



# Argentina Programa





# Taller de Github parte 2





# Configuraciones de un repositorio Git



# Configuraciones Git



A continuación estudiaremos una serie de parámetros requeridos para la configuración de nuestro entorno de trabajo, y posterior vinculación entre nuestro desarrollo local y nuestro trabajo remoto.

**Identidad:** inmediatamente luego de instalar Git, será necesario establecer nuestro nombre de usuario y dirección de email, ya que una de las ventajas de git es que podemos ver quién modifica el código, a partir de los commits y sus respectivos autores.

```
git config --global user.name "John Doe"  
git config --global user.email johndoe@example.com
```



# Configurando repositorio remoto



Una forma sencilla de comprender el vínculo entre el desarrollo local y el remoto es comenzar creando un nuevo repositorio vacío.

Esto se hace desde la página de GitHub , o la plataforma que se adapte a nuestras necesidades (GitLab, BitBucket, etcétera).

...or create a new repository on the command line

```
echo "# CoderHouseGit" >> README.md  
git init  
git add README.md  
git commit -m "first commit"  
git branch -M main  
git remote add origin https://github.com/Pedroottaviano/CoderHouseGit.git  
git push -u origin main
```



...or push an existing repository from the command line

```
git remote add origin https://github.com/Pedroottaviano/CoderHouseGit.git  
git branch -M main  
git push -u origin main
```



# Configurando repositorio remoto



Como vimos en la imagen anterior, ni bien creamos un repositorio vacío en la página de GitHub el sistema detecta esto, y nos propone dos soluciones para importar nuestro proyecto al repositorio (una tercera es directamente importar desde otro repositorio).

Dado que ya tenemos un proyecto a nivel local con el que venimos trabajando, avanzamos con la segunda opción, ejecutando lo siguiente:

```
git remote add origin https://github.com/Pedroottaviano/CoderHouseGit.git
git remote -v
origin  https://github.com/Pedroottaviano/CoderHouseGit.git (fetch)
origin  https://github.com/Pedroottaviano/CoderHouseGit.git (push)
```



# Configurando repositorio remoto



En el caso anterior, como Git soporta múltiples repositorios remotos dentro de un mismo proyecto, al hacer “**git remote add origin**” le estamos dando el nombre “**origin**” (se suele usar ese por convención) a nuestro repositorio remoto.

Una forma de chequear esto es mediante “**git remote -v**”. La salida de este comando nos indica que enlaces del tipo fetch o pull emplea el remote llamado “**origin**”, con la URL que se detalla.



# fetch, merge, push and pull



Al momento de relacionar el contenido local con el contenido remoto, haremos uso de los siguientes comandos:

- **git merge “mergedBranch”**: fusiona, une contenidos; es el único comando de los citados que también podemos encontrarlo trabajando sólo de manera local. En este caso, fusionamos los cambios de “mergedBranch” a donde estemos ubicados.
- **git fetch**: se encarga de bajar los cambios a una rama temporal, para que el usuario pueda ver los cambios, y a partir de allí decidir si mergear o no.
- **git pull**: descarga los cambios en la rama y automáticamente los fusiona a la rama en cuestión (combinación de fetch y merge). Por lo general, sobre todo en etapas iniciales, se recomienda el proceso previo, para mayor seguridad.
- **git push**: “empuja” los cambios de la rama local a la rama remota





# Vamos a practicar lo visto!

- Flujo local y remoto de git -



# Flujo de trabajo entre local y remoto (1/4)



Para la demostración de cómo trabajan los comandos anteriormente mencionados, haremos modificaciones desde la plataforma de GitHub a un archivo, y veremos las distintas formas de introducir esos cambios en nuestro repositorio local:

```
git fetch  
remote: Enumerating objects: 5, done.  
remote: Counting objects: 100% (5/5), done.  
remote: Compressing objects: 100% (2/2), done.  
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0  
Unpacking objects: 100% (3/3), 669 bytes | 15.00 KiB/s, done.  
From https://github.com/Pedroottaviano/CoderHouseGit  
      b146f5b..b03aebb  master       -> origin/master
```



# Flujo de trabajo entre local y remoto (2/4)



El comando anterior nos informa que hay modificaciones en el repositorio remoto; sino hubiera cambios, **git fetch** no devolvería nada. Para ver estos cambios directamente, podemos movernos a la rama que nos sugiere (equivalente local a la rama remota) **origin/master**.

Otro recurso es usar el comando **git diff rama1..rama2**, versión que dará un detalle completo. Para verlo resumido ejecutamos lo siguiente:

```
git diff --stat master..origin/master
index.html | 4 +++
1 file changed, 4 insertions(+)
```



# Flujo de trabajo entre local y remoto (3/4)



Si al observar los cambios introducidos a la rama no detectamos interferencias o cambios indeseados, podremos fusionar los cambios que la rama remota pretende traer a nuestra rama local.

Obsérvese cómo se detallan los cambios introducidos en la actualización, y si ejecutamos git fetch no regresa nada, dado que la rama se encuentra actualizada.

```
git merge origin/master
Updating b03aeabb..478f8c6
Fast-forward
 index.html | 6 -----
 1 file changed, 6 deletions(-)
-----
git fetch
```



# Flujo de trabajo entre local y remoto (4/4)



```
git pull origin master
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 1 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 651 bytes | 43.00 KiB/s, done.
From https://github.com/Pedroottaviano/CoderHouseGit
 * branch           master      -> FETCH_HEAD
   478f8c6..4e54ef8  master      -> origin/master
Updating 478f8c6..4e54ef8
Fast-forward
 index.html | 6 -----
 1 file changed, 6 deletions (-)
```

Por último, volveremos a efectuar cambios en el archivo remoto, pero esta vez actualizaremos la rama local de manera directa.





# CONSULTAS?

Break





# Clone vs. Fork



# Colaborando en un repositorio



Existen dos formas difundidas de trabajar de manera colaborativa con repositorios:

- **clone (clonar)**
- **fork (ramificar)**

En las próximas slides detallaremos sus características, casos conocidos, y en qué momento se emplea cada sistema.



# Git clone



Probablemente estemos más familiarizados con este proceso que con el de **fork**.

Como su nombre lo indica, básicamente implica **clonar**, es decir, replicar de manera completa todo lo incluido en el repositorio inicial.

Todos los contribuidores al proyecto original seguirán ingresando cambios a lo que obtuvimos, así como la información relativa a estos cambios, pudiendo sincronizar nuestra versión clonada con cualquier versión posterior.





Por otro lado, al ramificar estamos creando una copia totalmente nueva del repositorio original, dándonos control completo sobre como irá variando este nuevo proyecto.

Los autores previos no tendrán acceso a ninguno de nuestros cambios introducidos, a menos que les demos el permiso.



# Célebres ramificaciones



En la industria de software, muchos son los proyectos conocidos que derivan de otros proyectos conocidos. Por nombrar algunos...

- Jenkins - Hudson
- Fire OS para Kindle Fire - Android
- LibreOffice - OpenOffice
- Ubuntu - Debian
- MariaDB - MySQL



# ¿Clonar o ramificar?



¿Por qué elegimos un proceso u otro? Se debe pensar en el contexto. En un desarrollo más **colaborativo**, lo habitual es clonar, dado que como estudiamos anteriormente, de esta forma es sencillo sincronizar con los comandos push y pull. Pero si el nuevo desarrollo necesita de mayor “aislamiento”, la elección será un **fork**.

A large, abstract graphic at the bottom of the slide features a grid of fine lines forming a perspective-like surface that slopes upwards from left to right. This graphic serves as a decorative background for the text below.

# ¿Clonar o ramificar?



Cuando se ramifica un repositorio, quienes vayan a trabajar sobre el código, de igual manera tendrán que clonar el repositorio original, pero a partir de haber ramificado previamente.

En este caso, se exemplifica con la web inicial de todo proyecto Angular, disponible en el repositorio

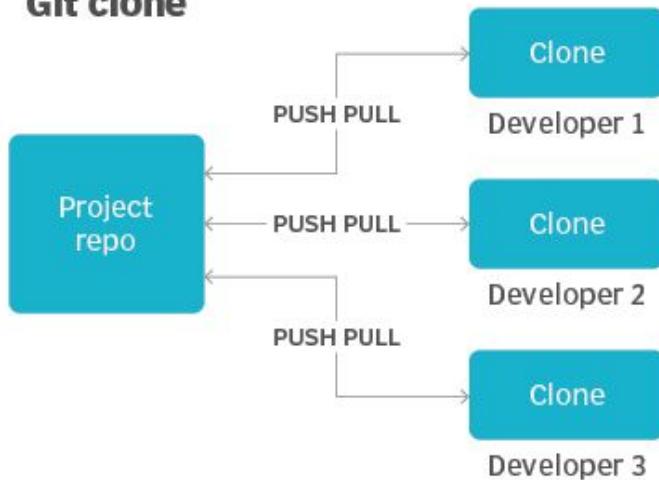
<https://github.com/johnpapa/angular-tour-of-heroes>. Haremos fork a este proyecto, y luego clonaremos lo generado para trabajar de manera local.



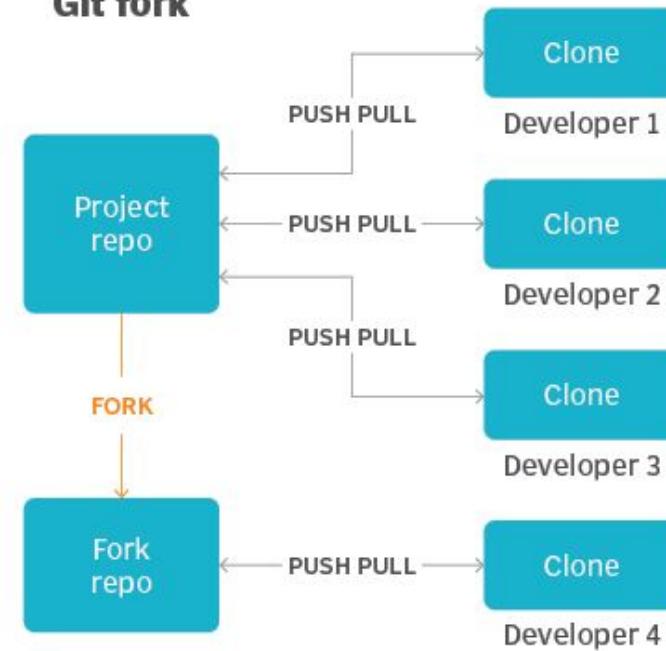
# De manera visual...



## Git clone



## Git fork





# Git stash, git log y git rebase



# Git stash (apartar)



Se emplea para “apartar” cambios desarrollados que aún no han sido confirmados. En situaciones, puede ser que urja dejar cierto proceso de lado para atender otro. Para no hacer un commit con procesos intermedios (se dice que esto “ensucia” el flujo de desarrollo), se aparta lo desarrollado hasta el momento con git stash. Por otro lado, podemos observar todos los stash creados con git stash list.

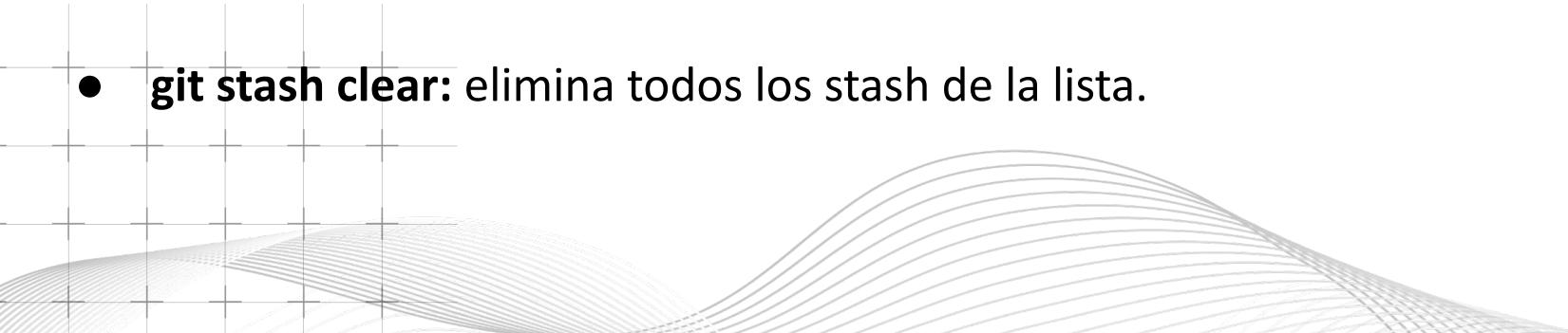
```
git stash
Saved working directory and index state WIP on master: 4e54ef8 Update index.html
-----
git stash list
stash@{0}: WIP on master: 4e54ef8 Update index.html
stash@{1}: WIP on master: 4e54ef8 Update index.html
```



# Git stash (variantes)



- **git stash drop <nombreDelStash>**: elimina el stash, sin aplicar los cambios.
- **git stash apply <nombreDelStash>**: aplica el stash sin eliminarlo.
- **git stash pop <nombreDelStash>**: aplica los cambios que guardaba el stash, y lo elimina de la lista (podemos pensar a este comando como a una unión de los dos anteriores).
- **git stash clear**: elimina todos los stash de la lista.



# Git log



Un detalle mencionado anteriormente, sobre ensuciar el historial de commits (se supone que como desarrolladores, no debemos realizar commits innecesarios, los mismos deben ser significativos). Se puede observar, empleando git log, que este comando nos devolverá el historial de la rama e información pertinente.

A su vez, nos proporciona los hash de cada commit, en caso de que queramos volver a alguno en específico. El mismo admite muchas variantes, una de las más usadas es **git log --online** o **git log --online --graph**

```
git log
commit d74220d6a89e0e1cbee39c4c343a018aa71c1b7d (HEAD -> master)
Author: Pedroottaviano <pedroottaviano@gmail.com>
Date:   Tue Nov 9 20:45:34 2021 -0300

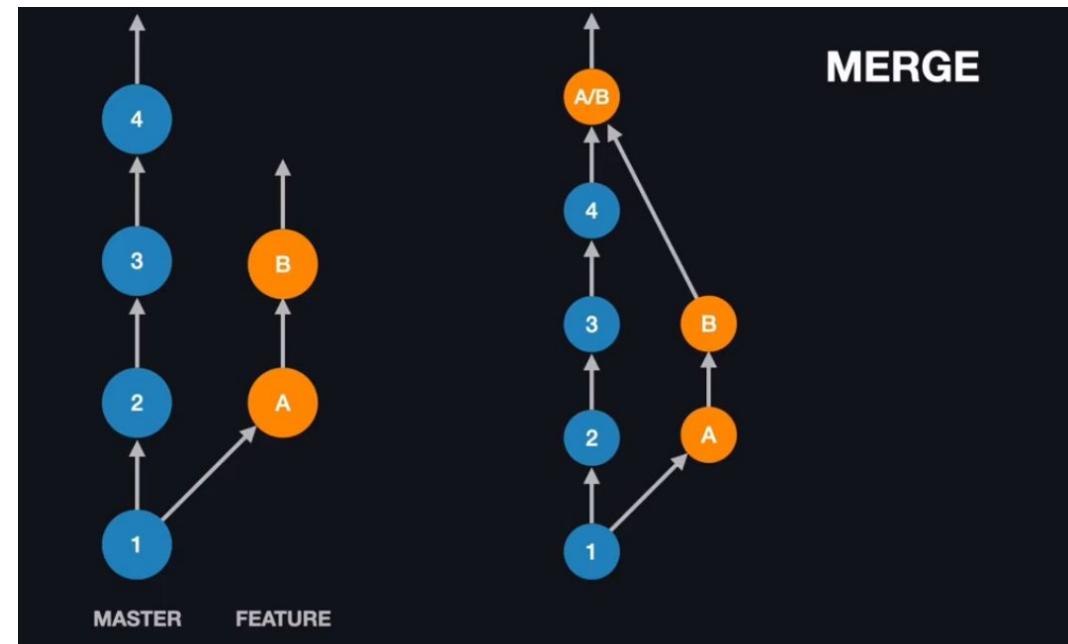
    trabajando con stash
```



# Git merge



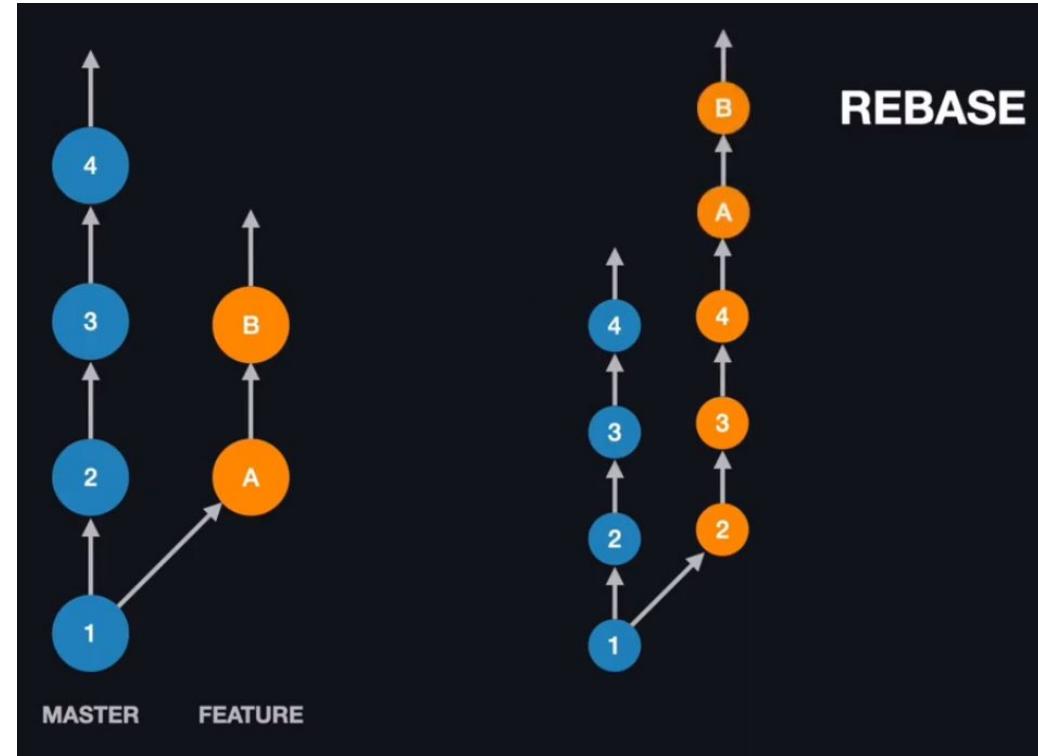
En la gráfica podemos ver cómo trabaja git merge. Cuando fusionamos una rama2 a una rama1, los cambios de la rama2 se insertan inmediatamente después de los cambios de la rama1.



# Git rebase



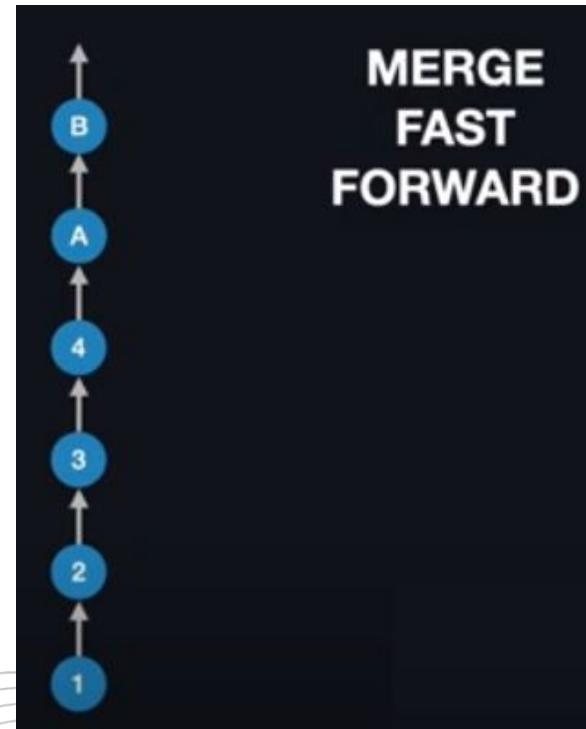
Por otro lado, el proceso de rebase se encarga de eliminar los commits de la rama2, traer a esta los commits de la rama1, volver a agregar los cambios de la rama1, y fusionar ambas en lo que se conoce como fast forward.



# Git rebase



A grandes rasgos, pueden pensarse como procesos equivalentes. Hay que tener presente que git rebase recopila uno a uno los cambios confirmados sobre una rama, y los aplica sobre otra. Esto puede pensarse como una actualización de la rama secundaria respecto de la principal, siendo de suma utilidad al momento de evitar conflictos.





# Git cherry pick, git revert y git reset



# Git cherry pick



git cherry pick, permite traer commits específicos de una rama a otra. Para esto nos apoyaremos en git log --oneline, para ver el listado de commits y sus respectivos hash, y en git log -p para conocer los cambios introducidos en cada commit. Hecho esto podremos seleccionar los cambios deseados.

En este caso, hemos decidido introducir los cambios generados por el commit “5ac213a”, “estructura lista”

```
git log --oneline
2b5984e (HEAD -> secondBranch) mensaje listo
5ac213a estructura lista
bb270a2 primer commit secondBranch
46dbf9b (master) primer commit en master
-----
git cherry-pick 5ac213a
```



# Git cherry pick



Para deshacer cambios en GIT, existen dos opciones, la primera y menos recomendado es git reset --hard [hashCommit], ya que esta resetea y por lo tanto elimina todos los commits posteriores al que estamos regresando. Por otro lado aparece revert, git revert [hashCommit], la cual nos guiará a través de un proceso para resolver conflictos, y volver al commit deseado, sin perder información.

```
git reset --hard aab45
```

```
-----  
git revert aab45
```





# CONSULTAS?

Muchas Gracias!

