GIT Y GITHUB

1. QUE ES GIT

Git es un sistema de control de versiones distribuido, gratuito y de código abierto, que se utiliza para realizar un seguimiento de los cambios en el código fuente de un proyecto a lo largo del tiempo.

Permite a los desarrolladores trabajar colaborativamente, controlar la evolución de un proyecto, revertir a versiones anteriores y gestionar la sincronización de cambios entre diferentes equipos.

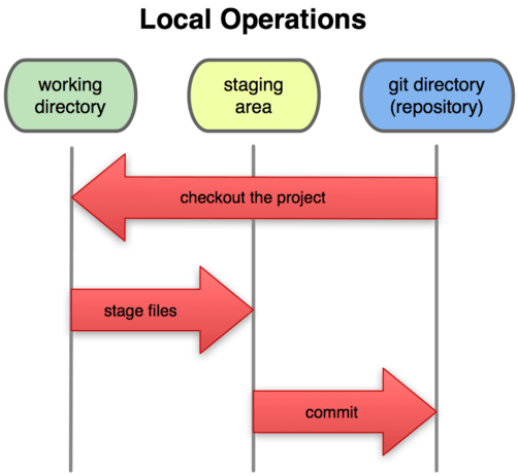
Características principales de Git

- **Distribuido:** Cada desarrollador tiene una copia completa del repositorio en su máquina local, lo que permite trabajar sin necesidad de conexión a internet y facilita la colaboración.
- **Historial de cambios:** Git registra cada modificación en el código, creando un historial detallado que permite ver quién hizo qué y cuándo.
- **Ramas:** Git permite crear ramas para desarrollar nuevas funcionalidades o arreglar errores sin afectar la rama principal, lo que facilita el desarrollo paralelo y la gestión de cambios.
- **Fusión de ramas:** Git permite combinar las ramas, integrando los cambios de una rama a otra de forma eficiente.
- **Confirmaciones (commits):** Git guarda instantáneas del código en diferentes momentos, lo que permite ver el historial de cambios y revertir a versiones anteriores.
- **Control de versiones:** Git permite volver a versiones anteriores del código, revertir cambios o comparar diferencias entre versiones.
- **Colaboración:** Git facilita el trabajo en equipo, permitiendo que varios desarrolladores trabajen en el mismo proyecto al mismo tiempo.

Que es un repositorio: es un almacenamiento centralizado donde se guarda el código fuente, documentos y recursos de un proyecto. Permite a los desarrolladores trabajar de forma colaborativa, rastrear cambios, revertir versiones y gestionar el flujo de trabajo.

Sistema de control de versiones (VCS): es una herramienta de software que realiza un seguimiento de los cambios en un archivo o conjunto de archivos a lo largo del tiempo, permitiendo a los usuarios volver a versiones anteriores, comparar cambios y colaborar de forma eficiente en proyectos.

El flujo de GIT funciona de la siguiente manera:



1. Working Directory

Es nuestra estación de trabajo local, aquí es donde podemos hacer cualquier cambio y no afectar nuestro repositorio en lo absoluto.

En cuanto modificamos algo de nuestro código, éste tiene status de **modificado**.

Si ejecutamos el comando **git status,** nos mostrará qué archivos han sido modificados (o creados).

2. Staging Area

Una vez que hemos hecho los cambios necesarios, pasamos nuestros archivos al "**staging area**" o área de preparación con el comando:

```
git add nombreDelArchivoModificado.js
```

nota: solo si queremos guardar un archivo especifico le damos el nombre del archivo y listo

o si queremos que guarde todos los cambios escribimos el siguiente comando

```
git add .
```

y con esto, agregamos todos los archivos modificados dentro de **working directory** a **staging area**.

Cuando pasas el código de Working Directory a Stagin Area, cambias el estado del código de **modificado** a **preparado**.

3. Git Repository local

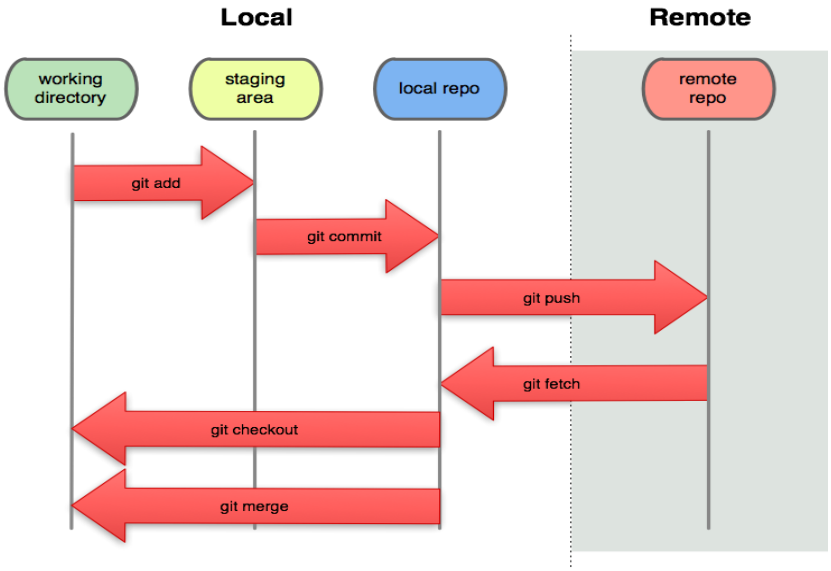
Una vez estando agregado en el área de preparación, aquí es donde le podemos dar nombre a nuestra nueva versión. Y crear una "copia" de cómo quedaría nuestro repositorio en producción.

Para pasar nuestro código de *staging area* al Git Repository (aún no se publica el código en la nube Github→ojo), escribimos el siguiente comando:

```
git commit -m "Nombre del la nueva versión"
```

esto lo que hace confirma los cambios y los guarda en un repositorio local

Nota: cuando haces el commit el código pasa de estado **preparado** a **confirmado**.



Notas:

Rama master/main: es la rama principal de tu proyecto

HEAD: es la versión mas reciente de tu proyecto o rama

Registrar nuevo usuario asociado a git usamos los siguientes comandos:

1. **Colocamos un nombre de usuario**

   git config --global user.name "mi nombre"

nota: No colocar como nombre de usuario el correo de su cuenta de Github, podría traer problemas a futuro.

2. **Configuramos el correo que estará asociado nuestro git**

   ```
   git config --global user.email "myemail@example.com"
   ```

nota: es recomendable utilizar el correo asociado a Github

3. **Iniciamos nuestro primer repositorio**

   ```
   git init

   // Iniciar un nuevo repositorio
   // Crear la carpeta oculta .git
   ```

4. **Agregar los cambios realizados en nuestro directorio de trabajo a área de staging (puesta en escena) o preparación**

   git add .

   // Con el punto agregar todos los archivos modificados.

5. **Pasar lo agregado en el área de preparación (STAGING AREA) a confirmarlo en nuestro repositorio local con:**

   git commit -m "primer commit"

   // Crear commit, es decir una confirmación de los cambios realizados (fotografía del proyecto en ese momento) y los guarda en un almacenamiento local (repositorio local)

Nota: para poder inserta y modificar una información por ejemplo un **commit sin nombre**, si queremos editarlo, utilización las teclas (**ESC + I**) y nos dejara modificarlas; ahora para guardar esa inserción de modificaciones, utilizamos las teclas (**ESC+ SHIT+Z+Z**) **y se guardaran esos cambios y también nos permite salir y volver a la consola**

Otros comandos:

Git reset : usarlo con mucho cuidado porque de pendiendo el uso borra los archivos. permite deshacer cambios en el historial y el área de staging de un proyecto. Es una herramienta potente que puede modificar el estado de la rama y los archivos, pero también puede causar problemas si no se usa correctamente. Es crucial entender los diferentes modos de git reset para evitar la pérdida de datos.

- git reset --soft:

Mueve el HEAD a una confirmación anterior, manteniendo los cambios en el directorio de trabajo y en el área de staging. Es útil si quieres deshacer una confirmación y luego volver a editar y confirmar los cambios.

- git reset --mixed (predeterminado):

Mueve el HEAD a una confirmación anterior, manteniendo los cambios en el directorio de trabajo pero removiendo los cambios del área de staging. Es útil si quieres deshacer una confirmación pero no descartar los cambios que estaban preparados para ser confirmados.

- git reset --hard:

Mueve el HEAD a una confirmación anterior y elimina los cambios del directorio de trabajo y del área de staging. Es útil si quieres deshacer una confirmación y descartar todos los cambios que se habían realizado desde esa confirmación.

- Entender el efecto de git reset:

Antes de usar git reset, es importante entender el efecto que tendrá en el directorio de trabajo y en el área de staging.

Nota: No usar git reset --hard en ramas públicas: Si se usa git reset --hard en una rama pública, se reescribe el historial y puede causar problemas a otros desarrolladores.

- Uso de git revert:

Si se necesita deshacer una confirmación en una rama pública, es más seguro usar git revert, que crea una nueva confirmación que deshace los cambios de la confirmación anterior, en lugar de reescribir el historial.

Ejemplos de utilización de reset:

- git reset --soft HEAD~1

Mueve el HEAD a la confirmación anterior (la confirmación padre de la actual) y mantiene los cambios en el directorio de trabajo y en el área de staging.

- git reset --mixed HEAD~1

Mueve el HEAD a la confirmación anterior, mantiene los cambios en el directorio de trabajo, pero los desorganiza del área de staging.

- git reset --hard HEAD~1

Mueve el HEAD a la confirmación anterior y descarta todos los cambios del directorio de trabajo y del área de staging.

- git reset <archivo>

Desorganiza un archivo específico del área de staging, manteniendo los cambios en el directorio de trabajo.

- git reset .

Desorganiza todos los archivos del área de staging, manteniendo los cambios en el directorio de trabajo.

RAMAS.

Crear ramas

Git branch: nos permite crear una nueva rama la cual será una copiada del head principal y a partir de allí se creará una nueva rama.

Además, nos permite saber en qué rama estamos

Ejemplo: git Branch nombredelarama

```
YETREN@LAPTOP-384VTT80 MINGW64 ~/Documents/PROGRAMACION/nav-bar (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean

YETREN@LAPTOP-384VTT80 MINGW64 ~/Documents/PROGRAMACION/nav-bar (main)
$ git branch
* main
```

Navegar entre ramas y versiones

Git checkout: sirve para moverse entre las versiones y las ramas

Solo necesitas saber el código de la versión o la rama

Y si necesitas regresar a la última versión debes colocar **main**

Ejemplo: git checkout main // permite ir a la última versión de la rama

GIT MERGE:

Sirve para para unir ramas, esta se une dependiendo desde donde queremos unirlas, es decir, es importante donde estemos parados porque esta definirá la unión. **Ejemplo:** para unir las versiones a la rama principal invocamos el **comando + el nombre de la rama que deseamos unir** y se creara una nueva versión de unión de ramas **(merge)**; cabe destacar el punto de versiones donde nos encontremos también influye en esa unión es recomendable hacerlo desde los **head---> master/main (la última versión)**

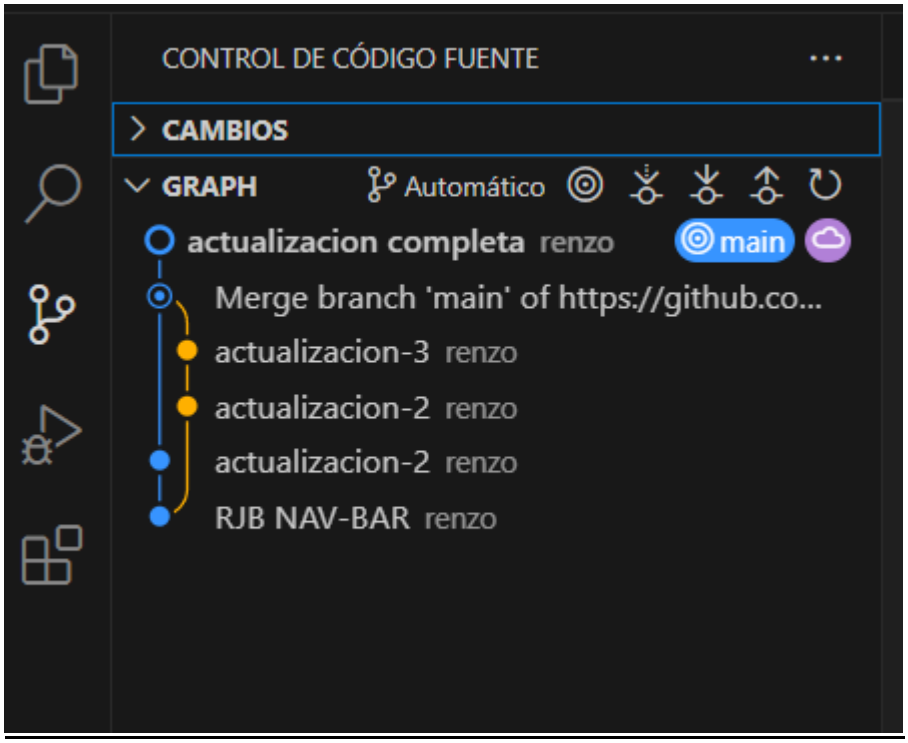


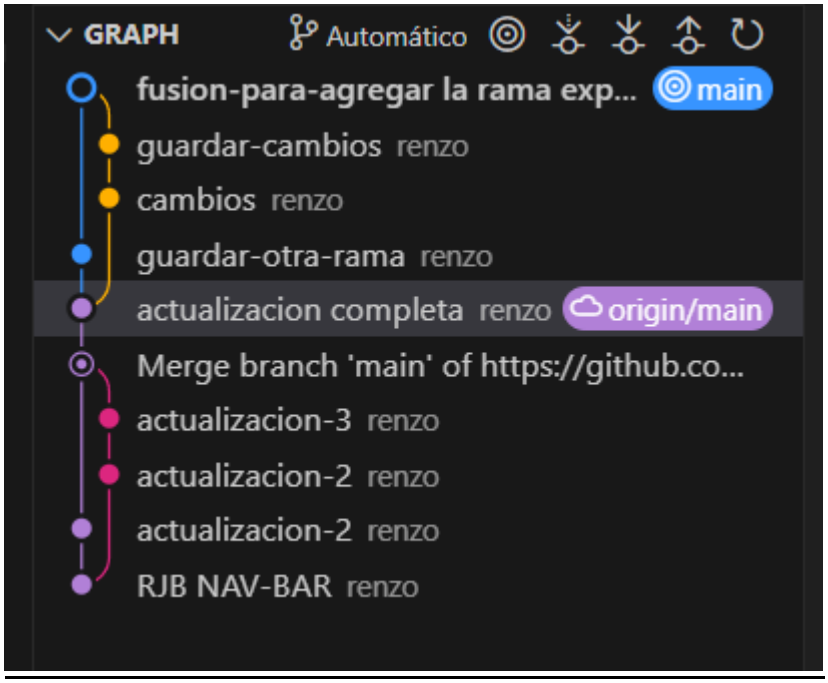
IMAGEN DE UNA RAMA Y COMIIT -AM

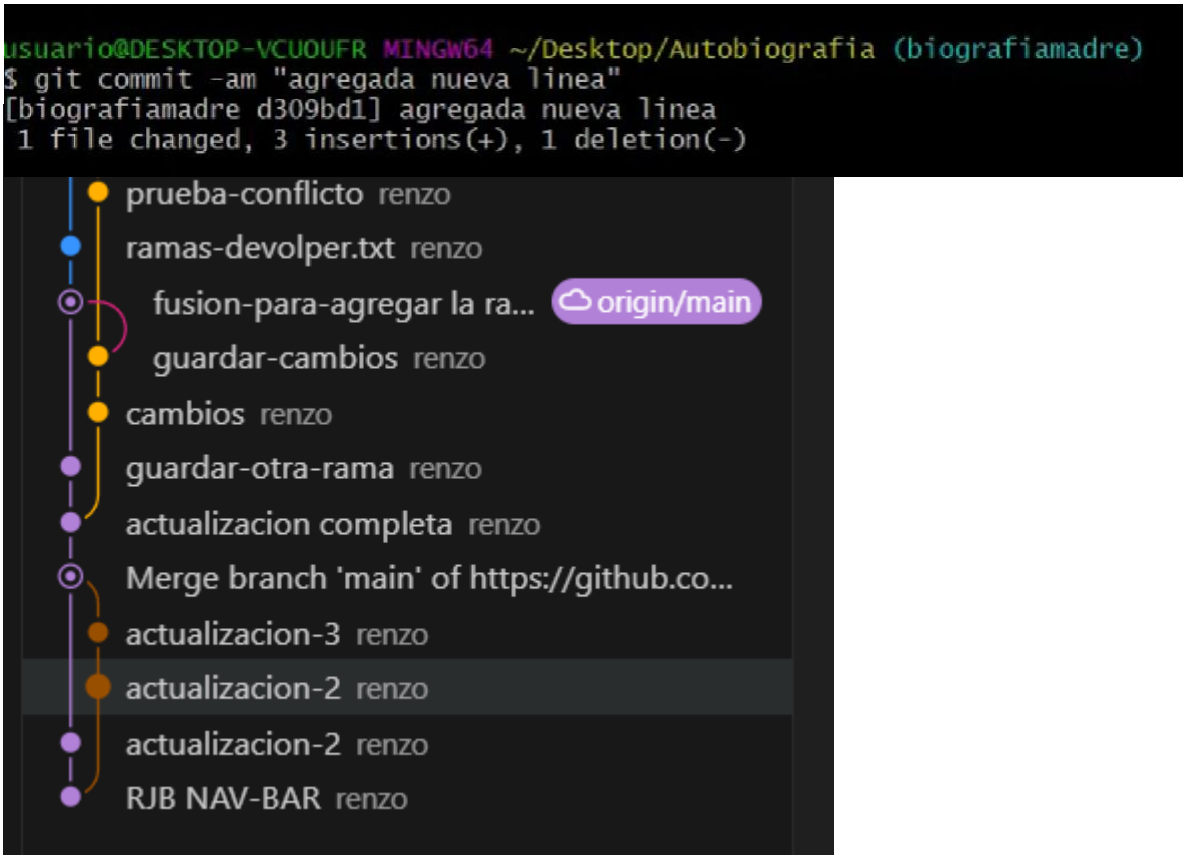
```
YETREN@LAPTOP-384VTT80 MINGW64 ~/Documents/PROGRAMACION/nav-bar (experimentar)
$ git status
On branch experimentar
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   ramas-devolper.txt

no changes added to commit (use "git add" and/or "git commit -a")

YETREN@LAPTOP-384VTT80 MINGW64 ~/Documents/PROGRAMACION/nav-bar (experimentar)
$ git commit -am prueba-conflicto
[experimentar b16fe08] prueba-conflicto
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Diferentes ramas

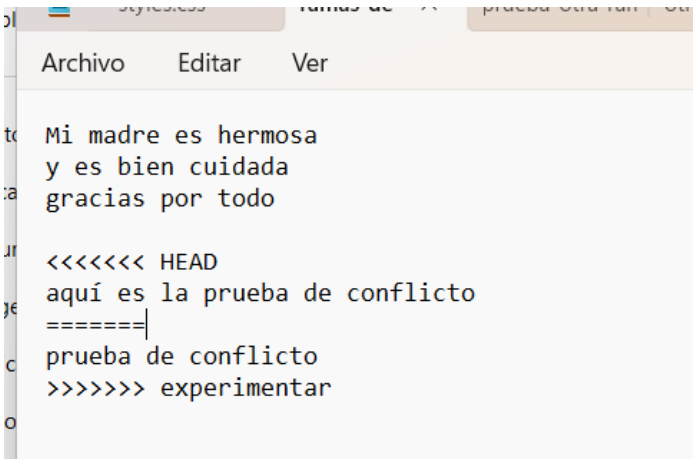




¿Qué es y cómo solucionar un conflicto?

Un conflicto es cuando se modifica una misma línea de código en dos ramas diferentes y se confirman (commit -am) y guardan en el repositorio local. luego se quieren unir esas dos ramas con un **git merge** desde la rama **master/main** y se genera un conflicto porque hay dos versiones distintas en la misma línea de código y git te pregunta cuál de las dos dejar y se coloca en estatus **main/mering**

Ejemplo: Este es el mensaje que aparece en el archivo modificado, Te notifica >>> a que rama pertenece

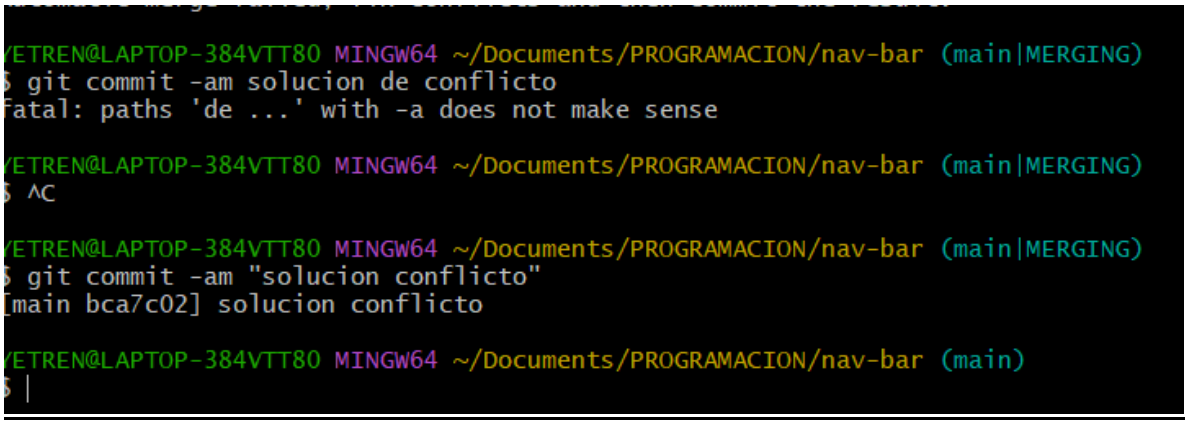


Solución

Para solucionar el conflicto simplemente se ponen de acuerdo los desarrolladores cual línea de código dejar y una vez tomada la decisión, en la línea de código que aparece el mensaje con las líneas modificadas de las versiones.

Se borra la línea descartada y se hace un nuevo commit -am con los cambios.

Ejemplo en la consola cuando hay un conflicto se pone en estado MERGING



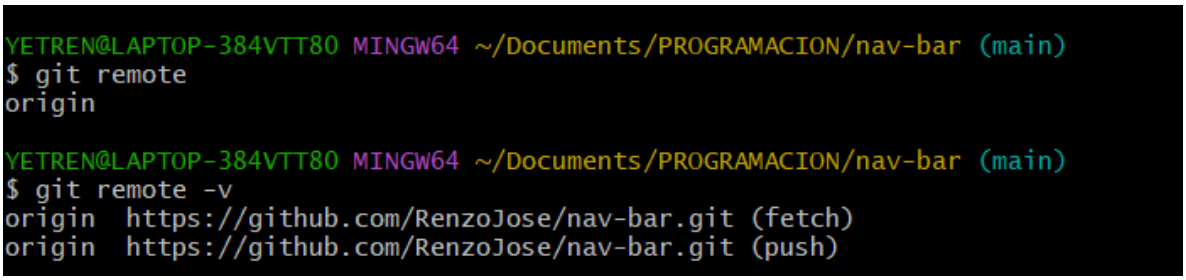
> **Nota:** ¡Antes de navegar entre los commit y las ramas es necesario que hallas guardado todos los cambios para evitar que se pierdan!

git commit -am “mi commit”

Nos permite ejecutar tanto git add como git commit, pero siempre y cuando ya se halla realizado con anterioridad**. ----→ojo**

<u>Que es Github</u>

Es un repositorio remoto, es decir es una estación de trabajo desde una nube donde puedes acceder a repositorios remoto y trabajar en comunidad en proyectos de desarrollo



Git clone: crea una copia local de un repositorio remoto. Básicamente, descarga todos los archivos, ramas e historial del repositorio remoto a tu equipo local, lo que te permite trabajar en el proyecto localmente y luego enviar los cambios al repositorio remoto.

Example:

git clone https://github.com/RenzoJose/nav-bar.git

y lo guarda en el home

Git pull: descarga los cambios más recientes de la rama remota que estás siguiendo y los fusiona con tu rama local actual.

Si queremos que traiga los datos de una rama especifica colocamos las siguientes sintaxis

Git pull origin nombre de la rama

Git push: envía los cambios guardados en nuestro repositorio local a nuestro repositorio remoto (la nube github) cabe destacar, que si queremos enviar los cambios a una rama especifica colocamos origin y el nombre de la rama

Ejemplo

Git push origin nombrelarama

Flujo de trabajo

Git clone => Clonar el repositorio completo.

Git push => enviar un commit al repositorio remoto.

Git pull => traer la última versión de una rama determinada.

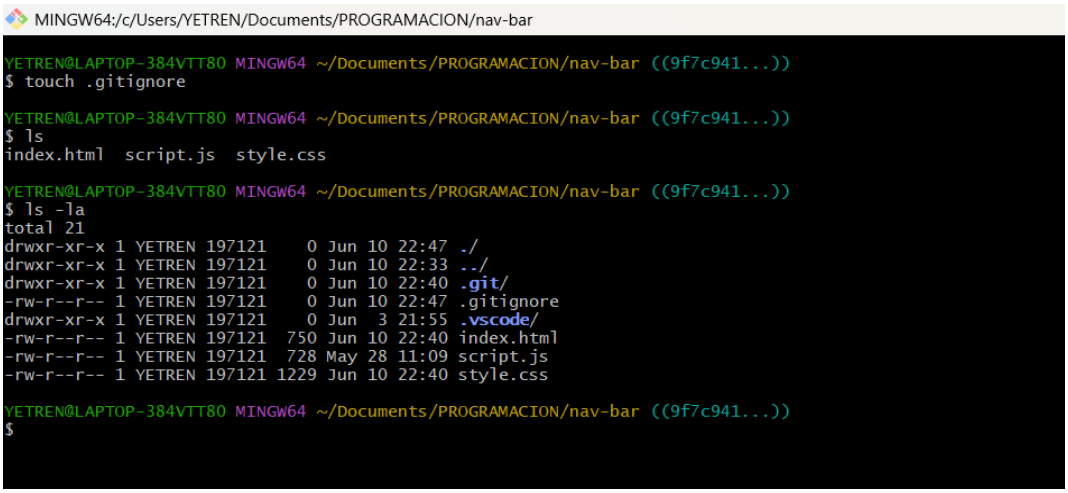
¿Qué es y cómo utilizar el Git ignore?

Un archivo .gitignore es un archivo de texto que se utiliza en Git para especificar qué archivos y directorios deben ignorarse al rastrear los cambios en un proyecto. Esto significa que esos archivos no serán incluidos en el repositorio de Git ni en las actualizaciones.

En resumen, el archivo .gitignore ayuda a evitar la inclusión de archivos innecesarios o confidenciales en el repositorio de Git.

Ejemplo para crear una carpeta gitignore utilizamos la siguiente línea de comando

touch .gitignore // esta línea nos crea una carpeta .gitignore para colocar losarchivos que no queremos que cargue



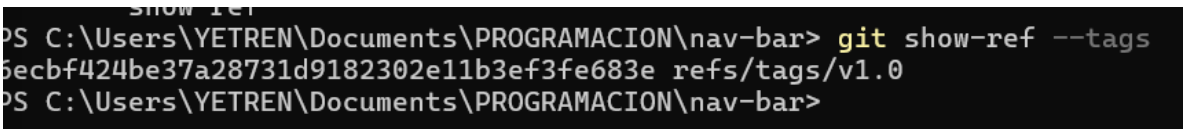
En Git, un tag (etiqueta): es un puntero fijo a una confirmación específica en el historial del repositorio. Sirven para marcar hitos importantes, como versiones de software, y no cambian como las ramas, que pueden moverse a medida que se hacen nuevas confirmaciones.

Para crear un tag utilizamos: git tag -a **nombre-tag** -m **“mensaje del tag” versión-del-tag**

Ejemplo de de la terminal

Si queremos buscar cual es el commit de la versión que colamos la etiqueta tag utilizamos el siguiente código

git show-ref –tags



Si queremos enviar el tag al repositorio github utilizamos

Git push origin --tags

```
  otra-rama
PS C:\Users\YETREN\Documents\PROGRAMACION\nav-bar> git push origin --tags
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 166 bytes | 83.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/RenzoJose/nav-bar.git
 * [new tag]         v1.0 -> v1.0
```

Si queremos eliminar un tag utilizamos el siguiente comando:

git tag -d **nombre-del-tag** **//esto es el repositorio local**

git tag push origin :refs/tags**/versión-tag** **// esto elimina el tag en el repositorio remoto**

Ejemplo:

```
PS C:\Users\YETREN\Documents\PROGRAMACION\nav-bar> git push origin :refs/tags/v1.1
To https://github.com/RenzoJose/nav-bar.git
 - [deleted]         v1.1
PS C:\Users\YETREN\Documents\PROGRAMACION\nav-bar>
```

Touch: sirve para agregar carpetas dentro del la termianl Basch

Interfax de desarrollo se puede ver con el comando desde la terminal

gitk