Notas de Git & Github

Nahuel Passano
nahue.passano@gmail.com

Enero del 2022

1. ¿Que es Git?

Git es un **sistema de control de versiones local**, se utiliza en proyectos con cambios constantes en los scripts y en el desarrollo colaborativo.

Un **sistema de control de versiones registra los cambios del código.** En base a una versión inicial del código, y se guardan únicamente los cambios del código, **no se genera un nuevo archivo actualizado**. Esto permite localizar los cambios en el código (que líneas se modificaron), quien hizo los cambios y la posibilidad de volver a un estado anterior del código actual, entre otras funcionalidades.

2. ¿Que es GitHub?

GitHub es un sitio web que **almacena repositorios de git en un servidor remoto**. Esto facilita el trabajo colaborativo, donde **varios desarrolladores trabajan sobre el mismo repositorio remoto pero cada uno edita en sus versiones locales**. Además, amplía aún más su funcionalidad con respecto a Git ya que permite hostear páginas web estáticas, previsualizar archivos .ipynb (notebooks de Python), entre otros. Existen mas servidores web que almacenan archivos de Git y permiten desarrollar de manera colaborativa como GitHub como GitLab y Bitbucket, entre otros.

3. ¿Como se usa Git?

Hay distintas formas de utilizarlo, tanto con GUI (GitHub Desktop, GitKraken) o por comandos de consola. En este caso, haremos hincapié en el uso mediante la **consola de bash**. Bash es la consola que viene por default en sistemas operativos como Linux y MacOS, pero no en Windows (PowerShell), y es importante previo a utilizar Git, conocer alguno de sus comandos básicos.

- `pwd`: Devuelve el directorio actual.
- `cd <path>`: Cambiar al directorio que se le indique.
- `ls`: Devuelve todos los archivos contenidos en el directorio actual.

4. ¿Como se manejan los cambios dentro de Git?

En Git, de manera local, hay 3 estados: *Working directory*, *staging area* y *repository*. El *working directory*, que es donde se trabaja con todos los archivos, el *staging area* es donde se agregan todos los archivos para guardarlos, y donde finalmente se guardan los cambios es en el *repository*.

Para iniciarse no tiene mucho sentido

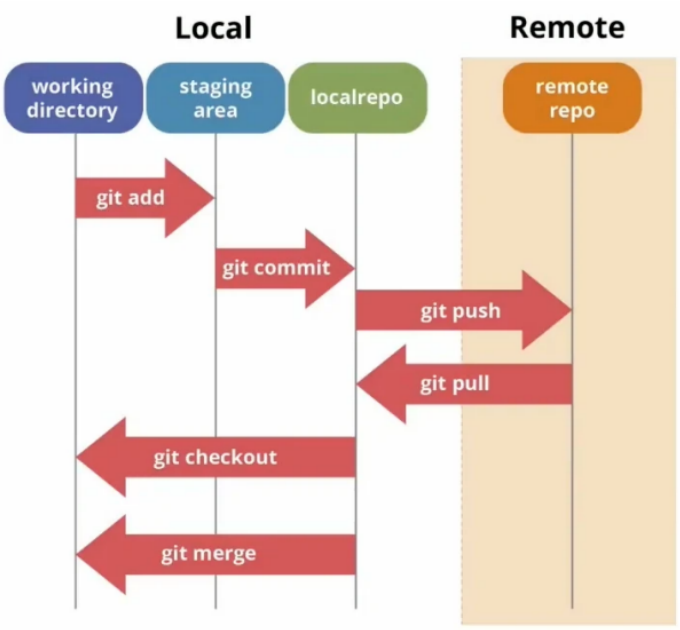

Figura 1: Estados de los archivos dentro de Git

5. Comandos básicos

Los comandos básicos son:

- `git init`: Se utiliza para indicar a Git que se creará un proyecto nuevo, o para trackear los archivos de un proyecto ya en curso.
- `git status`: Se utiliza para saber en que estado estan los archivos, si estan en el *working directory*, *staging area* o *repository*.
- `git add <file>`: Se utiliza para pasar los archivos del *working directory* al *staging area*
- `git commit -m "<msg>"`: Se utiliza para pasar los archivos del *staging area* al *repository*. Se crea un primer checkpoint de la primera versión del código.
- `git push origin branch_name`: Se utiliza para subir el código a la rama *branch_name* de un repositorio remoto.
- `git pull origin branch_name`: Se utiliza cuando se trabaja en desarrollo colaborativo. Trae los cambios que han hecho los otros desarrolladores de la rama *branch_name*
- `git clone <url>`: Clona un repositorio remoto a la pc de manera local.

6. Instalar Git

Se descarga de `https://git-scm.com/`

7. Primer ejemplo

Una vez instalado, una de las herramientas que tendremos disponibles es **Git Bash**, es la terminal de Git donde ejecutaremos nuestros comandos básicos. En primer lugar, creamos una carpeta con algún

archivo adentro (en mi caso es un archivo `.py`). Para inicializar la consola de Git en la carpeta, se aprieta click derecho sobre la carpeta y se selecciona *"Git Bash Here"*.

Para inicializar el Git dentro de mi proyecto nuevo, ejecuto el comando `git init`. Luego, para ver en que estado están nuestros archivos dentro del proyecto, ejecutamos `git status`. Lo que aparecerá en la consola de Git es:

```
git status
On branch master

No commits yet

Untracked files: (use "git add <file>..." to include in what will be committed)
  ejemplo.py

nothing added to commit but untracked files present (use "git add" to track)
```

Justamente, no hemos agregado nuestros archivos al entorno de trabajo (al *staging area*). Por lo tanto, para añadir el archivo al entorno de trabajo ejecutamos:

```
git add ejemplo.py
```

Ahora, si ejecutamos `git status` saldrá:

```
git status
On branch master

No commits yet

Changes to be committed:
  new file: ejemplo.py
```

Tip: Para añadir todos los archivos que hay en mi directorio actual, se hace con:

```
git add .
```

No siempre es recomendable utilizar esta funcionalidad ya que a veces no todos los archivos que están por commitear pertenecen al mismo checkpoint (es una sugerencia para organizar bien los commits).

Ya se añadió mi archivo al espacio de trabajo pero aún no se guardó en un repositorio. Para crear nuestro primer punto de control, debemos utilizar el comando `git commit`, pero antes, debemos decirle a Git quien está realizando los cambios en los archivos, para ello se utilizan los comandos:

```
git config --global user.email "<email>"
```

```
git config --global user.name "<name>"
```

En este caso:

```
git config --global user.email "nahue.passano@gmail.com"
```

```
git config --global user.name "Nahuel Passano"
```

Ahora si podemos hacer nuestro primer checkpoint con `git commit`. Si se ejecuta únicamente:

```
git commit
```

Nos abrirá una nueva consola, donde para tipear en la primera fila debemos hacerlo con *Insert*. Esta primera fila es para añadir un comentario sobre que se está subiendo y/o actualizando. Para finalizar el commit, apretamos escape (*Esc*) y luego tipeamos `:wq` (`w` de *Write* y `q` de *Quit*). Presionando *Enter* ya se actualiza nuestro proyecto. Por eso es mas sencillo utilizar

```
git commit -m "<msg>"
```

ya que incorporamos el mensaje asociado al commit en la misma línea.

Con respecto al mensaje dentro del commit, es recomendable tener algunas keywords al inicio del commit para saber de que se trata. Por ejemplo yo utilizo:

```
git commit -m "[feat]: ..."
git commit -m "[fix]: ..."
git commit -m "[docs]: ..."
```

Donde *feat* la utilizo para hacer commits de upgrades, o features nuevas que se añaden al código. *fix* la utilizo para commitear alguna solución a un bug o simplemente a algun error de tipeo dentro de los scripts. Y finalmente *docs* la utilizo para commitear cambios en las documentaciones, tanto en el README.md como en los docstrings de las funciones desarrolladas.

8. Otros comandos importantes

Para poder ver los puntos de control creados es con el comando:

```
git log
```

Cuando se modifica un archivo que está dentro del repositorio, podemos volver a la ultima versión que fue guardada en Git con el comando

```
git checkout -- <file>
```

De hecho, si tipeamos `git status`, veremos en consola los archivos modificados a partir de el último checkpoint del archivo.

A su vez, para analizar las diferencias que hay entre las versiones de un archivo se debe insertar el comando:

```
git diff
```

En rojo aparecerán las partes que faltan y en verde las partes que se agregaron. Para actualizar el archivo en el repositorio, debemos añadirlo nuevamente con el comando `git add`, y crear el punto de control con `git commit`.

Para ignorar archivos o carpetas dentro de mi repositorio, debo crear el archivo

```
.gitignore
```

En dicho archivo, se escriben los nombres de los archivos o carpetas que se desean ignorar, uno por linea. Para finalmente ignorarlos, añadimos nuestro archivo `.gitignore` al repositorio. Para ignorar un tipo de archivo se debe poner `*.class`. Por ejemplo si quiero ignorar todos los archivos .jpg tendría que agregar una línea en el `.gitignore` con `*.jpg`.

9. Versiones alternativas

El manejo de versiones alternativas es un punto fuerte en el control de versiones de un proyecto. En Git se denominan **ramas** o **branchs**. Por default, si se inicia un repositorio con el comando `git init`, se crea la rama *master*, y si se inicia un repositorio desde GitHub para despues clonarlo en nuestra PC, se crea la rama *main*. Las versiones alternativas sirven para ramificar el trabajo, y esto nos permite corregir errores, o mismo desarrollar nuevas features sin intervenir en la rama principal que es donde se alojará la versión estable de nuestro proyecto. Al crear una nueva branch o rama, se copian los archivos de la rama desde la cual se generó, pero la gran ventaja es que las modificaciones que hagamos sobre la nueva rama son independientes al resto de ramas. Mismo cuando se desarrollan distintas versiones de un software, se mantiene en la rama principal la versión estable del software, mientras que en las ramas alternativas se van desarrollando los cambios para la próxima versión, para luego fusionar la versión estable con los cambios a implementar.

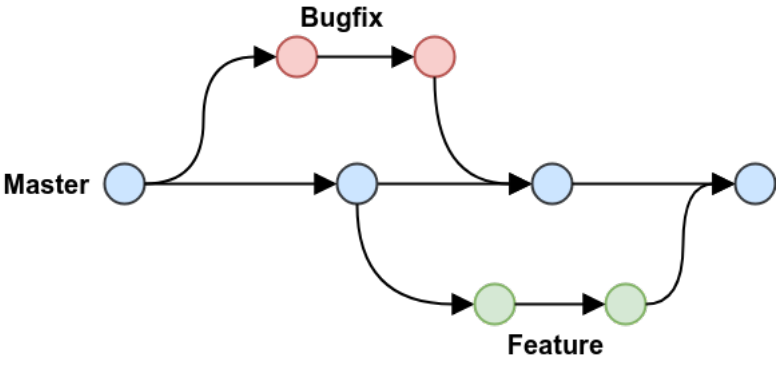

Figura 2: Ejemplo de un diagrama de ramas o branchs

Para construir versiones alternativas a un proyecto se utiliza el comando:

```
git branch <branch_name>
```

Donde <branch_name>será el nombre con el cual identificaremos la nueva rama. Para ver todas las branchs dentro de un proyecto se hace con

```
git branch
```

Y para cambiar en que versión estaremos trabajando con Git, se hace con:

```
git checkout <branch_name>
```

Cuando finalmente corregimos algún bug o desarrollamos alguna feature nueva y queremos integrarla a la versión principal, se deben 'fusionar' ambas versiones con el comando:

```
git merge <branch_name>
```

Si por ejemplo tengo las branchs *main* y *new_feature*, y quiero integrar *new_feature* a *main*, debo cambiarme a la rama *main* con

```
git checkout main
```

y luego fusionar dicha rama con *new_feature* a través de:

```
git merge new_feature
```

En la consola de bash se puede ver en la rama que estamos, asi como el path actual y el usuario.

Figura 3: Ejemplo del uso de `git checkout`

10. Repositorio remoto

Para trabajar y subir nuestro proyecto a un repositorio remoto, primero debemos crear una cuenta en GitHub (`https://github.com/`) y crear nuestro primer repositorio. Al crear el repositorio remoto, el mismo tiene una dirección la cual será del tipo {https://github.com/user-name/repository-name.git}.

- En el caso de generar el repositorio de manera local con `git init`, al mismo debemos asociarle un repositorio remoto, el cual será donde subamos nuestros cambios e interactuemos con los otros colaboradores del proyecto. Para asociar nuestro repositorio local con dicho remoto se hace con:

```
git remote add origin https://github.com/user-name/repository-name.git
```

- En el caso de crear el repositorio de manera remota en GitHub para luego clonarlo en nuestra computadora, la asociación del repositorio local clonado con el repositorio remoto ya se realiza con la clonación:

```
git clone https://github.com/user-name/repository-name
```

Y para subir nuestros ckeckpoints y cambios al repositorio remoto se hace con:

```
git push origin branch_name
```

donde *branch_name* será la rama del remoto a la cual queremos subir nuestros cambios.

11. Desarrollo colaborativo

11.1. Primeros pasos

1. Crear un repositorio remoto en GitHub
2. Clonar el repositorio en nuestra computadora con:

```
git clone https://github.com/user-name/repository-name
```

3. Añadir todos los scripts/archivos/etc a utilizar y desarrollar de manera común y corriente.
4. Pasar los archivos con los que se estuvo trabajando al Staging Area con:

```
git add <file>
```

5. Crear un checkpoint de los cambios hechos (o tambien conocido como commitear) pasando los archivos del Staging Area al Local Repository con:

```
git commit -m "<msg>"
```

6. Subir mis checkpoints al repositorio remoto

- **Si son los primeros cambios que se van a subir al repositorio:** Simplemente pusheamos los checkpoints con:

```
git push origin branch_name
```

- **Si mi versión local está desactualizada con la versión remota:** Esto sucede cuando otro colaborador desarrolló y subió sus cambios al remoto, por lo tanto nuestra versión local estará desactualizada y nos dirá que hay conflicto entre los checkpoints si deseamos pushear. Para ello, antes de pushear (subir los checkpoints) debemos pullear la versión mas reciente desde el remoto con el comando:

```
git pull origin branch_name
```

Ahora tendremos nuestra versión local actualizada con los checkpoints del remoto, por lo tanto esta versión tendrá los checkpoints desarrollados por el otro colaborador junto con mis checkpoints. Ahora sí puedo pushear y quedará todo unificado con:

```
git push origin branch_name
```

11.2. Ejemplo: Flujo de trabajo en Infiniem Labs

En general, para los proyectos en GitHub hay dos branchs, *main* y *develop*. La rama *main* tiene código que pasó por el proceso de revisión, y se encuentra en funcionamiento, mientras que en la rama *develop* se van subiendo los commits del desarrollo.

La primera vez que se inicia un proyecto, se debe clonar la rama *develop* con el comando `git clone <url>`a la rama *develop_local*. Luego, de manera local, se genera la rama *test_local*, la cual será donde hagamos pruebas del código y desarrollemos de forma local.

Una vez terminado un fix o update, si se desea subir un commit al repositorio remoto, primero se debe pullear la rama *develop* a *develop_local* para actualizar nuestro *develop_local* ante cambios que pudo haber hecho otra persona luego de mi ultimo commit (`git pull origin develop` estando dentro de la rama *develop_local*). Una vez actualizado *develop_local*, se debe fusionar la actualización con la rama *test_local*, que es donde estuvimos desarrollando nuestro código, para ello utilizamos el comando `git merge test_local` estando en la rama *develop_local*. Una vez fusionado, estando en la rama *develop_local*, subimos nuestros cambios a *develop* con el comando `git push origin develop`.

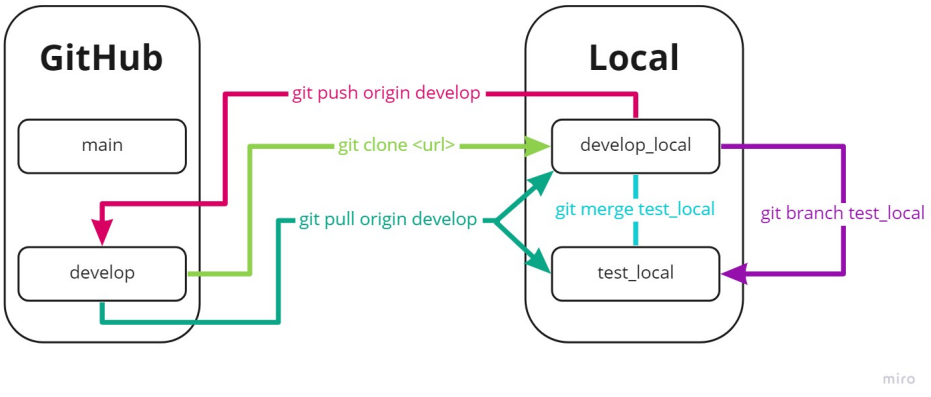

Figura 4: Flujo de trabajo utilizando varias branchs de manera local