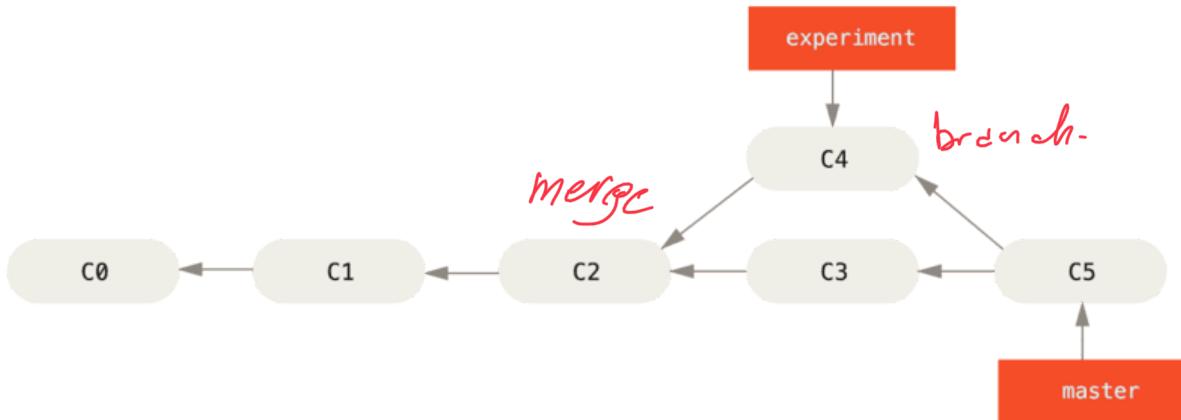


GIT

CLASE 3 : FLUJOS DE TRABAJO

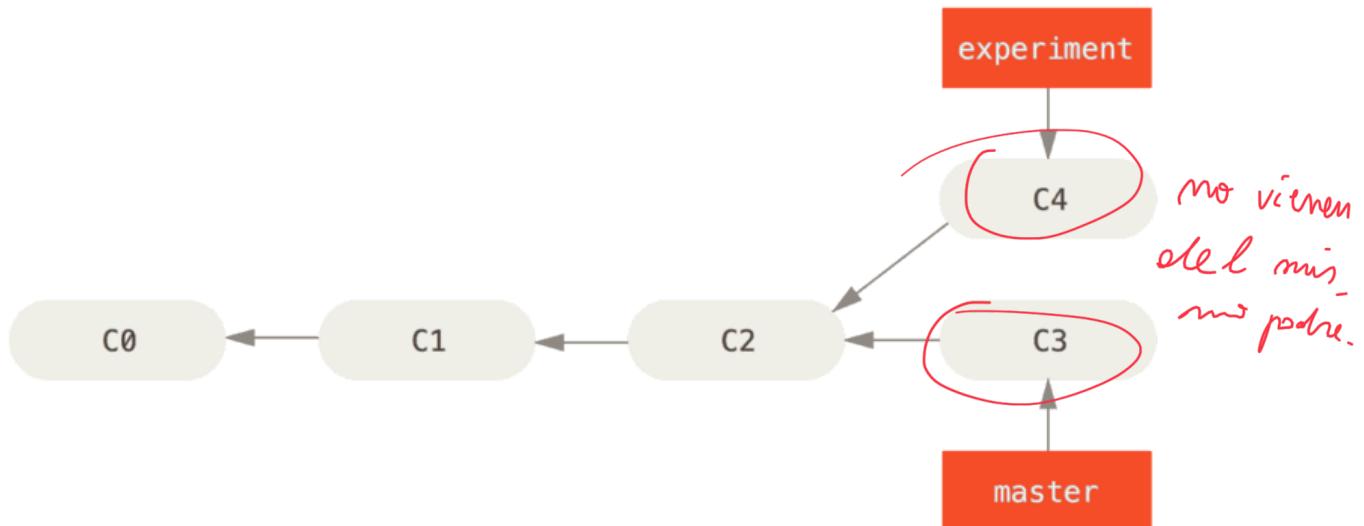
ESTRATEGIAS DE UNIFICACIÓN : REUBICACIÓN

Como vimos en clases anteriores, en **GIT** existen dos maneras de integrar cambios de un branch en otro : *merge* y *rebase*. La manera más sencilla y popular de fusionar nuestro trabajo nuevo con trabajo ya existente dentro de nuestro proyecto es la de realizar un *merge*, el cual realizará una fusión a tres bandas entre los dos últimos snapshots de cada rama que queremos integrar, en el caso en donde tengamos ramas bifurcadas y su ancestro común; de esta manera **GIT** nos va a crear un nuevo snapshot y su correspondiente commit.



EL COMANDO REBASE

Imaginemos por un momento qué pasaría si pudiéramos ordenar nuestros branches bifurcados de forma tal que en vez de separar su historial desde un parente en común, pudieran formar una estructura de grafo lineal, es decir, como si nunca se hubieran bifurcado. Intentemos imaginar dicho escenario en el siguiente historial :



En este caso, podríamos reubicar los cambios introducidos en el commit **C4** y aplicarlos encima del commit **C3**. Como comentamos anteriormente, si este fuera el caso, en vez de tener **C3** y **C4** en caminos distintos, tendríamos a **C4** por delante de **C3** dandonos el historial **C0 - C1 - C2 - C3 - C4** y de esta manera podemos volver a tener una estructura lineal de trabajo, lo cual nos permite interpretar mejor y más rápido nuestro historial.

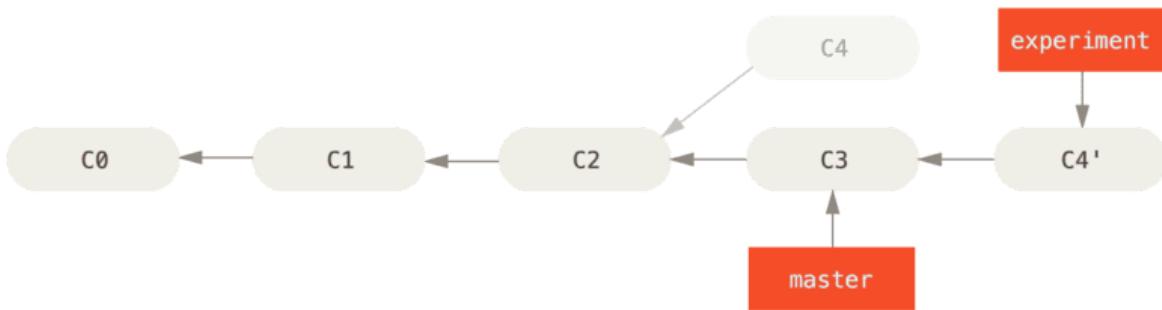
Para realizar esta operación podemos utilizar el comando **git rebase**, el cual nos permite llevarnos los cambios confirmados en un branch y aplicarlos sobre otra en vez de fusionar a tres bandas.

Para esto vamos dirigirnos a aquel branch que se había bifurcado y que queremos tener de manera lineal y luego correr el siguiente comando :

```
> git rebase <branch>
```

De esta manera le estamos diciendo después de que rama queremos ver sus cambios :

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



Haciendo que **GIT** vaya al ancestro común de ambas ramas (donde estás actualmente y de donde quieras 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 quieras reorganizar, y, finalmente, vuelva a aplicar ordenadamente los cambios.

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

MERGE VS REBASE : CONTRAS Y BENEFICIOS

La dicha del rebase no la alcanzamos sin sus contrapartidas, las cuales pueden resumirse en una línea :

Nunca reorganices confirmaciones de cambio (commits) que hayas enviado (push) a un repositorio público.

Si sigues esta recomendación, no tendrás problemas. Pero si no lo haces podrás causar problemas en el repositorio, generando conflictos en el resto de tu grupo de trabajo.

Cuando le hacemos rebase a algo, estamos abandonando las confirmaciones de cambio ya creadas y estás creando unas nuevas; que son similares, pero diferentes. Si envías (push) confirmaciones (commits) a alguna parte, y otros las recogen (pull) de allí; y después vas tú y las reescribes con git rebase y las vuelves a enviar (push); tus colaboradores tendrán que re fusionar (re-merge) su trabajo y todo se volverá tremadamente complicado cuando intentes recoger (pull) su trabajo de vuelta sobre el tuyo.

Para algunos, el historial de confirmaciones de tu repositorio es un registro de todo lo que ha pasado. Un documento histórico, valioso por sí mismo y que no debería ser alterado. Desde este punto de vista, cambiar el historial de confirmaciones es casi como blasfemar; estarías mintiendo sobre lo que en verdad ocurrió. ¿Y qué pasa si hay una serie desastrosa

de fusiones confirmadas? Nada. Así fue como ocurrió y el repositorio debería tener un registro de esto para la posteridad.

La otra forma de verlo es que el historial de confirmaciones es la historia de cómo se hizo tu proyecto. Tú no publicarías el primer borrador de tu novela, y el manual de cómo mantener tus programas también debe estar editado con mucho cuidado. Esta es el área que utiliza herramientas como rebase y filter-branch para contar la historia de la mejor manera para los futuros lectores.

Ahora, sobre qué es mejor si *merge* o *rebase*: verás que la respuesta no es tan sencilla. **GIT** es una herramienta poderosa que te permite hacer muchas cosas con tu historial, y cada equipo y cada proyecto es diferente. Ahora que conoces cómo trabajan ambas herramientas, será cosa tuya decidir cuál de las dos es mejor para tu situación en particular.

Normalmente, la manera de sacar lo mejor de ambas es reorganizar tu trabajo local, que aún no has compartido, antes de enviarlo a algún lugar; pero nunca reorganizar nada que ya haya sido compartido.

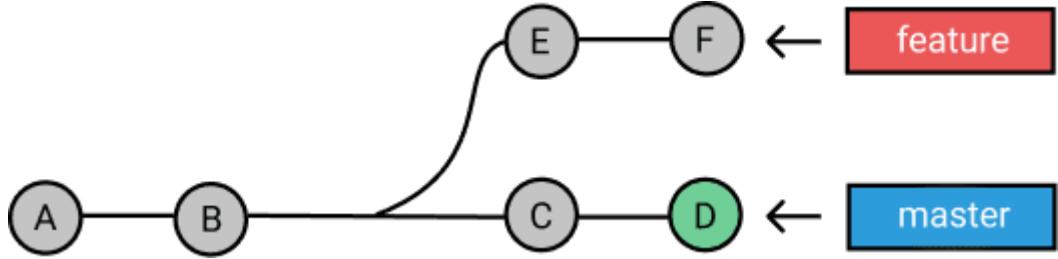
ESTRATEGIAS DE UNIFICACIÓN : CAMBIOS ESPECÍFICOS.

Varias veces los equipos de trabajo pueden tener miembros individuales trabajando sobre el mismo código. Tal vez una nueva funcionalidad de algún producto tiene un componente del front-end y del back-end, con lo cual hay código compartido entre los dos sectores de un mismo producto.

Dado este escenario, los desarrolladores back-end pueden llegar a encontrar y/o desarrollar estructuras que también van a ser utilizadas en el front-end. En este caso los desarrolladores front-end pueden usar el comando *git cherry-pick* para elegir commits en los cuales esta solución hipotética fue creada, de esta manera el desarrollador front-end puede continuar progresando en su lado del proyecto sin afectar el branch del lado back-end.

EL COMANDO CHERRY-PICK

Imaginemos que estamos trabajando sobre dos branches distintos : *master* y *feature* - y que queremos usar el commit **C** del branch *master* dentro de nuestro branch *feature* :

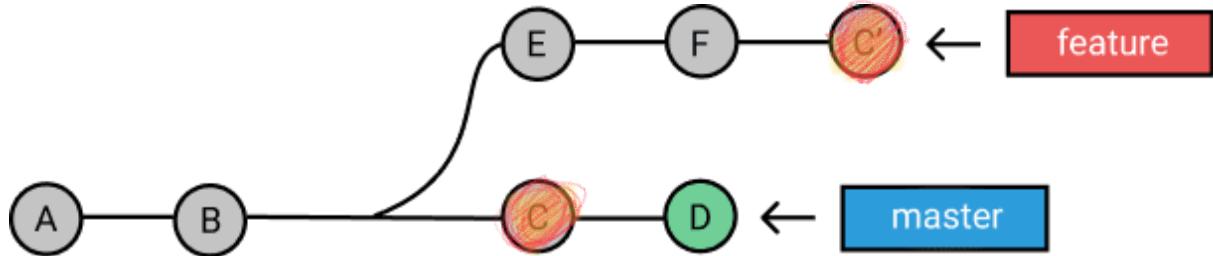


Tenemos que considerar varios pasos a seguir para realizar este trabajo :

1. Obtener el hash del commit. Podemos hacer esto de, por lo menos, dos maneras:
 - Utilizando el comando `git log --oneline`, para obtener todo el historial de commits. Tenemos que asegurarnos de estar en el branch correcto, en este caso *master*, ya que queremos usar un commit de ahí.
 - Yendo a GitHub y seleccionando el hash del commit desde ahí.
2. Hacer un `git checkout` al branch en el cual queremos aplicar dicho commit; en este caso : `git checkout feature`. *switch*.
3. Utilizar el comando `git cherry-pick` :

```
> git cherry-pick C
```

Si corremos luego el comando `git log` podemos ver que hemos aplicado un commit nuevo, pero con los cambios viejos del branch *master* a lo último de nuestro branch *feature* :



Hay que tener en cuenta que el commit nuevo agregado por encima del branch *feature* va a tener un hash distinto al hash que originalmente tenía en el branch *master*, pero los cambios van a ser los mismos en ambos lados del proyecto.

TRABAJO TEMPORAL

Usualmente cuando estamos trabajando en partes de nuestro proyecto, queremos cambiar de branch por varias razones momentáneamente para resolver otro tipo de problemas que se han generado en otras áreas del repositorio. El problema es que no queremos realizar un commit de trabajo a medio hacer solo para volver y completarlo más tarde, pero necesitamos nuestro espacio de trabajo limpio para poder movernos entre branches.

QUÉ ES EL STASH ?

El stash toma nuestro estado “sucio” de trabajo del Working Directory, es decir, las modificaciones de archivos en seguimiento y cambios ya listos que forman parte del Stage Area , y los guarda en una pila de cambios sin terminar, los cuales podemos aplicar luego en cualquier momento de nuestro desarrollo sobre el mismo branch o bien, uno distinto.

CREANDO NUESTRO PRIMER STASH

Para demostrar lo que es el stash, imaginemos que estamos trabajando en un proyecto y que comenzamos a trabajar en varios archivos y posiblemente preparando cambios sobre los mismos. Si ejecutamos el comando `git status` podríamos ver un estado “sucio” de trabajo:

```
> git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   lib/simplegit.rb
```

Ahora necesitamos cambiar de branch, pero no queremos generar un commit con lo que venimos trabajando hasta el momento, entonces podemos guardar los cambios de manera temporal con el comando `git stash` ó `git stash push` :

```
> git stash  
Saved working directory and index state \  
"WIP on master: 049d078 added the index file"  
HEAD is now at 049d078 added the index file  
(To restore them type "git stash apply")
```

Ahora podemos ver que nuestro Working Directory está "limpio" :

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

En este punto podemos tranquilamente cambiarnos de branch sin problemas dado que los cambios temporales quedaron almacenados en nuestra pila de cambios temporales.
Podemos incluso ver qué cambios hemos guardado usando el comando `git stash list` :

```
> git stash list  
stash@{0}: WIP on master: 049d078 added the index file ✓  
stash@{1}: WIP on master: c264051 Revert "added file_size" ✓  
stash@{2}: WIP on master: 21d80a5 added number to log ✓
```

RESTAURANDO TRABAJO DESDE EL STASH

Podemos volver a aplicar ahora cualquier modificación temporal que hubiéramos guardado en el stash usando el comando `git stash apply`:

```
> git stash apply  
# On branch master  
# Changes not staged for commit:  
#   (use "git add <file>..." to update what will be committed)  
#  
#       modified:   index.html  
#       modified:   lib/simplegit.rb  
#
```

Si ejecutamos este comando tal cual el ejemplo anterior, vamos a notar que vamos a estar restaurando únicamente los últimos cambios que guardamos en el stash sin la posibilidad de elegir cual queremos de todos los que habíamos guardado con anterioridad. Si en cambio queremos aplicar los cambios de un stash más viejo, podemos especificarlo usando el nombre de referencia del mismo :

```
> git stash apply stash@{2}
```

DEPURACIÓN A TRAVÉS DEL TIEMPO

Imaginemos ahora que tenemos un repositorio el cual viene funcionando correctamente. Un día nos damos cuenta que por algún motivo nuestro proyecto se encuentra corrupto por el motivo que sea (Ej.: Uno de nuestros programas no pasa algún test), entonces tenemos que realizar una depuración o debugging de todo nuestro código, lo cual nos podría llevar una gran cantidad de tiempo en el caso en que nuestro proyecto sea demasiado grande.

Podemos mejorar nuestra técnica de depuración utilizando la búsqueda binaria de **GIT** llamada *git bisect*.

EL COMANDO BISECT

Este comando nos permite ir revisando nuestro historial de commits, testeando nuestro proyecto en cada uno de los puntos de restauración anteriores revisando e identificando dónde surgió el problema realmente.

Si bien esta tarea es un procedimiento que tranquilamente podemos realizar manualmente realizando un checkout a un commit anterior en nuestro historial, realizar los test correspondientes en nuestro programa, verificar si el mismo presenta dicha inconsistencia e ir avanzando commit por commit hasta identificarlo; pero todo este procedimiento podría llevarnos mucho tiempo y esfuerzo.

Ahí es donde el comando *git bisect* entra en acción. El mismo nos permite “automatizar” este trabajo de ir revisando commits realizando una búsqueda binaria sobre nuestro historial de cambios. Pongámoslo en un modo más gráfico para entender el concepto :

```
$ git log --oneline --graph --all
* b7485ac (HEAD -> master, origin/master, origin/HEAD) Fix header size
* 3def738 Add links to screenshots
* 8d517bb Consistently use bash
* f80c3a9 Try with sh
* 329c3f1 Use shell instead of bash for formatting
* a5f5283 Fix formatting in clone command
* 3f9d8de Edit Readme some more
* 98e7889 Edit Readme
* 1773712 Add repo to readme instructions
* 143e5ed Add repo to package.json
* 093a764 Delete whitespace
* 7407c96 Add section about automating git bisect
* a0007bc Delete whitespace
* 4796c0c Update Readme
* 9dfa43b Add more instructions
* 32d3f53 Delete whitespaces
* 24cfbe8 Update Readme some more
* 645c892 Update Readme
* 10ec17f Add linting to test script
* 16a74f5 Add more comments
* 1697fc8 Explain logic() function
* e3f3e49 Fix wording in Readme
* d3d2319 Delete whitespace
* bc9b1f9 Improve Readme
* d8951c8 Delete new lines
* 115637f Edit Readme
* 3d7a348 Add comment
* 6994316 Delete whitespace
* 6f35270 Add mocha to eslint environment
* 26da05b Fix most lint problems
* 56cf49 Add lint job, lots of failures
* 88f6e15 Move dependencies to devDependencies
* 599f373 Install eslint
* a332e39 Delete whitespace
* 179f12a Edit Readme some more
* 1ddca7e Edit Readme
* a2017ba Add Readme file
* b58217f (tag: allGreen) Implement logic(), all tests passing
* f34dea3 Install mocha, add dummy test
* c35e481 Initial import, no functionality
* d9abc2b Initial commit
```

Vamos a poner como ejemplo el repositorio de la imagen anterior. Como podemos ver, tenemos un historial bastante extenso y actualmente nuestro HEAD se encuentra en el commit **b7485ac** que es también donde está nuestro branch master. El mismo presenta un problema al realizar un test en el programa :

```
$ npx mocha test*.js

  Git bisect demo
    #foo()
      ✓ should not do anything

  Git bisect demo
    #logic()
      ✓ should return the sum when adding zero
      ✓ should return the sum when adding to zero
      ✓ should return the sum
    1) should return the sum for negative numbers
      ✓ should return the sum
      ✓ should return zero
      ✓ should not equal zero

  7 passing (13ms)
  1 failing

  1) Git bisect demo
     #logic()
       should return the sum for negative numbers:
         AssertionError [ERR_ASSERTION]: -20 == 20
         + expected - actual
           -20
           +20
         at Context.<anonymous> (test1.js:16:11)
```

Y queremos revisar dónde se generó el error puntualmente. Vamos a tomar como referencia el commit **b58217f** el cual muestra un tag llamado **allGreen** donde sabíamos de antemano que hasta ese momento puntual del proyecto todo venía funcionando correctamente.

Lo primero que tenemos que hacer es iniciar la operación de bisect de **GIT** con el subcomando **git bisect start**:

```
spame@DESKTOP-97V6LM8 MINGW64 ~/Documents/Proyектs/git-bisect-demo (master)
$ git bisect start
spame@DESKTOP-97V6LM8 MINGW64 ~/Documents/Proyектs/git-bisect-demo (master|BISECTING)
```

Lo primero que vamos a notar es que nuestro shell va a mostrarnos que ahora estamos en modo “**BISECTING**”. Lo próximo que tenemos que hacer es decirle a **GIT** un punto fijo donde nosotros consideramos que el proyecto funcionaba sin problemas y otro donde el mismo ya no esté funcionando; a este punto de la operación da igual cuales dos commits le digamos a **GIT**, de cualquier manera el procedimiento siempre va a ser binario y va a intentar ahorrarnos el mayor tiempo posible.

Para decirle ahora a **GIT** que commits es el que sí funciona y cual es el que no, podemos utilizar los subcomandos **git bisect good** y **git bisect bad**; tenemos mitad del trabajo hecho ya que tenemos ambos dos : el commit donde estamos actualmente nos presenta fallas y tenemos identificado un commit anterior donde el programa funcionaba correctamente(b58217f) :

```
spame@DESKTOP-97V6LM8 MINGW64 ~/Documents/Proyектs/git-bisect-demo (master|BISECTING)
$ git bisect bad
spame@DESKTOP-97V6LM8 MINGW64 ~/Documents/Proyектs/git-bisect-demo (master|BISECTING)
$ git log --oneline --graph --all
* b7485ac (HEAD -> master, origin/master, origin/HEAD, refs/bisect/bad) Fix header size
* 3def738 Add links to screenshots
* 8d517bb Consistently use back
```

Como podemos observar en la imagen anterior, simplemente con ejecutar el comando **git bisect bad**, automáticamente **GIT** genera un marcador temporal que identifica ese commit como “malo”. A continuación vamos a movernos hasta el commit que consideramos que funcionaba correctamente y volvemos a realizar los tests correspondientes :

```
$ git checkout b58217f
Note: switching to 'b58217f'.

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 b58217f Implement logic(), all tests passing

spame@DESKTOP-97V6LM8 MINGW64 ~/Documents/Proyects/git-bisect-demo ((allGreen)|BISECTING)
$ npx mocha test*.js

  Git bisect demo
    #foo()
      ✓ should not do anything

  Git bisect demo
    #logic()
      ✓ should return the sum when adding zero
      ✓ should return the sum when adding to zero
      ✓ should return the sum
      ✓ should return the sum for negative numbers
      ✓ should return the sum
      ✓ should return zero
      ✓ should not equal zero

  8 passing (11ms)
```

Como confirmamos que realmente en este commit si funcionaba todo, vamos a decirle a GIT entonces que este commit es "bueno" utilizando el subcomando `git bisect good`:

```
$ git bisect good
Bisecting: 18 revisions left to test after this (roughly 4 steps)
[16a74f57873ff63043c5b846c2e91eed4e64ceb6] Add more comments

spame@DESKTOP-97V6LM8 MINGW64 ~/Documents/Projects/git-bisect-demo
$ git log --oneline --graph --all
* b7485ac (origin/master, origin/HEAD, master, refs/bisect/bad) Fix
* 3def738 Add links to screenshots
* 8d517bb Consistently use bash
* f80c3a9 Try with sh
* 329c3f1 Use shell instead of bash for formatting
* a5f5283 Fix formatting in clone command
* 3f9d8de Edit Readme some more
* 98e7889 Edit Readme
* 1773712 Add repo to readme instructions
* 143e5ed Add repo to package.json
* 093a764 Delete whitespace
* 7407c96 Add section about automating git bisect
* a0007bc Delete whitespace
* 4796c0c Update Readme
* 9dfa43b Add more instructions
* 32d3f53 Delete whitespaces
* 24cfbe8 Update Readme some more
* 645c892 Update Readme
* 10ec17f Add linting to test script
* 16a74f5 (HEAD) Add more comments
* 1697fc8 Explain logic() function
* e3f3e49 Fix wording in Readme
* d3d2319 Delete whitespace
* bc9b1f9 Improve Readme
* d8951c8 Delete new lines
* 115637f Edit Readme
* 3d7a348 Add comment
* 6994316 Delete whitespace
* 6f35270 Add mocha to eslint environment
* 26da05b Fix most lint problems
* 56cf49 Add lint job, lots of failures
* 88f6e15 Move dependencies to devDependencies
* 599f373 Install eslint
* a332e39 Delete whitespace
* 179f12a Edit Readme some more
* 1ddca7e Edit Readme
* a2017ba Add Readme file
* b58217f (tag: allGreen, refs/bisect/good-b58217f071509f8da9ac298f
* f34dea3 Install mocha, add dummy test
* c35e481 Initial import, no functionality
* d9abc2b Initial commit
```

Si volvemos a revisar nuestro log podemos observar que **GIT**, en vez de hacernos avanzar commit por commit como quizás hubiéramos hecho nosotros manualmente, nos movió automáticamente nuestro HEAD hasta un punto intermedio. En este punto, podemos volver a realizar los test correspondientes así le podemos, nuevamente, decir a **GIT** si este commit es “bueno” o “malo” :

```
$ npx mocha test*.js

Git bisect demo
#foo()
  ✓ should not do anything

Git bisect demo
#logic()
  ✓ should return the sum when adding zero
  ✓ should return the sum when adding to zero
  ✓ should return the sum
1) should return the sum for negative numbers
  ✓ should return the sum
  ✓ should return zero
  ✓ should not equal zero

7 passing (16ms)
1 failing

1) Git bisect demo
  #logic()
    should return the sum for negative numbers:
      AssertionError [ERR_ASSERTION]: -20 == 20
      + expected - actual
      --20
      +20
      at Context.<anonymous> (test1.js:16:11)

spame@DESKTOP-97V6LM8 MINGW64 ~/Documents/Projects/git-bisect-demo ((16a74f5...)|BISECTING)
$ git bisect bad
Bisecting: 8 revisions left to test after this (roughly 3 steps)
[6f352705018971a87ed9f3335a6c3bd8c594cc23] Add mocha to eslint environment
```

Nuevamente **GIT** va a mover nuestro HEAD a un punto intermedio entre los commits que le hayamos dicho que eran “malos” y “buenos” :

```
$ git log --oneline --graph --all
* b7485ac (origin/master, origin/HEAD, master) Fix he
* 3def738 Add links to screenshots
* 8d517bb Consistently use bash
* f80c3a9 Try with sh
* 329c3f1 Use shell instead of bash for formatting
* a5f5283 Fix formatting in clone command
* 3f9d8de Edit Readme some more
* 98e7889 Edit Readme
* 1773712 Add repo to readme instructions
* 143e5ed Add repo to package.json
* 093a764 Delete whitespace
* 7407c96 Add section about automating git bisect
* a0007bc Delete whitespace
* 4796c0c Update Readme
* 9dfa43b Add more instructions
* 32d3f53 Delete whitespaces
* 24cfbe8 Update Readme some more
* 645c892 Update Readme
* 10ec17f Add linting to test script
* 16a74f5 (refs/bisect/bad) Add more comments
* 1697fc8 Explain logic() function
* e3f3e49 Fix wording in Readme
* d3d2319 Delete whitespace
* bc9b1f9 Improve Readme
* d8951c8 Delete new lines
* 115637f Edit Readme
* 3d7a348 Add comment
* 6994316 Delete whitespace
* 6f35270 (HEAD) Add mocha to eslint environment
* 26da05b Fix most lint problems
* 56cf49 Add lint job, lots of failures
* 88f6e15 Move dependencies to devDependencies
* 599f373 Install eslint
* a332e39 Delete whitespace
* 179f12a Edit Readme some more
* 1ddca7e Edit Readme
* a2017ba Add Readme file
* b58217f (tag: allGreen, refs/bisect/good-b58217f071
* f34dea3 Install mocha, add dummy test
* c35e481 Initial import, no functionality
* d9abc2b Initial commit
```

De la misma manera que anteriormente, volvemos a realizar nuestros tests para verificar el estado del programa :

```
spame@DESKTOP-97V6LM8 MINGW64 ~/Documents/Projects/git-bisect-demo ((6f35270...)|BISECTING)
$ npx mocha test*.js

Git bisect demo
#foo()
  ✓ should not do anything

Git bisect demo
#logic()
  ✓ should return the sum when adding zero
  ✓ should return the sum when adding to zero
  ✓ should return the sum
  ✓ should return the sum for negative numbers
  ✓ should return the sum
  ✓ should return zero
  ✓ should not equal zero

  8 passing (18ms)
```

Pero en este punto nos dimos cuenta que el programa si funciona correctamente. Entonces volvemos a usar nuestro subcomando *git bisect good* para indicarle que ese commit es

“bueno”. De este modo **GIT** va reduciendo nuestras opciones de búsqueda a dos campos, ahorrandonos mucho tiempo en depuración y sobre todo automatización. Llegado el caso, nos va a recomendar los commits que considere que presentan las fallas :

```
spame@DESKTOP-97V6LM8 MINGW64 ~/Documents/Proyects/git-bisect-demo ((1697fc8...)|BISECTING)
$ git bisect good
16a74f57873ff63043c5b846c2e91eed4e64ceb6 is the first bad commit
commit 16a74f57873ff63043c5b846c2e91eed4e64ceb6
Author: Franziska Hinkelmann <franzih@chromium.org>
Date:   Sat Nov 3 09:34:04 2018 -0400

Add more comments
Drive-by fix: Small refactoring.

index.js | 5 +++--
1 file changed, 3 insertions(+), 2 deletions(-)
```

Una vez que hayamos terminado el proceso, podemos simplemente ejecutar el subcomando *git bisect reset* el cual va a volver nuestro HEAD a la posición original, de esta manera permitiéndonos continuar con nuestro trabajo :

```
spame@DESKTOP-97V6LM8 MINGW64 ~/Documents/Proyects/git-bisect-demo ((1697fc8...)|BISECTING)
$ git bisect reset
Previous HEAD position was 1697fc8 Explain logic() function
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

spame@DESKTOP-97V6LM8 MINGW64 ~/Documents/Proyects/git-bisect-demo (master)
$
```

PUNTOS DE VERSIONADO : ETIQUETAS

Imaginemos por un momento que tenemos una versión de un proyecto nuevo que arrancamos la cual está lista para ser probada en producción. Su último commit probablemente haya sido el último arreglo que se le habrá hecho a un bug o la unificación (*merge*) de algún branch de test en master. De alguna u otra forma, si bien tenemos el identificador de cada commit, que pasaría si quisieramos dejar solo seguimiento de aquellas versiones que ya pasaron por todos los testeos necesarios y son versiones estables.

Cada vez que necesitamos generar un nuevo “lanzamiento” o release de nuestra aplicación y dejar rastro de ello para poder decirle a algún usuario específicamente qué versión del programa debería estar usando podemos generar tags.

QUÉ ES UN TAG ?

Un tag es simplemente una referencia al hash de un commit. Con solo esta definición podríamos interpretar entonces que un tag y un branch son la misma cosa; entonces qué los diferencia? Principalmente la idea de un branch es mantener un nombre como referencia a un commit, pero además poder ir actualizando el hash de dicha referencia en el caso en donde se hayan confirmado nuevos cambios. Un tag, por otro lado, una vez que crea su referencia con un commit, el mismo no puede cambiar de valor.

Podemos crear un tag en el commit donde estemos parados con el siguiente comando :

```
> git tag <nombre>
```

Tengamos en cuenta también que podemos crear tags como referencias de commits que ya fueron creados hace tiempo de la siguiente manera :

```
> git tag <nombre> <commit>
```

TAG ANOTADO VS NO ANOTADO

Podemos observar también que vamos a encontrarnos con dos tipos de tags : los anotados y los no anotados.

Hagamos foco primero que ambos dos sirven para exactamente lo mismo : dejar una referencia de un commit en nuestro historial.

Los tags no anotados nos permiten darle un nombre al mismo tag que está creando pero no nos deja establecer un mensaje. Los mismos se guardan bajo la referencia `refs/tags` pero no son enviados al momento de sincronización con el repositorio remoto.

Un tag no anotado siempre es el que se crea por defecto a menos que especifiquemos opciones de configuración del comando.

Los tags anotados, por otro lado, además de permitirnos también darle un nombre al tag nos da la posibilidad de dejar un mensaje adicional. Esto va a crear un nuevo objeto en la base de datos de objetos de **GIT** con lo cual son referencias ideales para establecer lanzamientos oficiales de una aplicación, ya que los mismos pueden ser subidos al repositorio remoto para que los demás usuarios puedan usar esas versiones específicas de la aplicación.

Podemos crear un tag anotado de la siguiente manera :

```
> git tag -a <nombre> -m <mensaje>
```

SUBIENDO NUESTROS TAGS A UN CLIENTE DE GIT

Tengamos en cuenta que los tags de **GIT** no se suben de manera automática como pasa con los commits cada vez que hacemos un push.

Podemos subir cada tag de manera individual de la siguiente manera :

```
> git push origin <nombre>
```

Ejemplo :

```
> git push origin v.1.0
```

Si tenemos demasiados tags que tenemos que subir a nuestro repositorio remoto podemos optar por subir todos los tags al mismo tiempo así :

```
> git push origin --tags
```