educación IT
CODE YOUR FUTURE

Git:
Desarrollo colaborativo

Módulo 5

Rebase interactivo

Rebase interactivo

Existe otra manera de poder deshacer cambios y de ordenar, simplificar, juntar y editar `commits`: con el comando **`git rebase`**. Si pensamos en su utilidad, recordaremos que **permitía ubicar todo un *branch* para que su *branch* de origen sea otro, distinto desde el que se originó.**

Gracias a la **herramienta interactiva del `rebase`** podemos entrar en una interfaz distinta que nos habilitará nuevas opciones para aquellos `commits` a los que queremos hacerles `rebase`:

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

Se utiliza el *flag* `--i` para poder entrar en esta interfaz interactiva del rebase:

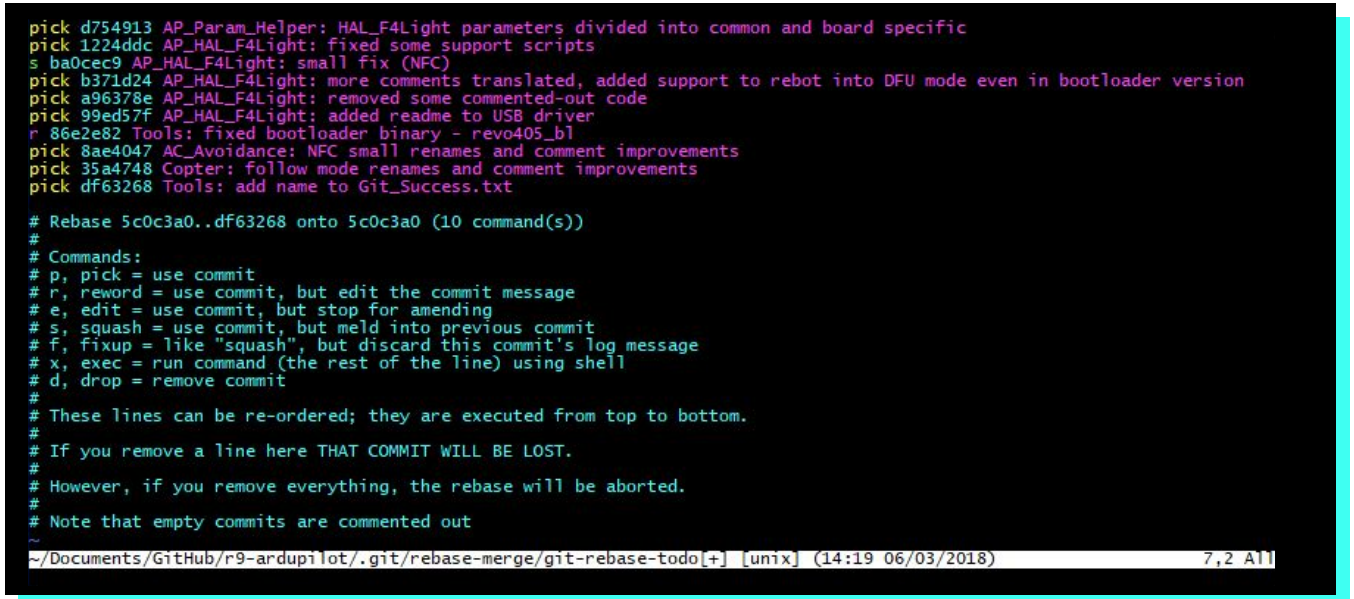
```
pick d2f7add Change 1
pick 390275c Change 2
pick 48f3b48 Change 3

# 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 All
<ers/gaui/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).**

Veamos una serie de comandos:

p	**pick**	Permite mover `commits` de lugar.
r	**reword**	Hace posible entrar en modo editor y poder volver a escribir el mensaje de ese `commit` en particular.
e	**edit**	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**	Sirve para 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**	Realiza una operación similar al `squash` pero descarta todos los mensajes de los `commits` que vamos a fusionar.

Todo lo que hay que hacer es cambiar los prefijos de aquellos `commits` que se desea editar con alguno de los comandos anteriores y salir de la interfaz de VIM:

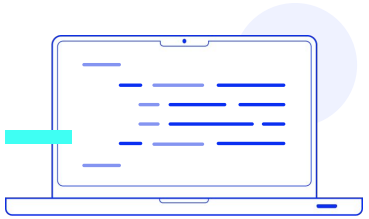


Al confirmar los cambios y salir, el **rebase** va a continuar y aplicará los comandos uno por uno. Se detendrá entre cada uno y revisará que no se generen conflictos. Si encuentra algún problema, todo el **rebase** se pausará. En ese momento, podremos solucionar lo que haya salido mal y continuar con el proceso.

Reflog

Las puntas de *branches* y otras referencias quedan guardadas en el *log de referencias* o `reflog`, en el repositorio local.

El **`reflog`** cubre todas las acciones recientes como cambios de *branch*, `resets`, `commits`, y otras. A medida que avanzamos con el trabajo, Git guarda cada movimiento que realiza el `HEAD`. **Cada vez que se efectúa una acción, se actualiza el `reflog`.**



Continuamos con el ejemplo anterior del archivo *server.js*.
Así se encuentra nuestro **`reflog`**:

```
> 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, por ahora es un simple listado de los últimos dos `commits` que se realizaron, que también podemos obtener con un simple **`git log`**. Sin embargo, la primera línea muestra el momento en el que hicimos el clon del repositorio en la máquina local.

Restaurar trabajo

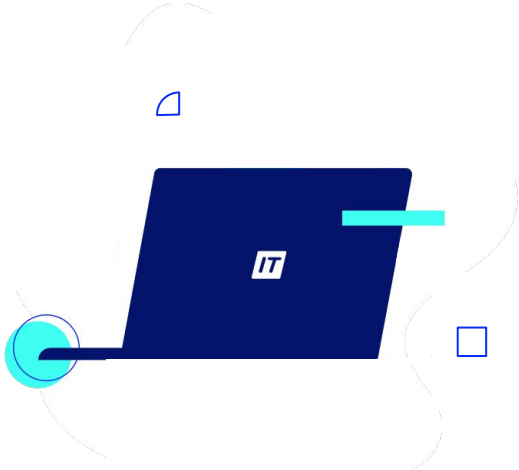
Vamos a proceder a realizar un experimento. Imaginemos que, sin querer, hacemos un **`git reset --hard`** sobre nuestros archivos. Esto pondría nuestro `HEAD` en un estado anterior de trabajo y haría que perdamos los `commits` en donde estaba el trabajo realizado.

Utilizar el **`reflog`**, podría servir para conocer el momento en el que se hizo ese cambio.

Ejemplo

```
> 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 es posible 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`. Veámoslo en la siguiente diapositiva.



```
> 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
```

Es importante notar 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, **usamos la referencia `HEAD@{1}` que se muestra en el listado del `reflog`**. Así nos movemos a un momento en el pasado, a través del cual transitó el `HEAD`.

Estaremos parados en un `commit` que aún no fue borrado por nuestro repositorio local, pero que no pertenece a ninguna parte del árbol real de commits, por eso en el ejemplo de la pantalla anterior aparece el mensaje de ***`detached HEAD`*** y en el `log`, que se muestra en el ejemplo debajo, no veremos 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, creamos un *branch* en el que estemos parados en ese momento. 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
```

¡Sigamos trabajando!