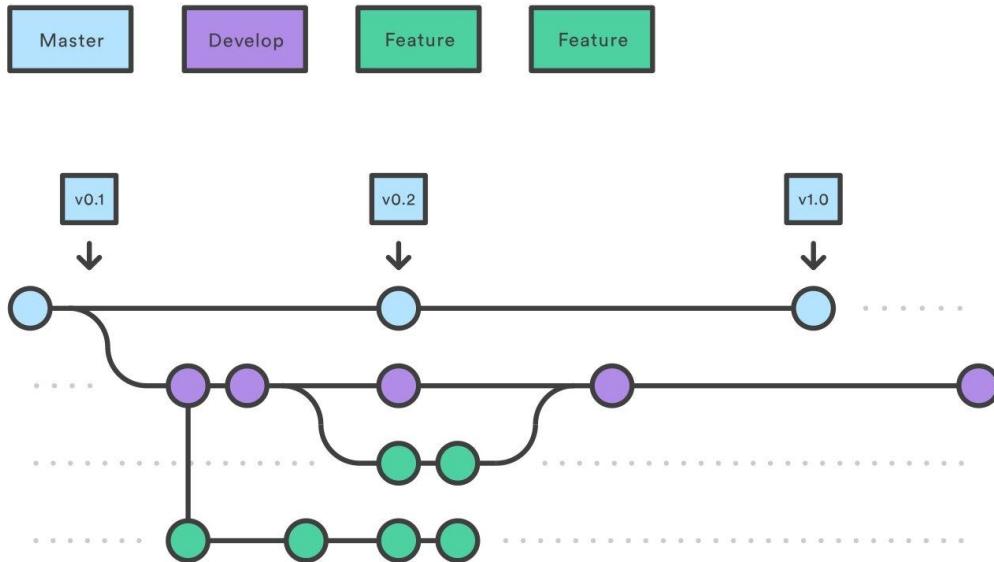
En lugar de una sola rama maestra, este flujo de trabajo usa dos ramas para registrar el historial del proyecto. La rama maestra almacena el historial de lanzamiento oficial, y la rama de desarrollo sirve como una rama de integración de características. También es conveniente etiquetar todos los commits en la rama maestra con un número de versión.



El primer paso es complementar el maestro predeterminado con una rama de desarrollo. Una manera simple de hacer esto es que un desarrollador cree una rama de desarrollo vacía localmente y la envíe al servidor :

```
> git branch develop  
> git push -u origin develop
```

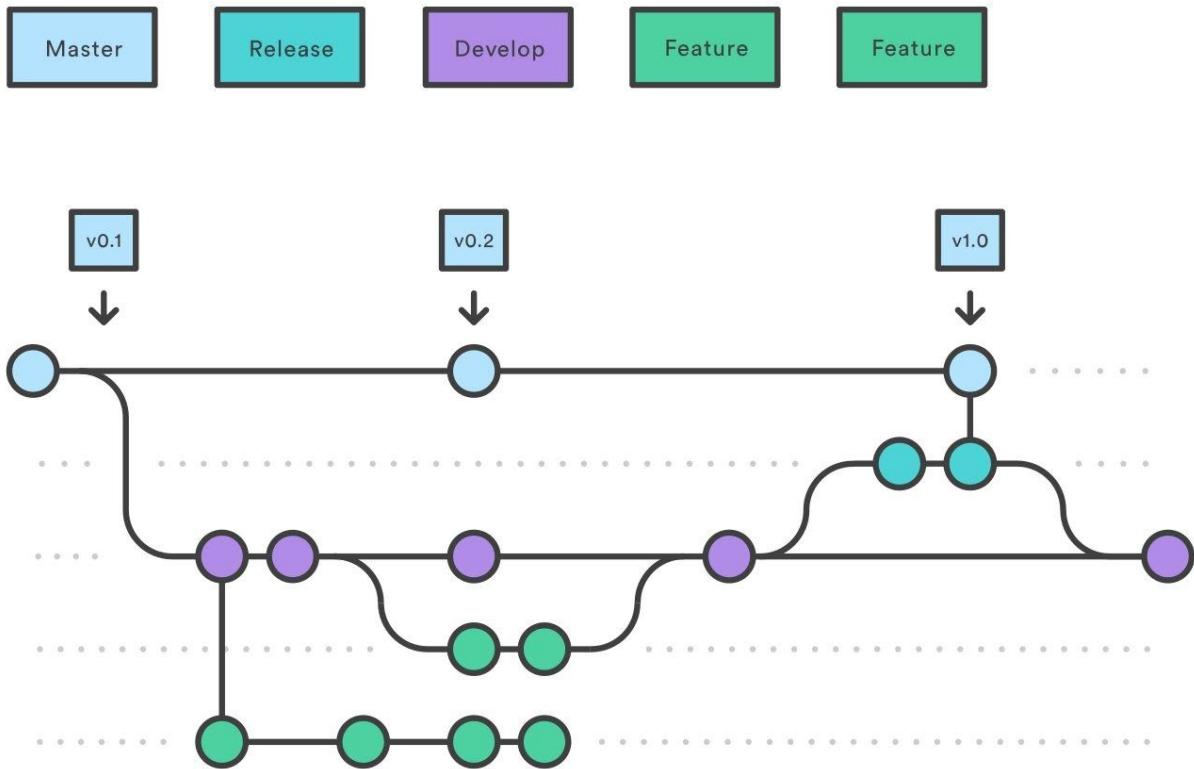
Cada nueva característica debe residir en su propio branch, que se puede enviar al repositorio central para respaldo / colaboración. Pero, en lugar de bifurcarse del maestro, las ramas de características usan la de desarrollo como su rama principal. Cuando se completa una característica, se fusiona nuevamente en el desarrollo. Las características nunca deberían interactuar directamente con el maestro.



Una vez que el desarrollo ha adquirido suficientes características para un lanzamiento (o se acerca una fecha de lanzamiento predeterminada), podemos bifurcar una rama de lanzamiento fuera del desarrollo. La creación de esta rama inicia el siguiente ciclo de lanzamiento, por lo que no se pueden agregar nuevas características después de este punto, solo las correcciones de errores, la generación de documentación y otras tareas orientadas a la versión deben ir en esta rama. Una vez que está listo para enviar, la rama de lanzamiento se fusiona en master y se etiqueta con un número de versión. Además, debe fusionarse nuevamente en el desarrollo, que puede haber progresado desde que se inició el lanzamiento.

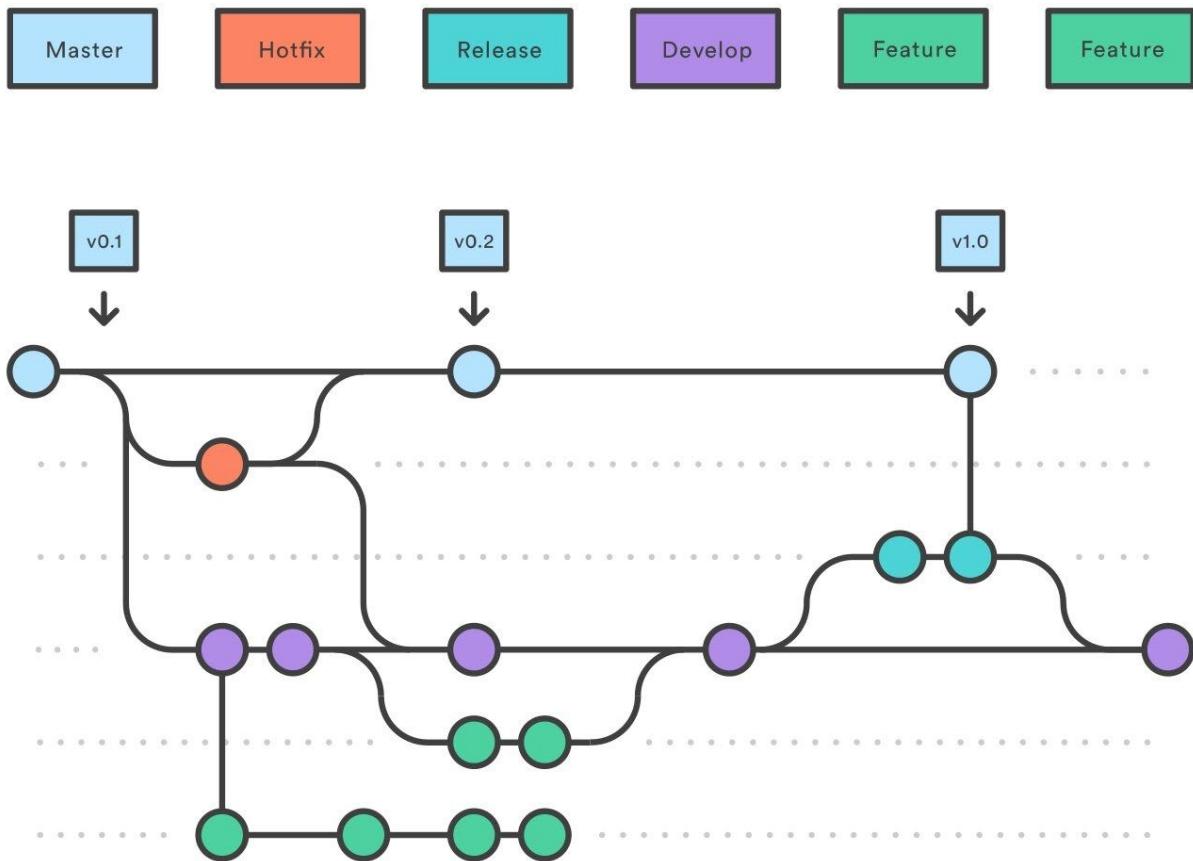
El uso de una rama dedicada para preparar versiones hace posible que un equipo pueda pulir la versión actual mientras que otro equipo continúa trabajando en las características para la próxima versión. También crea fases de desarrollo bien definidas (por ejemplo, es fácil decir: "Esta semana nos estamos preparando para la versión 4.0" y verlo realmente en la estructura del repositorio).

Hacer ramas de lanzamiento es otra operación de ramificación sencilla. Al igual que las ramas de características, las ramas de lanzamiento se basan en la rama de desarrollo.



Las ramas de mantenimiento, revisión o "**hotfix**" se utilizan para parchear rápidamente las versiones de producción. Las ramificaciones de revisión son muy parecidas a las ramificaciones de lanzamiento y ramificaciones de características, excepto que se basan en master en lugar de desarrollo. Esta es la única rama que debe bifurcarse directamente del maestro. Tan pronto como se complete la corrección, debe fusionarse tanto en el maestro como en desarrollo (o en la rama de la versión actual), y el maestro debe etiquetarse con un número de versión actualizado.

Tener una línea de desarrollo dedicada para la corrección de errores le permite a su equipo abordar problemas sin interrumpir el resto del flujo de trabajo o esperar el próximo ciclo de lanzamiento. Puede pensar en las ramas de mantenimiento como ramas de lanzamiento ad hoc que funcionan directamente con el maestro.



Tengamos en cuenta que si bien podemos realizar todo este trabajo de manera manual, es decir, ponernos de acuerdo con nuestro equipo de trabajo para que todos adopten el mismo tipo de flujo de trabajo, podrían presentarse errores hasta que el sistema de bifurcación esté bien pulido y practicado por todos.

Para resolver este inconveniente se pueden encontrar varios clientes gráficos de GIT que ya incorporan el sistema de Gitflow para comenzar o continuar con proyectos existentes o bien existen determinadas extensiones para la línea de comandos que realizan el mismo tipo de trabajo por nosotros sin que corramos ningún problema a la hora de realizar cada tipo de branch.

CONFIGURACIÓN AVANZADA

Antes de terminar con el curso, notemos parámetros de configuración adicionales en GIT que podrían llegar a hacernos el trabajo un poco más fácil.

ALIAS

Una de las capacidades más utilizadas por los usuarios de GIT en la línea de comandos es la capacidad de crear alias con el comando :

```
> git alias
```

El mismo nos permite crear atajos en los momentos que no queremos escribir un comando entero porque por ejemplo lo utilizamos constantemente, como por ejemplo **git log --oneline --all --graph** podemos ahora guardarlo en un alias de la siguiente manera :

```
git config --global alias.lo "log --oneline --graph --all"
```

De esta manera, la próxima vez que tengamos que ejecutar ese comando podemos simplemente hacer lo siguiente :

```
git lo

* 8d86da1 (HEAD -> restauracion) Add copy change
* 7737d6b add new change
| * 93aaac9 (origin/dependabot/bundler/) Bump activesupport in rendering-data-as-graphs
|/
* 297762a (origin/master, origin/HEAD, master) Merge pull request
| \
```

PLANTILLA DE COMMITS

Podemos utilizar también una configuración a veces poco conocida la cual nos permite agregar plantillas personalizadas a los mensajes del comando **commit** utilizando el siguiente parámetro de configuración :

```
> git config --global commit.template ~/.mensajecommit.txt
```

De esta manera le estamos diciendo a GIT que cada vez que se ejecute el comando **git commit**, muestre además el texto que está guardado en dicho archivo.

Esto es muy beneficioso ya que podemos unificar maneras de trabajo sin tener que constantemente acordarnos de ellas; por ejemplo, si queremos que en cada commit se esté dejando determinado mensaje en tal o cual formato, lo cual nos sirve no solamente a nosotros sino también especialmente si estamos trabajando con un equipo de personas.

EXPIRACIÓN DEL REFLG

Tenemos que tener en cuenta que el **reflog** por defecto viene configurado para que tenga una duración máxima de 30 días para casos **unreachable** y 90 días para casos normales. Podemos en tal caso extenderlo o disminuirlo dependiendo nuestra necesidad en cada repositorio con el siguiente parámetro de configuración :

```
> git config gc.reflogExpire  
> git config gc.reflogExpireUnreachable
```

Tengamos en cuenta que la diferencia entre casos normales y los **unreachable** es básicamente la capacidad que tenemos para acceder a referencias desde nuestro árbol. Si tenemos en cuenta el siguiente ejemplo :

```
1234567 <-- master  
 /  
 ...--9999999--8888888 [master@{1}]  
 \\\n    \ 3333333 [master@{2}]
```

Podemos observar que si estuviéramos parados, por ejemplo, en el branch master y fuéramos para atrás en relación a nuestro log nos toparemos con el commit **888888...** , o en otras palabras **master@{1}**. Este commit es accesible de momento, el cual va a estar disponible en el reflog hasta 90 días. Si fuéramos un paso más atrás en el historial, nos vamos a topar con el commit **9999999...** y así sucesivamente.

Si prestamos atención al commit **333333...** , el mismo no se encuentra disponible desde nuestro árbol directo , con lo cual es **unreachable** y el mismo permanecerá disponible por un plazo de 30 días.