educación IT
CODE YOUR FUTURE

Git:
Desarrollo colaborativo

Módulo 5

Bisect

El comando Bisect

Este comando permite **revisar el historial de `commits` y testear el proyecto, en cada uno de los puntos de restauración anteriores,** para identificar dónde surgió el problema realmente.

También se puede realizar la misma tarea en forma manual al realizar un checkout a un `commit` anterior en el historial, realizar los test correspondientes en nuestro programa y verificar si presentan inconsistencia.

Es decir, **avanzar `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. **Permite "automatizar" el trabajo de verificación de `commits`** y realizar una búsqueda binaria sobre nuestro historial de cambios.

Aquí se observa el caso en forma gráfica:

```
$ 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
* 56cfd49 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
```

Tomemos como ejemplo el repositorio de la imagen anterior. Como se puede observar, el historial es bastante extenso y, actualmente, el **`HEAD`** se encuentra en el **`commit b7485ac`**, que es también donde está el *branch master.*

Este *branch* 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)
```

Para revisar dónde se generó el error puntualmente: se toma como referencia el **`commit b58217f`** que muestra un tag llamado **`allGreen`**. Se sabe de antemano que, hasta ese momento puntual, el proyecto venía funcionando en forma correcta.

Primero se debe iniciar la operación de bisect de Git con el subcomando **`git bisect start`**:

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

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

El shell indica que ahora estamos en **modo “Bisecting”.** El paso siguiente será indicarle a Git un punto fijo en el que se considera que el proyecto funcionaba sin problemas y otro en el que ya no esté funcionando.

En este momento de la operación, resulta indistinto qué dos `commits` se indiquen, de cualquier manera el procedimiento siempre va a ser binario e intentará ahorrarnos el mayor tiempo posible.

Para indicarle a Git qué **`commit`** es el que funciona y cuál no, se utilizan los subcomandos **`git bisect good`** y **`git bisect bad`**.

Entonces, la mitad del trabajo está hecho porque ya contamos con ambos: el `commit` donde estamos actualmente presenta fallas y hemos identificado un `commit` anterior donde el programa funcionaba correctamente (`b58217f`).

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

spame@DESKTOP-97V6LM8 MINGW64 ~/Documents/Proyects/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
```

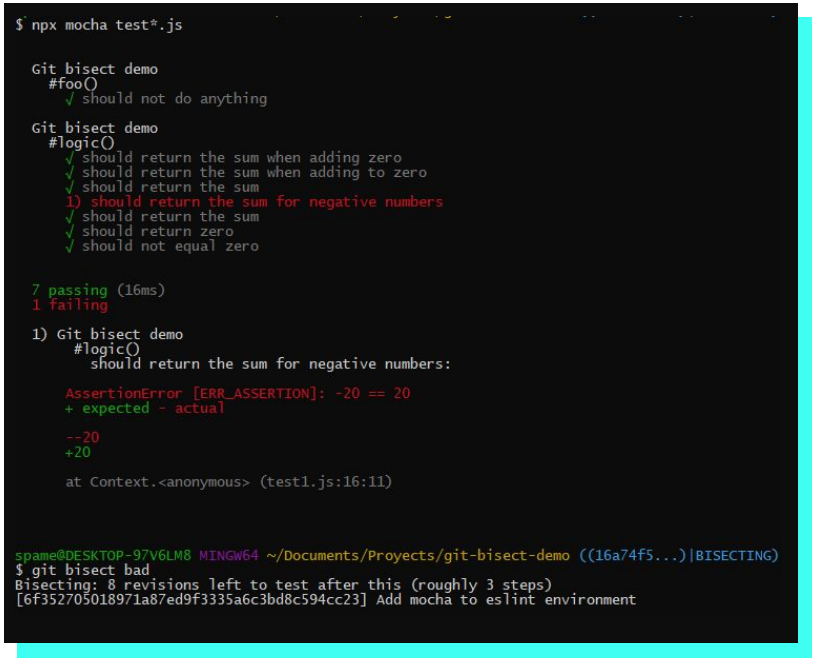
Como se observa en la imagen anterior, simplemente con ejecutar el comando **`git bisect bad`**, **Git generará un marcador temporal que identifica a ese `commit` como "malo".** A continuación, nos moveremos hasta el `commit` que consideramos que funcionaba correctamente y volvemos a realizar los tests 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)
```

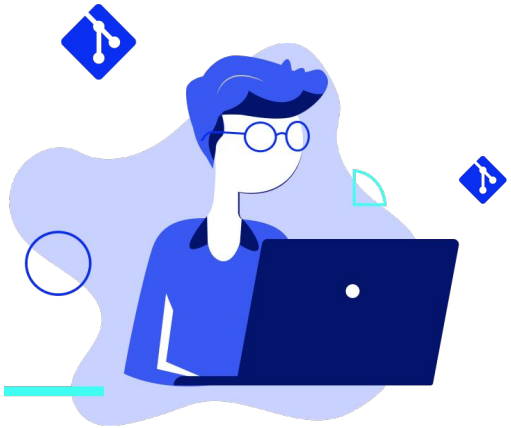
Como se confirmó que, realmente, en este `commit` funcionaba todo, **indicamos a Git que este `commit` es "bueno" con el subcomando `git bisect good`**:

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

spame@DESKTOP-97V6LM8 MINGW64 ~/Documents/Proyects/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
* 56cfd49 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
```

Al volver a revisar el log, se observa que Git, en lugar de hacernos avanzar `commit` por `commit` como quizás hubiéramos hecho manualmente, movió automáticamente el `HEAD` hasta un punto intermedio. En este punto, se pueden volver a realizar los test correspondientes para indicar, nuevamente, a Git si este `commit` es “bueno” o “malo”.



Nuevamente Git va a mover el `HEAD` a un punto intermedio entre los `commits` que le hayamos indicado 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
* 56cfd49 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
```

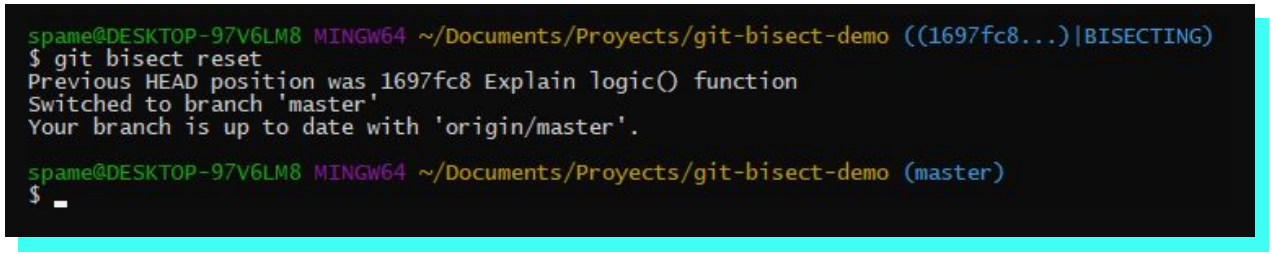
De la misma manera, volvemos a realizar los tests para verificar el estado del programa:

```
spame@DESKTOP-97V6LM8 MINGW64 ~/Documents/Proyects/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)
```

En este punto, se advierte que el programa funciona correctamente. Entonces, se vuelve a usar el subcomando **`git bisect good`** para indicar que ese **`commit`** es “bueno”. De este modo, Git va reduciendo nuestras opciones de búsqueda a dos campos y nos ahorra mucho tiempo de depuración. Llegado el caso, recomendará 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 terminado el proceso, se puede ejecutar el subcomando **`git bisect reset`** que **volverá el HEAD a la posición original.** De esta manera, el comando nos permite continuar con el trabajo:



¿Qué es un Tag?

Es una referencia al *hash* de un `commit`. Con esta definición podríamos interpretar que un `tag` y un *branch* son la misma cosa, ¿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 esa referencia en el caso en que se hayan confirmado nuevos cambios.

A diferencia de eso, **un `tag` no puede cambiar de valor una vez que crea su referencia con un `commit`.**

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

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

Se pueden **crear `tags` como referencias de `commits` que fueron creados hace tiempo** de la siguiente manera:

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

Tag anotado vs. no anotado

Podemos encontrarnos con **dos tipos de `tags`: los anotados y los no anotados.**

Ambos sirven para exactamente lo mismo: dejar una referencia de un `commit` en el historial.

Los **`tags` no anotados permiten darle un nombre** al mismo **`tag`** que está creando pero **no dejan establecer un mensaje.** 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 se especifiquen opciones de configuración del comando.

Los tags anotados, también permiten darle un nombre al **tag** pero, además, **brindan la posibilidad de dejar un mensaje adicional**. Se creará un nuevo objeto en la base de datos de objetos de Git.

Por eso son referencias ideales para establecer lanzamientos oficiales de una aplicación, ya que se pueden subir al repositorio remoto para que los demás usuarios usen esas versiones específicas de la aplicación.

Un tag anotado se crea de la siguiente manera:

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

Subir Tags a un cliente de Git

Los tags de Git no se suben de manera automática, como pasa con los `commits` cada vez que se realiza un `push`.

Se puede **subir cada tag de manera individual** de la siguiente manera:

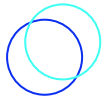
```
> git push origin <nombre>
```

Ejemplo

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

Si tenemos demasiados `tags` para subir al repositorio remoto, es posible optar por **subir todos los tags al mismo tiempo:**

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



El comando Revert

Antes de revertir, **es necesario chequear si los cambios que se desean eliminar ya fueron publicados al repositorio remoto o no**.

Revertir cambios que otras personas pudieron potencialmente haber descargado 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 a la de mis propios compañeros.

En algún momento del trabajo alguien no va a poder actualizar su propio trabajo o/ni subir cambios al remoto, ya que no se estaría tratando técnicamente del mismo repositorio, el grafo entero cambió.

Si este es el caso, contamos con el comando **`git revert`**, **que permite elegir uno o un conjunto de `commits` y revertir los cambios que ellos implementan.**

```
> git revert <commit>
```

Como muestra el gráfico anterior, hay un `commit` conflictivo en nuestro *branch*, el *abcd12ef* y queremos revertir sus cambios. Para eso, usamos el siguiente comando:

```
> git revert abcd12ef
```

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.

¡Sigamos trabajando!