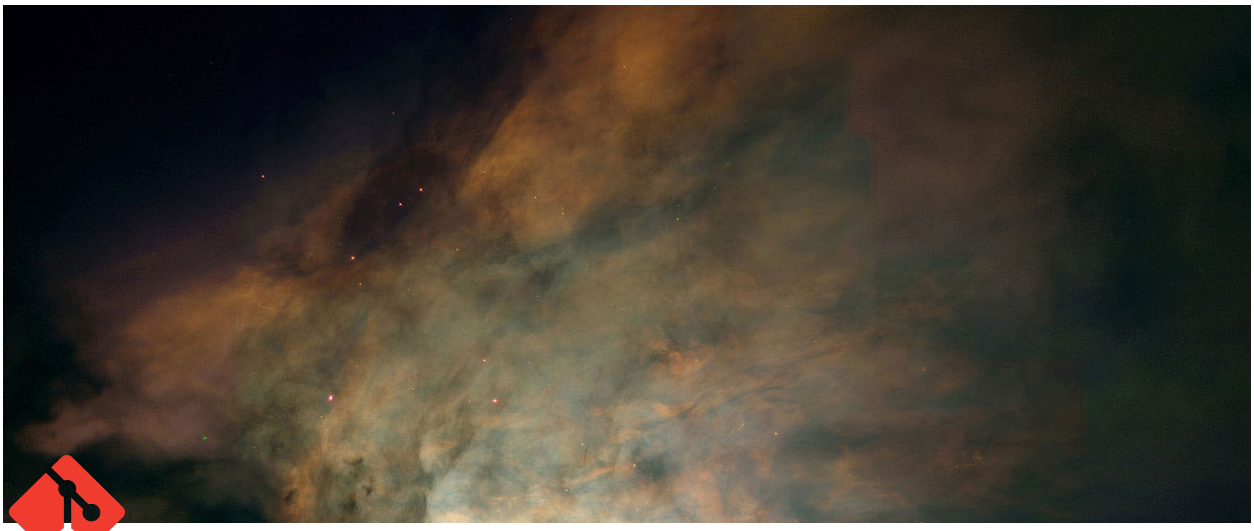


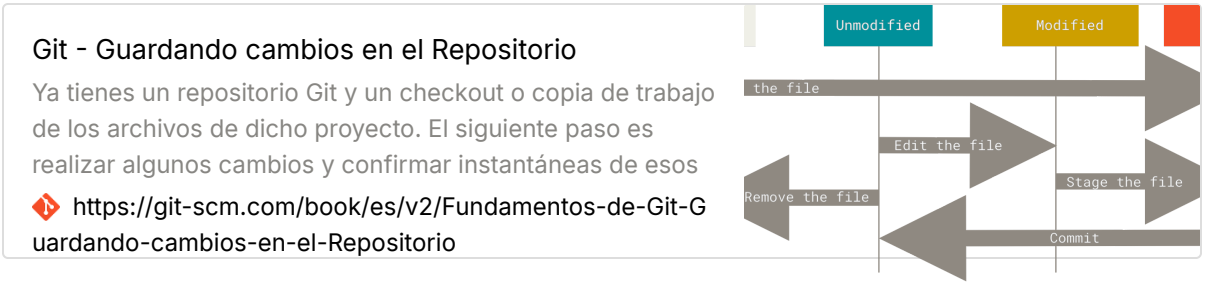
GIT

Que es Git

Control de versiones para poder trabajar en equipo y que cada version que se realice quede almacenada en un repositorio.

Git bash es la terminal de git la cual nos permite trabajar con comandos de linux dentro de windows

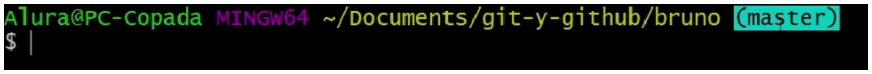
Algo sobre los comandos de Git



git init

Es decir que en una carpeta hay un repositorio, ahi tenemos el repositorio inicializado

Lo que esta en azul quiere decir que esta en una rama y en este caso es la rama
"master"



git init —bare ese bare indica que ese repositorio es puro y contiene las modificaciones del archivo, solo controla y trae unas facilidades

git remote add <Ruta del servidor remoto>

git remote ⇒ me saldra el servidor remoto

git remote -v ⇒ saldra los servidores que estan activos

git clone ⇒ directorio

git push servidorLocal master

git remote rename origin local ⇒ renombrar

▼ git status

Nos informa el estado de nuestros archivos, si tienen commits, modificaciones

▼ git add

Nos añade el archivo en el monitero de git

▼ git commit -m

▼ git log

Miramos el historico de nuestro repositorio

git log —oneline

git log -p

https://devhints.io/git-log

> https://devhints.io/git-log

▼ git config —local user.email “email registrado”

▼ git config —local user.name “Nombre”

▼ git config local

Configurar email→ significa que las configuraciones que vamos a realizar solo van a hacer para este proyecto

▼ git config global

Se realizara de manera global con todos los proyectos

▼ Algo sobre git

Al ejecutar el comando `git status`, recibimos información que puede no ser tan clara, especialmente cuando nos encontramos con términos como `HEAD`, `working tree`, `index`, etc.

Solo para aclarar un poco, ya que entenderemos mejor cómo funciona Git durante el curso, aquí hay algunas definiciones interesantes:

- `HEAD`: Estado actual de nuestro código, es decir, donde nos colocó Git
- `Working tree`: Lugar donde los archivos realmente están siendo almacenados
- `index`: Lugar donde Git almacena lo que será *commiteado*, es decir, la ubicación entre el *working tree* y el repositorio de Git en sí.

mkdir para crear carpetas

▼ Branches

Las *branches* ("ramas") se utilizan para desarrollar funcionalidades aisladas entre sí. La *branch* `master` es la *branch* "predeterminada" cuando creas un repositorio.

Es interesante separar el desarrollo de funcionalidades en diferentes *branches*, para que los cambios en el código de una no influyan en el funcionamiento de otra.

En esta aula, entenderemos mejor cómo trabajar con estas ramas, pero es muy importante que comprenda su propósito.

En otros cursos aquí en Alura, hablaremos más sobre estrategias para organizar tus *branches*, entonces no te preocupes tanto por eso ahora. ;-)

Por Convecncion cuando se trabaja en equipo es mas favorable trabajar en cada uno su rama y dejar la rama principal solo para integrar lo que se ha venido trabajando

- ▼ git branch

  Nos indica las ramas que hay y en la que nos encontramos parados

- ▼ git branch <Nombre de la rama>

  Crear nueva rama

- ▼ git checkout <Nombre de la rama>

  Cambiar de ramas

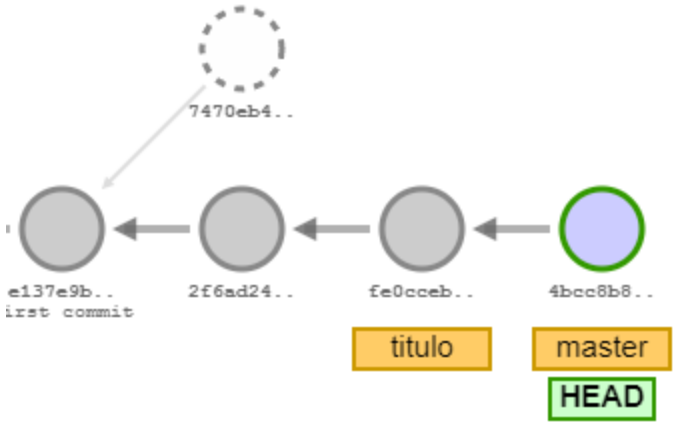
- ▼ git checkout -b <Nombre de la branch>

  Crear una rama y trasladarse de una vez a ella

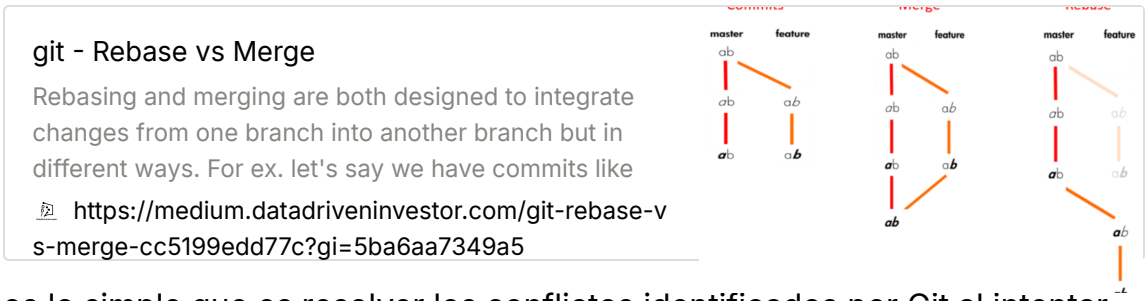
- ▼ git merge <Nombre de la rama a la que se hara el merge>

  Uniendo dos ramas por ejemplo estoy editado en la rama titulo y voy a hacer merge a la rama master, me ubico en la rama master y hago el git merge <nombre de la ram>

- ▼ git rebase <Nombre de la rama>



El `merge` junta los trabajos y genera un `merge commit`. El `rebase` aplica los *commits* de otra Branch en la *Branch* actual.



Vimos lo simple que es resolver los conflictos identificados por Git al intentar hacer `merge`.

Ahora, genera un conflicto y, en lugar de usar `merge`, usa `rebase` para actualizar el `master`:

- Vé a la *branch* `titulo`
- *Commitea* algo
- Vé a la *branch* `master`, *commitea* un cambio en la misma línea
- Ejecuta `git rebase titulo`

Mira la salida de Git y usa la información que te da; después de corregir el conflicto, continúa con el `rebase`.

▼ Control z en git

Con el `git restore` deshacemos una modificación que aún no fue agregada al `index` o `stage`, o sea, antes de hacer `git add`

. Después de agregar con `git add`, para deshacer una modificación, necesitamos sacarlo de este estado, con `git restore --staged`. Ahora, si ya hicimos el `commit`, el comando `git revert` puede salvarnos.

- ▼ git restore

  git restore <Nombre del archivo>

  git restore —staged <Nombre del archivo>

- ▼ git revert

  git revert <Codigo del commit>, este hace un rollback de un commit ya hecho y devuelve a la version anterior

- ▼ git stash

  Guardar las cosas que no se han comiteado

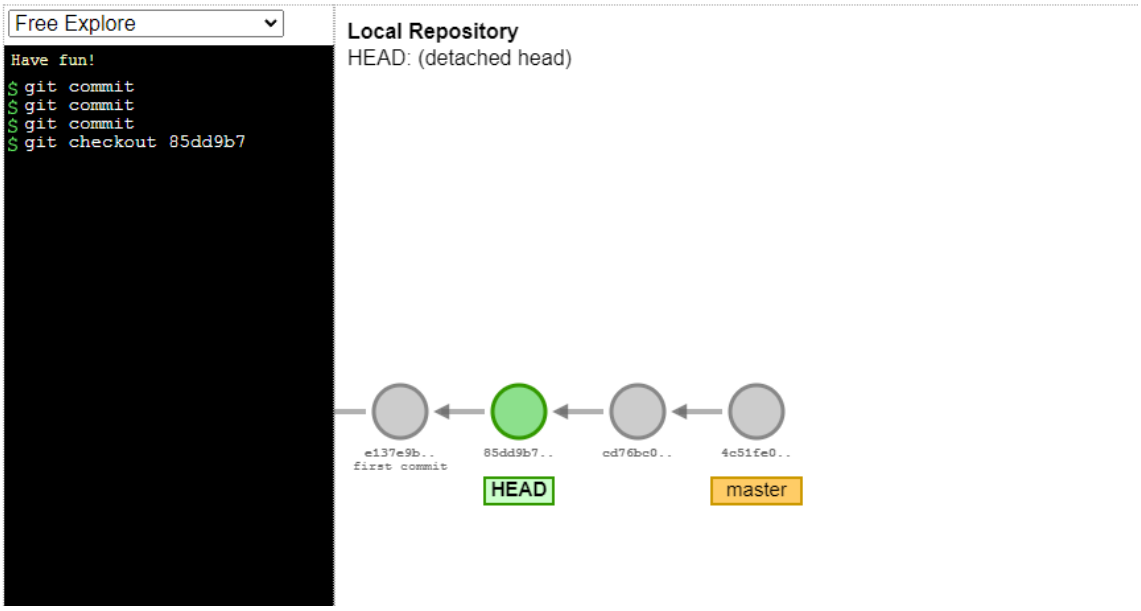
  git stash list

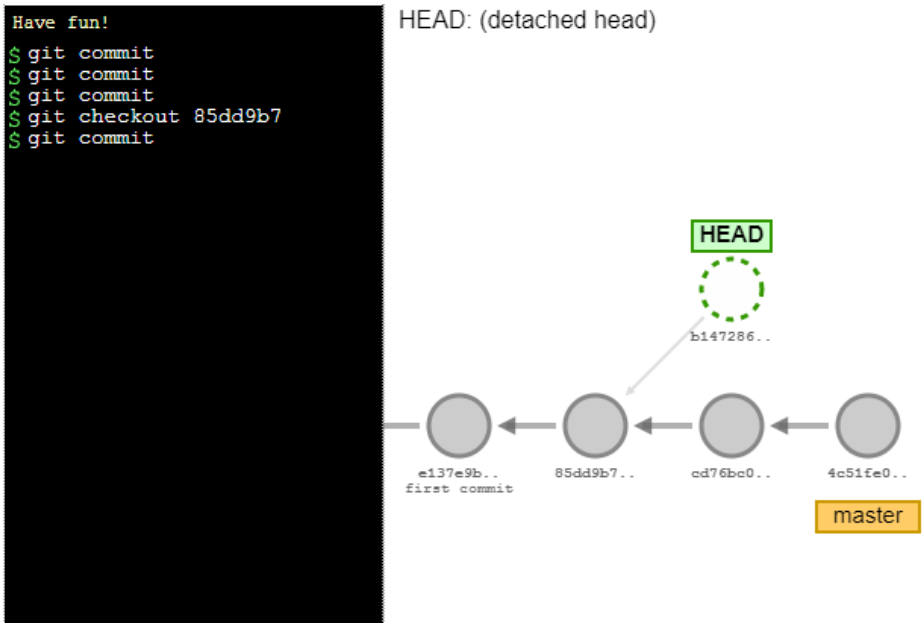
  git stash apply 0 ⇒ me aplica los cambios realizados pero me deja el la lista stash

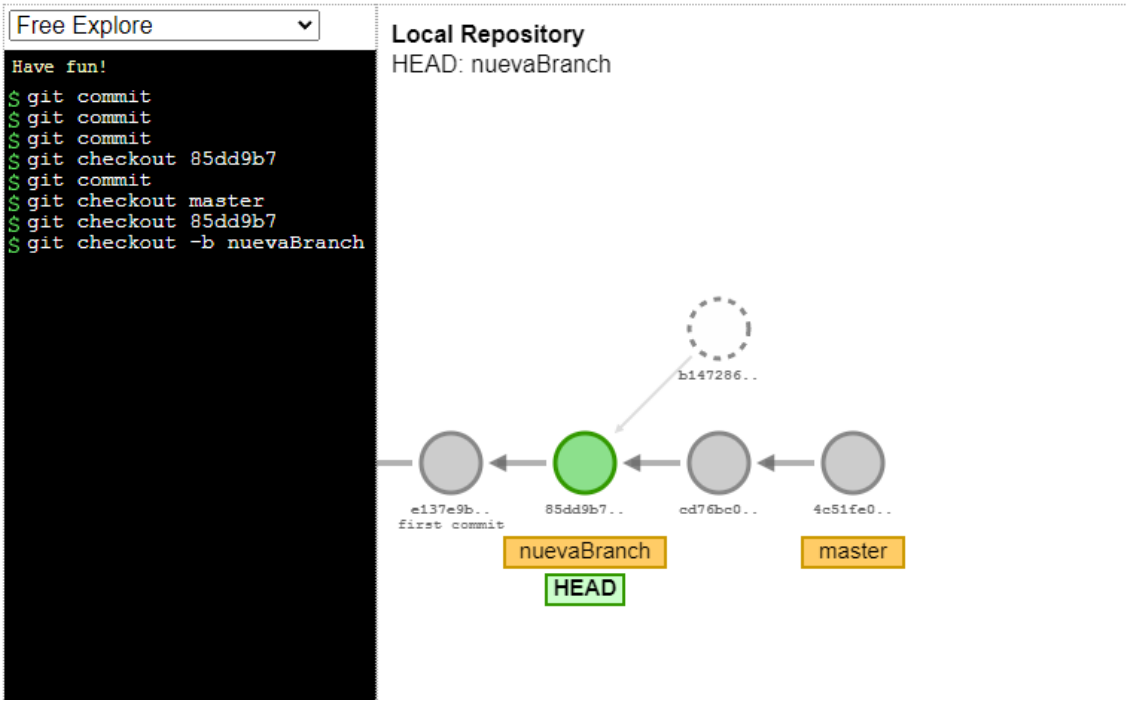
  git stash pop ⇒ me saca los cambios que habia dejado y sale de la lista stash para quedar en lo que habias trabajado con la actualizacion que se le hizo inmediatamente

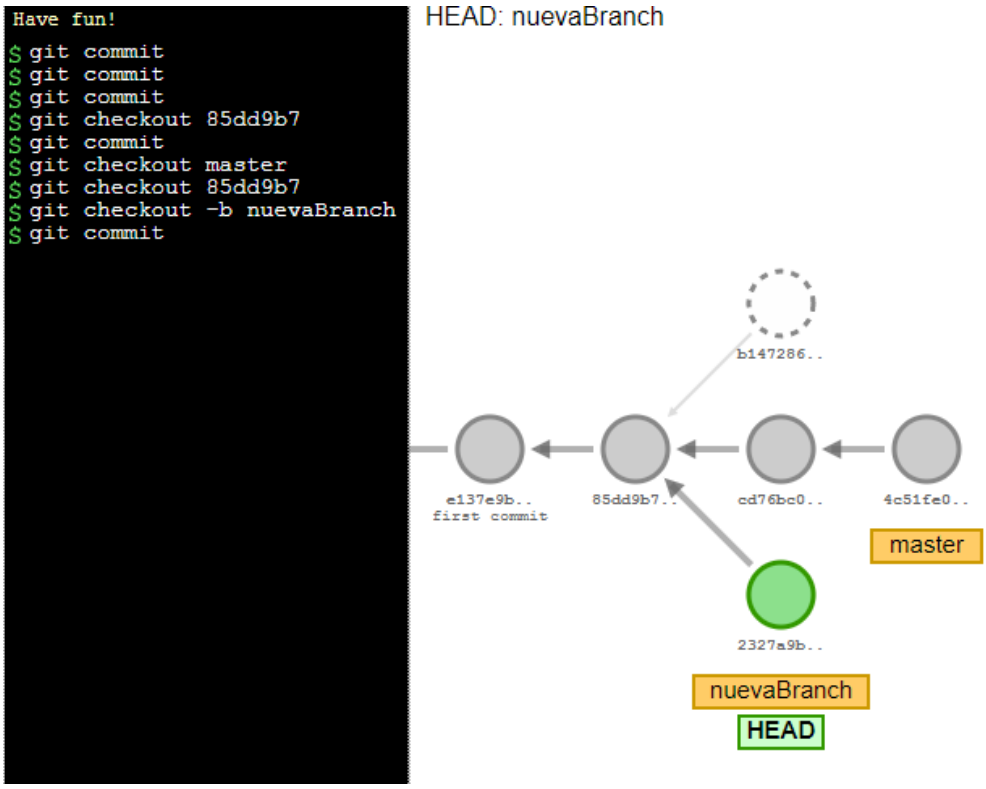
- ▼ git checkout <Numero commit>

  Podemos ver los cambios realizados en ese commit, ya sea cambiar, actualizar


Free Explore
Have fun!
$ git commit
$ git commit
$ git commit
$ git checkout 85dd9b7
Local Repository
HEAD: (detached head)
e137e9b..
first commit
85dd9b7..
cd76bc0..
4c51fe0..
HEAD
master


Have fun!
$ git commit
$ git commit
$ git commit
$ git checkout 85dd9b7
$ git commit
HEAD: (detached head)
HEAD
b147286..
e137e9b..
first commit
85dd9b7..
cd76bc0..
4c51fe0..
master





▼ Diferencia entre commits

▼ git diff

Se compara lo que se ha venido realizando

git diff <Codigo commit>..<Codigo commit>

▼ Version Final

▼ Tag

Un tag marca un punto en nuestra aplicación, un punto que no puede ser modificado, fijo, por ejemplo el lanzamiento de una versión y la versión 0.1, una vez lanzada nunca más es modificada. Cualquier modificación que yo quiera agregar va a ser agregada a la 0.2 y no a la 0.1.

git tag -a <Nombre que le voy a dar> -m ”Commit que se le dara”

git push <servidor> <nombre que le dimos al tag>

git log -n 2