# **GitHub**

# **Como iniciar empezar a usar el GitHub**

1. Crear una cuenta en GitHub.
2. En la esquina superior derecha, junto a tu avatar o ícono de identidad, haz clic + en y luego seleccione Nuevo repositorio.
3. Nombra tu repositorio con el nombre que quieras
4. Escribe una breve descripción.
5. Seleccione Inicializar este repositorio con un archivo README.
6. Abrir el commander apretando click derecho en la carpeta donde se desea realizar el trabajo y apretar la opción de git bash, lo cual abrirá un commander

# **Comandos básicos propios del commander**

cd (change directory): cambia de directorio poniendo por ejemplo cd Desktop. Para ir al directorio de atrás poner cd ..

ls(list): lista el contenido del directorio.

ls –la: también lista los archivos ocultos

Pwd (print working directory): me muestra el directorio actual.

Clear: limpia la pantalla.

# **Hay tres lugares donde van a ir pasando**

# **nuestros archivos dentro de Git**

|  |  |  |
| --- | --- | --- |
| Working Directory (wd) | Stage Area | Repositorio(index->carpeta.git) |
| Es nuestra carpeta de desarrollo donde tengo mis archivos  (Color Rojo) | Trae como una captura de pantalla al archivo  (Es lugar imaginario)  (color verde)  Para poder pasar al stage area se realiza un git add del archivo que se encuentra en el wd | Se confirma el cambio y pasa acá con el git commit, creando nuevas versiones y conservando las anteriores |

Sistema de graph (grafo) diagrama con ramificaciones

# 

Los git tienen 4 tipos de objetos:

* Commit (confirmación).
* Tree (árbol): es como si fuera la carpeta.
* Blob (gota): es el archivo. Tiene identificación hash.
* Tag (etiqueta): sirve para poner la versión en el que se encuentra por ejemplo V 1.0.

# **Creo repositorio en git**

* git init
* git add README.md(esto dsp de haber creado el readme en el vsc)
* git commit -m "first commit"
* git remote add origin https://github.com/14Tincho/Angular-yt.git
* git push -u origin master

# **Comandos de Git**

* git init: inicia un repositorio dentro del directorio actual. (el repositorio se debe crear manualmente desde GitHub). Se crea una carpeta escondida. Git.
* git status: me indica que está pasando en mi carpeta.
* git status -s: muestra UU “archivo.txt” si hay unión con conflicto.
* git add archivo.txt: pone el archivo en el stage area sirve también para crear el archivo si no lo tengo en wd.
* git add .: agrego todos los archivos juntos.
* git config: un help de todos los comandos config.
* git config - -list: lista mi configuración.
* git config - -local: Para definir mi nombre, mi usuario, mi mail, etc. (puede ser - -global también).

Por ejemplo:

git config - -local user.name “Lionel Messi”

git config - -local user. email [lionel\_messi@gmail.com](mailto:lionel_messi@gmail.com).

* git commit –m “mensaje”: pasa el archivo del stage area al repositorio. Me genera la carpeta objects en la carpeta. Git en caso de ser la primera vez y después todos los archivos commiteados pasan a esa carpeta con un hash
* git commit –am “mensaje”: -am significa que realiza simultáneamente el add y el commit siempre y cuando ya haya hecho al menos una vez un git add.
* git log: me trae todos los commits que tengo, me indica donde estoy parado dentro del árbol (en que rama con el HEAD) tira mucha información, te da el hash.
* git log - -oneline - -graph - -all -5: te trae una línea de información por cada archivo, muestra el grafico del tree, y con el all te muestra la rama completa, sin el all te muestra solo el actual, y el numero indica la cantidad de líneas que quiero que me muestre (las primeras líneas).
* git log - -author: te muestra el autor de donde estás parado.
* git cat –file –t hash: para inspeccionar tipos de objetos git.
* git cat –file –p hash: print del contenido.
* git cat –file –s hash: cuánto pesa el archivo.
* git diff: para ver una lista de conflictos por ejemplo si modifiqué un txt y no le hice el git add, o cuando hay branch(rama) paralelos con actualizaciones del mismo archivo y en la misma línea.
* git checkout master: es para posicionarme en una determinada rama, puede ser el master o si es otra rama pongo el hash o el nombre de la rama
* git checkout –b “f1”: crea un branch si no está creado y te posiciona en el mismo.
* git - -help: me trae todos los comandos permitidos del git.
* git rm - -cached(remove): borra el archivo de tu área de trabajo y lo deja en el stage área
* git rm: eliminar el archivo de tu directorio de trabajo, y además lo añade al área de staged files. La diferencia entre este comando y eliminar directamente el archivo, es que si sólo lo eliminas aparecerá en el área de unstaged files.
* git rm –f: Si ya habías modificado el archivo y lo tenías en el área de staged files, deberás forzar su eliminación con la opción -f. Si ya está en el commit no podrá ser recuperado.
* git rm –r: Idem al –f pero para carpetas.
* git rm - -cached archivo.txt: Es para mantener el archivo en tu directorio de trabajo, pero interrumpir su seguimiento por parte de Git.
* git reset: Lleva el archivo a una versión anterior.
* git remote: te muestra los repositorios ya existentes.
* git remote add origin url: para crear el repositorio llamado origin en la url definida. Puedo tener varios repositorios remotos asociados a uno solo local
* git remote –v: tira todos los alias con su push y fetch y con sus respectivas url.
* git remote –h: seria el help de los remotes.
* git remote set-url origin url: para cambiar la url del alias origin.
* git push origin master: envía al github el branch que le menciono (puede ser cualquier branch).
* git push –u origin master: es para asociar el nombre del alias y el branch, entonces las proximas veces solo pongo git push y me lo asocial.
* git push origin - -tag: lo pusheo en el github y la gente puede bajar la versión que desee.
* git clon url.git nombreClon: si me pasan un repositorio en el github lo descarga en mi pc.
* git merge f1:  fusionará varias secuencias de confirmaciones en un historial unificado. (para esto debes estar ubicado en el branch deseado) (f1 sería un branch el cual voy a unificar al branch “deseado”).
* git branch –r: muestra los branch remotos según donde me posicione.
* git branch –a: muestra todos los branches remotos y locales según donde me posicione.
* git branch –d “f1”: elimina la rama (no estar parado en el branch a deletear).
* git branch –D “f1”: fuerza el eliminado de la rama.
* git rebase “f1”: cambia la base de donde hago el merge.
* git rebase –i hash: (interactive) hace el rebase interactivo hasta el hash que yo le indique, me muestra todos los commits con la palabra pick adelante. Aprieto la i hasta ver el INSERT abajo, me posiciona el cursor arriba para poder escribir donde están todos los picks. Si quiero unir los commits, reemplazo los pick por squash, esto me une cada commit con el anterior. Cuando termino de poner todos los squash, aprieto el esc , me posiciona abajo el cursor y ahí abajo escribo, :wq luego enter y me guarda los cambios.
* git rebase --continue; continua los rebase que no se hayan completado
* git rebase - -abort: aborta los rebase que no se hayan completado
* git rebase - -skip: saltea/ignora los rebase que no se hayan completado
* git tag: sirve para marcar un commit, pushearlos y dejarle un mensaje. Pueden ser anotados y no anotados (estos últimos son menos útiles)
* git tag –a versión –m “mensaje”: el –a quiere decir que es anotado, versión puede ser V1.0, el mensaje puede ser el que queramos.
* git fetch: le pide al remoto que descargue los objetos nuevos. (sólo los descarga sin cambiar el puntero). Si hay más commit debo hacer merge.
* git pull: descarga y me hace simultáneamente el fetch y el merge.
* git revert hash: cuando ya esta pusheado, deshace todo lo que paso en el commit. No borra el historial, sino que genera un nuevo commit donde no tiene las ultimas modificaciones, y luego pusheo. Si me arrepiento del revert puedo revertirlo
* git submodule: (no es muy usado) sirve para tener un repositorio dentro de otro
* git reset: cuando aún no está pusheado vuelve el branch y el head a un commit anterior.
* git reset - -soft hash: Si hago un log no me figuran pero si hago un status si, eso es por si quiero volver a hacer los commits. Quedan en el stage área.
* git reset - -mixed hash: parecido al - -soft pero me lo deja en el work directory. Si quiero conservarlo debo hacer add y commit.
* git reset - -hard hash: no conserva nada a la vista. Pero lo puedo recuperar con el git reflog
* git reflog: me muestra todas las cosas que fueron pasando, cada una tiene un hash y lo pone así HEAD@{2} por ejemplo y luego para recuperarlo hago lo siguiente:
  + git checkout HEAD@{2}
  + git branch “f1backup”
  + git branch –d f1(el viejo)
  + git branch –m f1 (lo renombra)
  + git push origin f1.

# **Branch**

## Árbol lineal

Es una segmentación de mi repositorio, son las distintas etapas de mi repositorio. La suma de commits me forma por ejemplo el branch master. Por defecto arrancamos con branch master.

Si me piden modificar algo tomo el ultimo commit (que ya funciona y esta testeado), creo otro branch y trabajo tranquilo, si no funcionan los cambios no pasa nada porque no toque el original.

commit

Branch

Con el git branch “f1”, lo creo en donde estoy posicionado por ejemplo master, pero sigo en el master. Debo hacer git checkout “f1” para posicionarme en el nuevo branch, para poder trabajar. (f1 es el nombre del nuevo branch)

Head viejo

(HEAD 🡪 f1)

Head nuevo

Modifico o agrego archivos en el nuevo branch

Puedo pushear este nuevo branch a github, git push –u origin f1

Si quiero agregar estos cambios al master hago git checkout master y git merge f1. Me manda mensaje fast forward (la actualización es inmediata, sin análisis). Ahora subo los cambios del master con git push. Me figura que no subió nada, pero es porque ya estaban subidos en el f1, solo cambia el puntero al master.

master

## f1

## Árbol bifurcado

Master

F1

Cuando voy a hacer el merge del f1, al haber habido modificaciones en el master u otra branch que haya actualizado el master, el merge es recursive.

Master

Dev

F11

F12

Hago el merge del f11 y el f12 situado en el dev y son los dos fast forward porque no están pasándose los dos al master

## Conflictos

Cuando se actualizan más de un branch que modificaron en la misma línea del mismo archivo, genera un conflicto y nos muestra en el archivo en VSC la posibilidad de elegir qué cambio queremos adoptar de todos. Luego de esta elección debemos hacer un commit para que tome el cambio, pero no otro merge

# **Tags**

## No anotado

No son muy utilizados porque no nos aportan mucha información

## Anotados

Sirven para marcar un commit, dejarle un mensaje y puedo pushearlo (git push origin - -tags). En GitHub aparecen en la orejita tags y la gente puede bajar las cosas desde la versión elegida, por lo general a los tags se les pone V1.0, etc.

Por ejemplo:

git tag –a V1.0 –m “mensaje”

Donde menos a quiere decir que es anotado y V1.0 es la versión de modificaciones

# **Git flow**

* Git flow: Es la estructura estándar del git.

MASTER 🡪 DEV 🡪 f1, f2, etc

Master es el de producción, el definitivo ya vigente.

Dev (developer) es el de desarrollo, siempre sale del master.

f1, f2, etc. Son las funcionalidades que hacen los programadores.

Una vez terminado el f1 por ejemplo es pasado al dev y luego al master cuando se tome que ya el archivo es definitivo y se coordine con los demás subirlo al master.

# **Archivos especiales**

.gitignore: esto sería un archivo creado en el vsc el cual hace que ignores las carpetas o archivos que desees con solo poner el nombre de la carpeta o del archivo que queremos ignorar. El archivo .gitignore se tiene que llamar asi obligatoriamente para que funcione y se debe añadir, commitear y pushear. Si hago un git status no me muestra lo que hay adentro del .gitignore.

Por ejemplo:

.gitignore

archivo-f12.txt

carpeta-pruebas

README.md: este es otro archivo el cual se crea en el vsc y debe ser llamado así obligatoriamente para que funcione correctamente. También debe ser añadido, commiteado y pusheado. En el contenido pongo cualquier aclaración que pueda servir para el proyecto, tanto para mí como para mis compañeros