```
$ git init
$ git add biografia.txt
$ git commit -m “versión 1”

$ git add .
$ git commit -m “Cambios a v1”
$ git status
$ git log biografia.txt
$ git push
```

iniciar git

agregar archivo al repositorio

para hacer una actualizacion a un archivo dentro del repositorio

Guardar cambios en el repositorio

Ver el estado de nuestra base de datos

Ver la historia de un archivo

Enviar el repositorio a un servidor

pwd : nos muestra el path o ruta de la carpeta en donde nos encontramos ubicados

cd : me permite acceder (entrar) a una carpeta en un nivel o varios niveles

cd .. : me permite salir de una carpeta en un nivel o varios niveles OJO los dos puntos deben ser separados por un espacio del comando cd

ls : me muestra los archivos que contiene una carpeta, puede ser la ubicación actual o una ruta especifica, no muestra los archivos ocultos

ls -a : me muestra los archivos que contiene una carpeta, puede ser la ubicación actual o una ruta especifica, incluyendo los archivos ocultos

ls -l : me lista los archivos que contiene una carpeta con sus atributos, puede ser la ubicación actual o una ruta especifica, no muestra los archivos ocultos

ls -la : me lista los archivos que contiene una carpeta con sus atributos, puede ser la ubicación actual o una ruta especifica, incluyendo los archivos ocultos

clear : limpiar la consola o terminal, o un shortcut ctrl + l

mkdir <nombre carpeta> : nos permite crear una carpeta

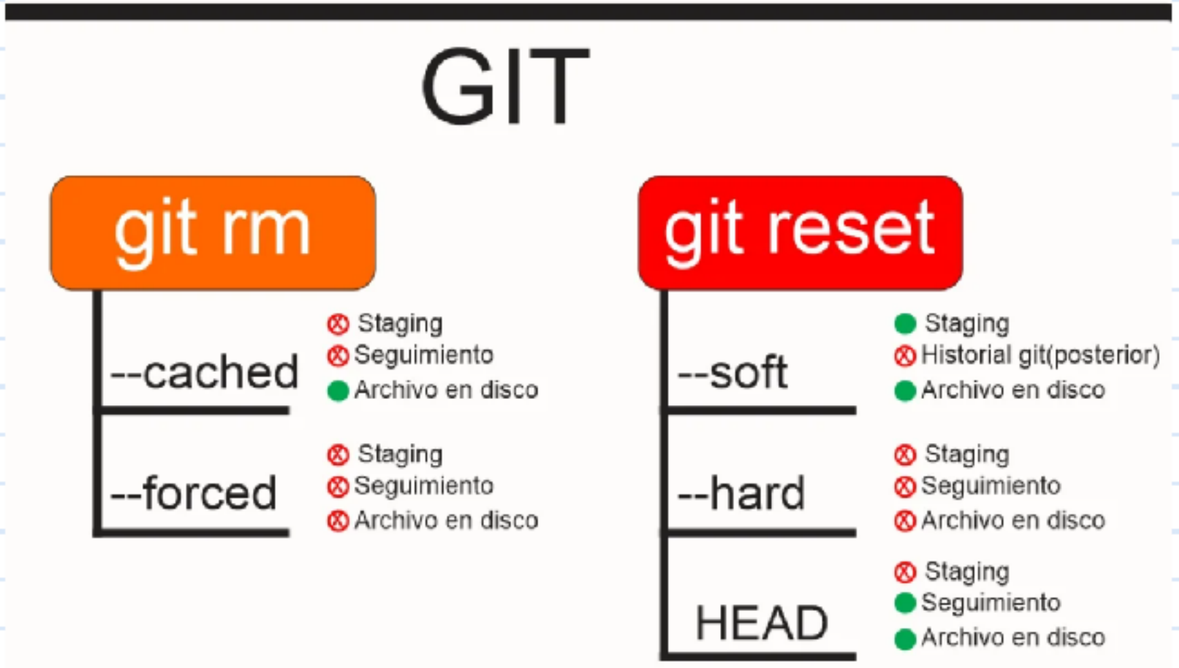
touch <nombre del archivo> : nos permite crear un archivo

cat <nombre del archivo> : me permite visualizar el contenido del un archivo y lo muestra en el terminal

history : nos muestra un historial de los comandos que hemos utilizado

rm <nombre del archivo> : me permite borrar un archivo

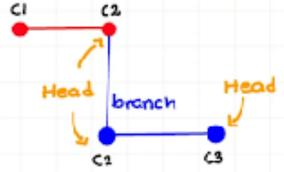
OJO en Windows el terminal no es case sensitive (Sensible las mayusculas), con Linux,y UNIX si son case sensitive



git init → Crea el staging y el repositorio remoto.
git add → Tracking de los cambios (staging).

git fetch + git merge = git pull.

git commit -a = commit + add.



Checkout Nombre branch
→ cambiar de rama a último commit.

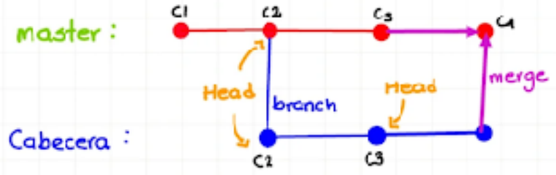
git branch nombre → Crear rama.

MERGE

- Un merge es un commit.
- Se debe hacer checkout a la rama a la que se quiere halar la rama.
- Si en las dos ramas se hicieron cambios sobre la misma línea, git lanza un error y no deja hacer el merge.

-> On branch master
git merche rama o unir → Trae a la rama master los cambios hechos en

- Con git log puedo ver que git merge trae a la base de datos los dos últimos commits de las dos ramas y su fusión (merge).



USO DE GITHUB

GitHub es una forja para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de ordenador. El software que opera GitHub fue escrito en Ruby on Rails.

Luego de crear nuestra cuenta en https://github.com podemos crear o importar repositorios, crear organizaciones y proyectos de trabajo, descubrir repositorios de otras personas, contribuir a esos proyectos, dar estrellas y muchas otras cosas.

- **git clone "URL"**

Comando para clonar un repositorio desde GitHub (o cualquier otro servidor remoto) debemos copiar la URL (por ahora, usando HTTPS) y ejecutar el comando git clone + la URL que acabamos de copiar.

Comandos útiles

`git fetch #actualizar mi copia local del repositorio remoto, no lo copia en el directorio local`

`git merge #para continuar los últimos cambios del repositorio remoto y nuestro directorio de trabajo`

`git pull origin master`

Comando para traer los cambios realizados en el repositorio de Github a nuestro repositorio local

README.md

Archivo que veremos por defecto al entrar a un repositorio. Sirve para describir el proyecto, los requerimientos y las instrucciones que debemos seguir para contribuir correctamente.

- **Conectar el repositorio de GitHub con nuestro repositorio local:**

1) Guardar la URL del repositorio de GitHub con el nombre origin

`git remote add origin "URL"`

2) Verificar que la URL se haya guardado correctamente:

`git remote`
`git remote -v`

*Si hacemos git push origin master, nos mostrará una adverencia debido a la diferencia de archivos de trabajo en ambos repositorios.

3) Debemos traer la versión del repositorio remoto y hacer merge para crear un commit con los archivos de ambas partes. Podemos usar git fetch y git merge o solo el git pull, pero para forzar se debe usar:

`git pull origin master --allow-unrelated-histories`

4) Por último, ahora sí podemos hacer git push para guardar los cambios de nuestro repositorio local en GitHub

`git push origin master`

GitHub conocido como la "red social de los programadores", es un super-servidor de Git que nos permite alojar nuestros proyectos de tal manera que cualquiera pueda acceder y colaborar de forma remota en el desarrollo de los mismos. De esta forma podemos tener nuestro portafolio de proyectos y dar a conocer que sabemos y que podemos hacer.

```
ssh-keygen -t rsa -b 4096 -C
"youremail@example.com"
```

Comprobar proceso y agregarlo (Windows)

- `eval $(ssh-agent - s)`
- `ssh-add ~/.ssh/id_rsa`

Comprobar proceso y agregarlo (Mac)

- `eval "$(ssh-agent -s)"`

¿Usas macOS Sierra 10.12.2 o superior?

Haz lo siguiente:

- `cd ~/.ssh`
- Crea un archivo config...
- Con Vim `vim config`
- Con VSCode `code config`
- Pega la siguiente configuración en el archivo...

```
Host *
  AddKeysToAgent yes
  UseKeychain yes
  IdentityFile ~/.ssh/id_rsa
```

FLUJO DE TRABAJO PROFESIONAL CON PULL REQUESTS

En un entorno profesional normalmente se bloquea la rama **master**, y para enviar código a dicha rama pasa por un **code review** y luego de su aprobación se unen códigos con los llamados **pull request**.

Los **pull request** no es una característica de Git, si no de GitHub.

Los **pull request** tambien son importantes porque permiten a personas que no son colaboradores, trabajar y apoyar en nuestro proyecto.

Equivalencia en otras plataformas:

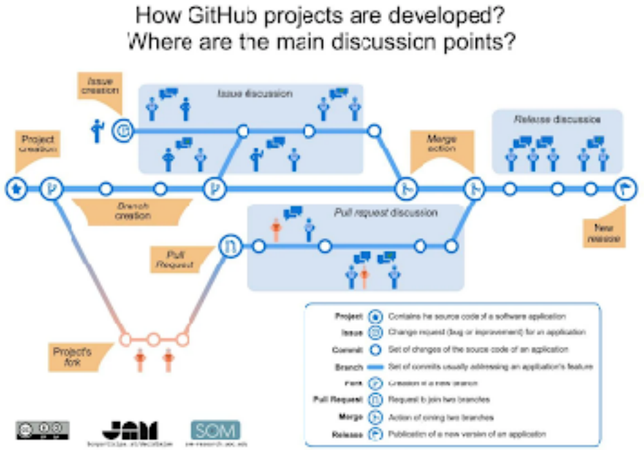
Bitbucket	GitHub	GitLab
Pull Request	Pull Request	Merge Request

De acuerdo a diversos estudios, resulta más barato encontrar y corregir incidencias en etapas tempranas del desarrollo que encontrarlas y corregirlas en producción. De hecho, algunos estudios señalan que es 10 veces más caro corregir por cada fase del proceso qué pasa.

Para realizar pruebas enviamos el código a servidores que normalmente los llamamos **staging server** , luego de que se realizan las pruebas pertinentes tanto de código como de la aplicación estos pasan a el **servidor de producción**.

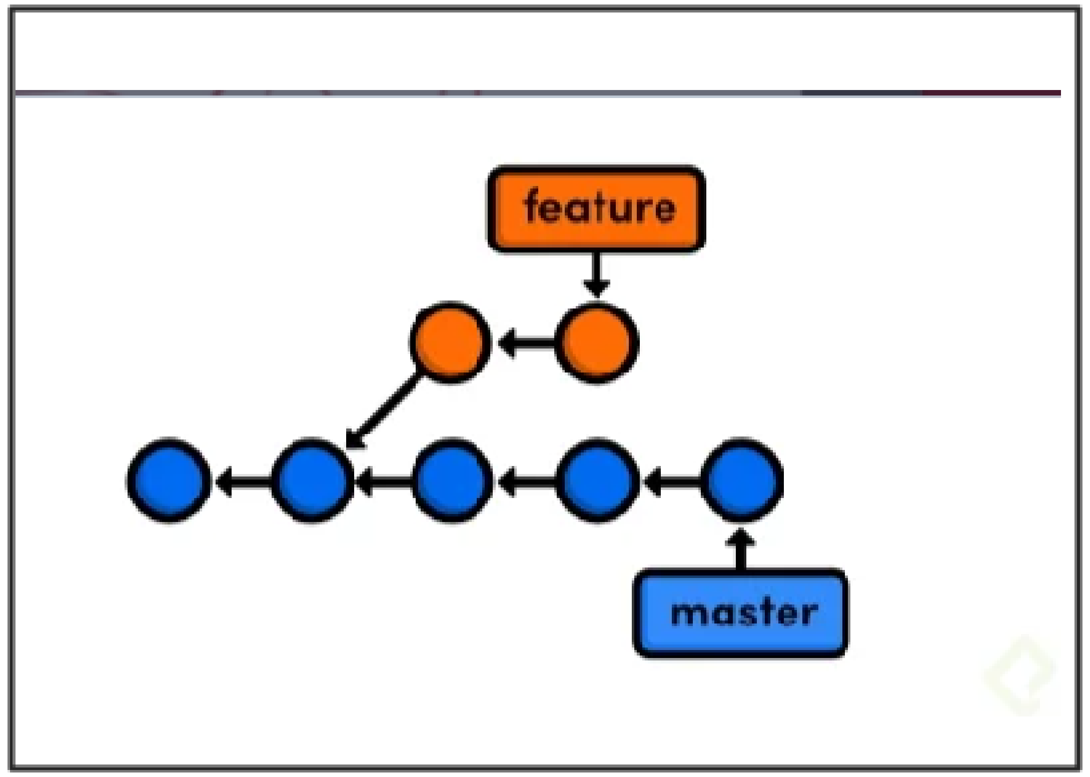
PULL REQUESTS

Es la acción de validar un código que se va a *mergear* de una rama a otra En este proceso de validación pueden entrar los factores que queramos: Builds (validaciones automáticas), asignación de código a tareas, validaciones manuales por parte del equipo, despliegues, etc.



La persona que hace todo esto, normalmente son los lideres de equipo o un perfil muy especial que se llama **DevOps** (permite que los roles que antes estaban aislados se coordinen y colaboren para producir productos mejores y más confiables).

Los pull request podrían compararse con un control de calidad interno donde el equipo tiene la oportunidad de detectar bugs o código que no sigue lineamientos, convenciones o buenas prácticas. Incluso puede presentar ahorros a la empresa. **Github** nos permite llevar un control e implementa un proceso para la atención y revisión de estas solicitudes.



GIT STASH: GUARDAR CAMBIOS EN MEMORIA Y RECUPERARLOS DESPUÉS

NOTA:

Por defecto git stash **NO** almacena los archivos no preparados (untracked o que no se hayan ejecutado con *git add*) y los archivos ignorados.

Si se necesita guardar en el stash estos archivos, ejecutar los siguientes comandos:

git stash -u #El stash considera los Untracked files

git stash -a #El stash considera los archivos ignorados

git stash save -u "mensaje" #Considera los Untracked y se agrega un comentario

Comandos Importantes

git stash save "mensaje" #Comando para comentar los stashes con una descripción

git stash apply #Aplicar los cambios en el código en el que estás trabajando y conservarlos en tu stash

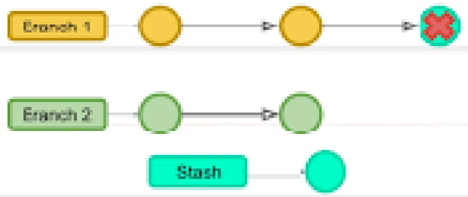
git stash show #Comando para visualizar un resumen de un stash

git stash show -p #Comando para ver todas las diferencias de un stash

git stash pop stash@{n} #Comando para elegir que número "n" de stash se desea volver a aplicar

git stash drop stash@{n} #Comando para eliminar un determinado número "n" de stash.

El comando **git stash** almacena temporalmente (o guarda en un stash) los cambios que hayas efectuado en el código en el que estás trabajando para que puedas trabajar en otra cosa y, más tarde, regresar y volver a aplicar los cambios.



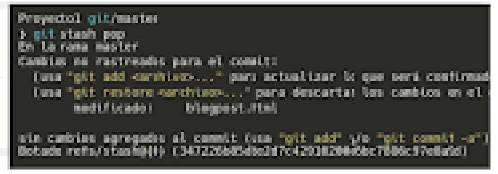
- **git stash**

El comando ***git stash*** coge los cambios sin confirmar (tanto los que están preparados como los que no lo están), los guarda aparte para usarlos más adelante y, acto seguido, los deshace en el código en el que estás trabajando.

```
Proyecto1 git/master*
> git stash
Directorio de trabajo guardado y estado de índice WIP on master: f43b9fa Experimento 2
```

- **git stash pop**

Puedes volver a aplicar los cambios de un stash mediante el comando ***git stash pop***. Al hacer *pop* del stash, se eliminan los cambios de este y se vuelven a aplicar en el código en el que estás trabajando. De forma predeterminada, ***git stash pop*** volverá a aplicar el último stash creado.



- **git stash branch nombre-rama**

Puedes usar ***git stash branch*** para crear una rama nueva basada en la confirmación a partir de la cual creaste el stash y, después, se hace pop en ella con los cambios del stash.

- **git stash list**

Se puede listar los stash realizados con el comando ***git stash list***. Este comando muestra el número de stash @{n}

```
stash@{0}: WIP on master: f43b9fa Experimento 2
(END)
```

- **git stash drop**

Si decides que ya no necesitas algún stash en particular, puedes eliminarlo mediante el comando ***git stash drop***.

```
Proyecto1 git/master
> git stash drop
Botado refs/stash@{0} (ce8c16a31a67d846d6741d9760fe622ccffc6164)
```

- **git stash clear**

Comando para eliminar todos los stashes.

```
Proyecto1 git/master
> git stash clear
```

Git stash es uno de los comandos más útiles de Git, es una forma rápida de tener en temporal tus cambios, poder moverte entre ramas y luego recuperar esos cambios. ***Git stash*** se recomienda cuando haces pequeños cambios que no merecen ramas o cuando llevas trabajo adelantado y necesitas datos de

GIT CLEAN: LIMPIAR TU PROYECTO DE ARCHIVOS NO DESEADOS

GIT CLEAN ACTÚA EN ARCHIVOS SIN SEGUIMIENTO.

Los archivos sin seguimiento son aquellos que se encuentran en el directorio del repositorio, pero que no se han añadido al índice del repositorio con *git add*.

Los archivos o carpetas sin seguimiento especificados como .gitignore no se eliminarán si ejecutamos ***git clean -f***

Comandos importantes

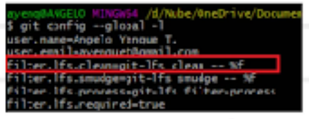
```
git clean -dn  #Comando para verificar que elementos se eliminan incluyendo directorios

git clean -df  #Comando para eliminar archivos incluyendo directorios

git clean -xdn  #Comando para verificar que elementos se eliminan incluyendo los archivos ignorados y directorios

git clean -xdf  #Comando para eliminar archivos incluyendo los archivos ignorados y directorios

git clean -q  #No muestra en pantalla solo los mensajes de errores, no imprime los nombres de los archivos eliminados

git clean -df  #Comando para eliminar solo los archivos ignorados incluidos en .gitignore

git clean -i  #Comando interactivo de git clean, para ejecutar el proceso por medio de opciones, se puede combinar con otros comandos.
```

Git Clean es un método adecuado para eliminar los archivos sin seguimiento en un directorio de trabajo del repositorio.

• git clean

*Si ejecutamos **git clean** por defecto, la consola mostrará un error.*

*Esto porque por defecto y por seguridad **Git Clean** esta configurado para ser ejecutado con el parámetro **-f** (force).*



• git clean --dry-run

*Si ejecutamos **git clean --dry-run** o **git clean -n** , Git realizará un simulacro de borrado y podrás ver los archivos que se van a eliminar sin que se eliminen realmente.*

• git clean -f

*Si ejecutamos **git clean -f** , Git inicia la eliminación real de los archivos sin seguimiento del directorio actual.*

• git clean -d

Ya que se ignora los directorios de forma predeterminada, podemos agregar la opción **-d** a **git clean**, para indicar a Git que también quieres eliminar los directorios sin seguimiento.

• git clean -x

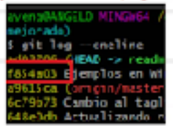
git clean -x indica a Git que incluya también los archivos **ignorados**. Al igual que ocurría con las anteriores invocaciones de git clean, se recomienda realizar un simulacro antes de la eliminación final. La opción **-x** actuará en los archivos ignorados.

Con ***git clean***, podemos eliminar archivos y/o carpetas, para tener una versión limpia de nuestro proyecto, debido a que en ocasiones tenemos elementos que han sido generados por herramientas de combinación o externas, teniendo siempre cuidado de eliminar algo por error.

GIT CHERRY-PICK: TRAER COMMITS VIEJOS AL HEAD DE UN BRANCH

git cherry-pick es una herramienta útil, pero **no siempre es una práctica recomendada**, ya que puede generar duplicaciones de confirmaciones

```
git log --oneline
#Comando para buscar rápidamente el Hash que necesitamos par a ejecutar cherry-pick
```



Comandos importantes

```
git cherry-pick HASH -e
#Comando para ejecutar cherry-pick pero permite editar el mensaje del commit original

git cherry-pick HASH -n
#Comando para ejecutar cherry-pick pero solo trae los cambios y no ejecuta un commit
```

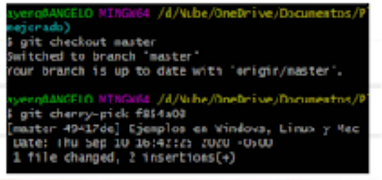
• GIT CHERRY-PICK [HASH]

Este comando permite elegir una confirmación de una rama y aplicarla a otra. Permite que las confirmaciones arbitrarias de Git se elijan por referencia y se añadan al actual HEAD de trabajo.

Para usar cherry-pick lo único que necesitamos saber es el commit específico que queremos aplicar en nuestra rama.

Ejemplo:

Queremos traer el commit **f8854a03** de la rama "readme-mejorado". Nos ubicamos en la rama donde vamos a llevar esa confirmación, en este caso, **master** y ejecutamos.



```
git cherry-pick f8854a03
```

Ahora si revisamos el log de la rama **master**, verificamos que se ha creado el mismo commit y que se aplicaron todos los cambios como si lo hubiéramos realizado en el mismo **master**.

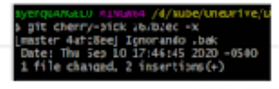
Hay que tener en cuenta que este comando crea un nuevo commit, por lo que antes de usar **cherry-pick** no debemos de tener ningún archivo modificado que no haya sido incluido en un ***commit***.

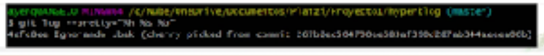
También, en caso de querer aplicar más de un commit, nos basta con indicárselos a cherry-pick

```
git cherry-pick [HASH1] [HASH2]
```

Podemos usar la opción **-x** en caso deseamos añadir una referencia al commit original

```
git cherry-pick [HASH] -x
```





El comando ***cherry-pick*** es útil en algunos casos, por ejemplo, cuando se necesita corregir algún error explícito. Permite ejecutar un commit puntual de una rama en otra pero no se debe aplicar equivocadamente en lugar de merge o rebase.

TÉRMINOS DE git QUE DEBES CONOCER

REPOSITORY/REPO

Base de datos donde se almacena el historial del código.

BRANCH / RAMA

Entorno o espacio de **trabajo independiente en Git.**

COMMIT

Registro de uno o varios cambios hechos en el repositorio.

MASTER

Rama principal de un repositorio.

STAGE

Lista de archivos que se usarán para un commit.

CHECKOUT

Es la acción de **moverse entre diferentes ramas.**

FORK

Copia de un repositorio.

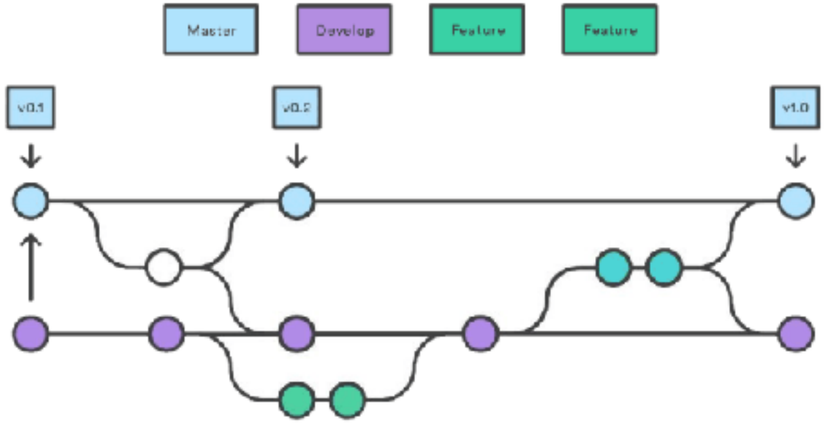
MERGE

Combina dos o más ramas en una.

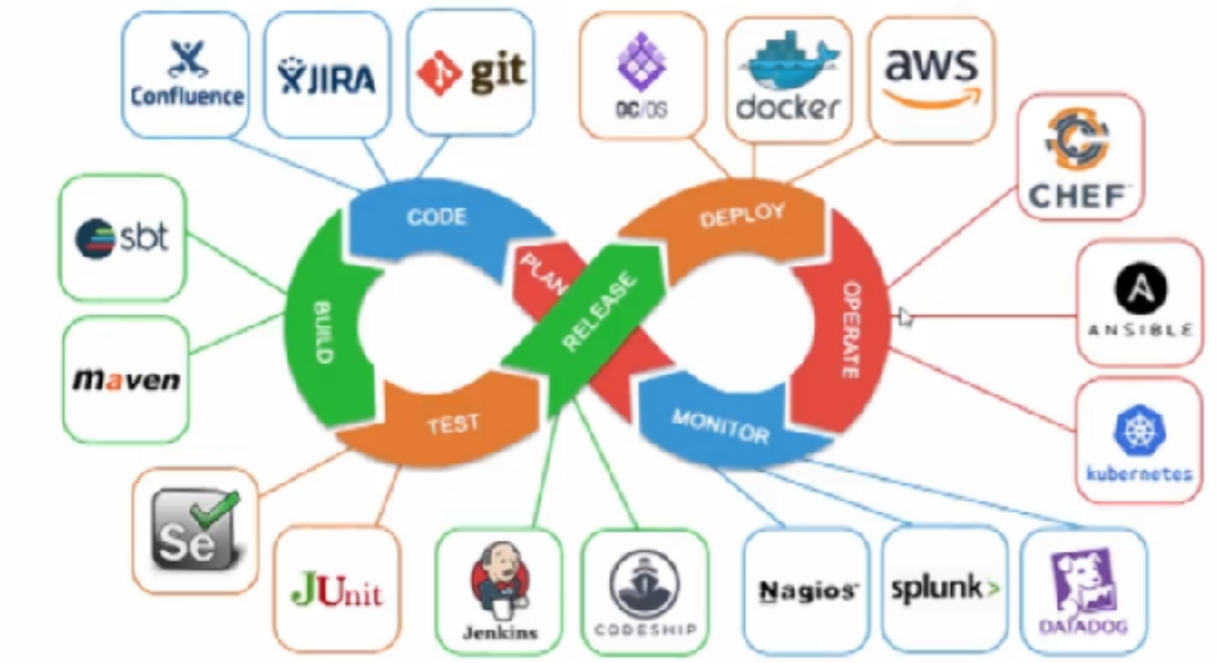
Si eres programador es **obligatorio** aprender GIT. Domínalo en:

ed.team/cursos/git

EDteam



ciclo de vida del desarrollo de software



Mientras tanto en git

-Yo haciendo hard reset al primer commit

- El equipo de trabajo

Git y GitHub

* git init

Estas en la carpeta en la que están los archivos de tu proyecto.
Abres la terminal y ejecutas git init para indicar que esta es la carpeta central de mis archivos.
Cuando ejecutamos el comando, ademas, creamos:

a. Staging, es el area en la memoria RAM que es donde al principio iremos agregando los cambios.
b. Repository (repositorio/master): Es el directorio que conocemos como /.git/ que es el lugar donde al final del proyecto quedarán todos los cambios.

* git config --global user.name "MiNombre"

-Configuramos el git a nuestro nombre.

* git config --global user.email "Tu@correo.com"

-Configuramos git con nuestro correo.

* git config -l

-Nos muestra la configuración que tenemos en nurstro git.

* git config --global alias.nombre_alias "shortlog -sn --all --no-merges"

-Le asignamos al alias **nombre_alias** el comando indicado entre las comillas

* git add archivo.txt

- Agregamos el archivo al Staging, en este punto el archivo espera a que lo agreguemos al repositorio.
Nota: antes de usar este comando, el archivo lo llamamos como un archivo untracked", luego de usar el comando el archivo lo podemos llamar un archivo Tracked, cuando está ya en el Staging

- git add .

- Agregamos TODOS los archivos dentro de nuestra carpeta al Staging.

* git rm –cached archivo.txt

-Eliminamos el archivo del Staging area y lo volvemos Untracked.

* git commit -m "mensaje"

Luego de escribir un mensaje descriptivo, mandamos el archivo al repositorio/master.
Nota: Cada commit es una nueva versión con cambios de tu archivo.

* git commit -am "mensaje"

- Agregamos el archivo al Staging, y hacemos el commit de una vez.
Nota: *Solo funciona con archivos a los que ya le habíamos hecho git add anteriormente.*

* git status

-Nos muestra cual es el status de nuestro proyecto.

COMANDOS Y RECURSOS COLABORATIVOS EN GIT Y GITHUB

Comandos utiles

```
git comando --help
#La opcion --help de cualquier comando de Git ejecuta un documento donde se puede leer y revisar todo el funcionamiento "por dentro" de dicho comando.
```

```
git blame -e "archivo"
#La opción -e muestra la dirección de correo electrónico de los autores en lugar de su nombre de usuario.
```

```
git blame -e "archivo" #Ejecuta el comando blame pero "idenfa" cada linea del archivo para mostrarlo más ordenado.
```

```
git blame -M "archivo"
#Detecta las líneas que se han movido o copiado dentro del mismo archivo. Esto indicará el autor original de las líneas en lugar de indicar el último autor que las ha movido o copiado.
```

GitHub Insights proporciona métricas e informes analíticos para ayudar a los equipos de ingeniería a comprender y mejorar su proceso de entrega de software.

- Pulse
- Contributors
- Community
- Traffic
- Comnits
- Code frequency
- Dependency graph
- Network
- Forks

El propósito de cualquier sistema de control de versiones es registrar cambios en su código. Esto te da el poder de volver a la historia de tu proyecto para ver quién contribuyó con qué, averiguar dónde se introdujeron los errores. Pero, tener toda esta historia disponible es inútil si no sabes cómo navegar por ella. Git ofrece herramientas colaborativas y de inspección como git ***shortlog***, ***blame***, etc.

• GIT SHORTLOG

Agrupa cada confirmación por autor y muestra la primera línea de cada mensaje de confirmación. Es una forma fácil de ver quién ha estado trabajando en qué. Tiene opciones que se pueden ejecutar combinadas:

`git shortlog -n`
Ordena la salida según el número de confirmaciones por autor en lugar de hacerlo por orden alfabético.

`git shortlog -s`
Proporciona sólo un resumen del recuento de la confirmación.

`git shortlog -e`
Muestra la dirección de correo electrónico de cada autor.

`git shortlog --all --merges`
Muestra solo las confirmaciones de fusión o merge.

`git shortlog --all --no-merges`
Muestra todas las confirmaciones y excluye las que son por merge o fusión.

• GIT "ALIAS"

Si no quieras teclear el nombre completo de cada comando de Git, puedes establecer fácilmente un alias para cada comando mediante ***git config***.

git config --global alias.alias-que-quiero-usar 'comando-a-ejecutar'

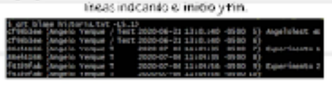
• GIT BLAME

El funcionamiento de ***git blame*** es la visualización de metadatos de autor adjuntos a líneas específicas confirmadas en un archivo. Esto se usa para examinar puntos específicos del historial de un archivo y poner en contexto quién fue el último autor que modificó la línea.

git blame "nombreArchivo"

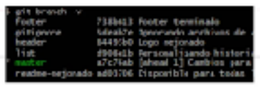
git blame "nombreArchivo" L,[linea_isicio],[linea_fin)

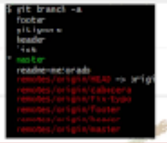
Agregando la opción **L**, permite filtrar por una cantidad de líneas indicando el inicio y fin.



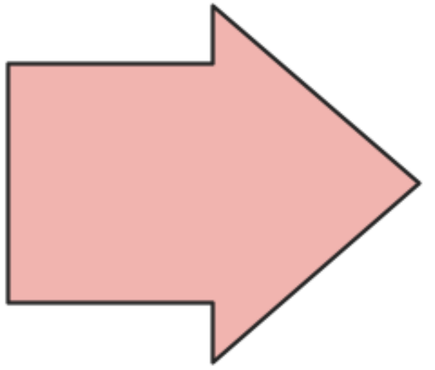
• GIT BRANCH

El comando ***git branch*** tiene opciones adicionales para listar las ramas existentes en nuestro repositorio. Con la opción **-v**, lista las ramas locales. La opción **-r** lista las ramas remotas, permite comparar que ramas se han enviado a Github. Con la opción **-a** puedes listar todas las ramas.





git NUNCA PARES DE APRENDER github



Fin de los apuntes, muchas gracias :)