



Les damos la bienvenida!





Argentina Programa





Taller de Github



Objetivos de la clase



- Manejo de GitHub a nivel local y remoto.

- Crear nuestro primer repositorio GitHub.



¿Qué es un VCS? (Version control system)



Con el avance en el desarrollo de software, se volvió necesaria la gestión de los diversos cambios que los elementos de un proyecto, van sufriendo a lo largo de la construcción del mismo. Un VCS justamente es un sistema encargado de resolver esa gestión.



Nacimiento de GIT



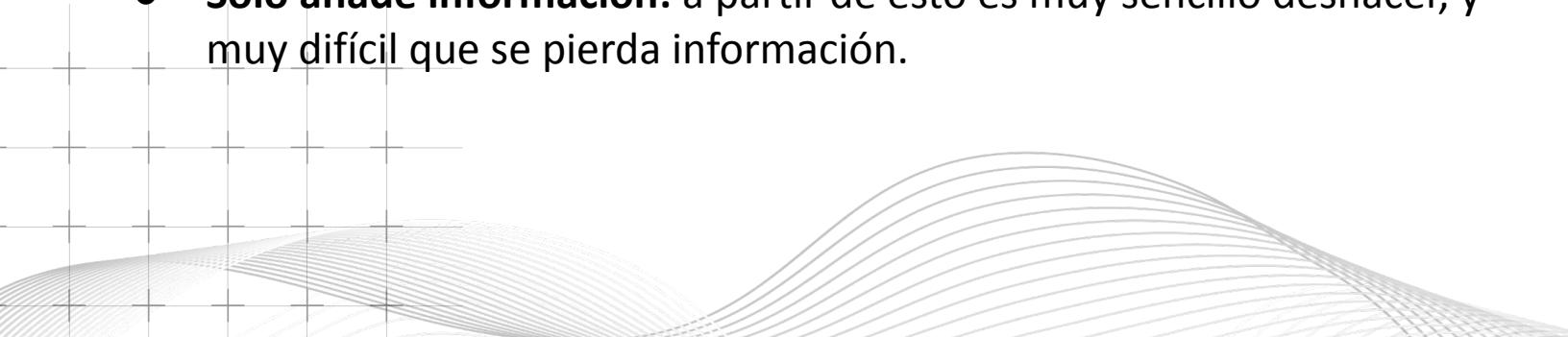
A partir de la necesidad anteriormente descrita es que surge GIT. Este sistema fue desarrollado por Linus Torvalds (creador del Sistema Operativo Linux, y gran precursor del software libre), haciendo especial énfasis en el desarrollo de código a nivel cooperativo, partiendo de la eficiencia y la confiabilidad como pilares necesarios dentro de un VCS.



Características de GIT



- **Integridad:** mediante sistemas de comprobación, es imposible cambiar cualquier elemento del contenido sin que GIT lo sepa (será muy común ver hash como este: 24b9da6552252987aa493b52f8696cd6d3b00373).
- **Funcionamiento local:** la mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para funcionar. Por lo general, no se necesita información de ningún otro computador de tu red.
- **Sólo añade información:** a partir de esto es muy sencillo deshacer, y muy difícil que se pierda información.





Git a nivel local

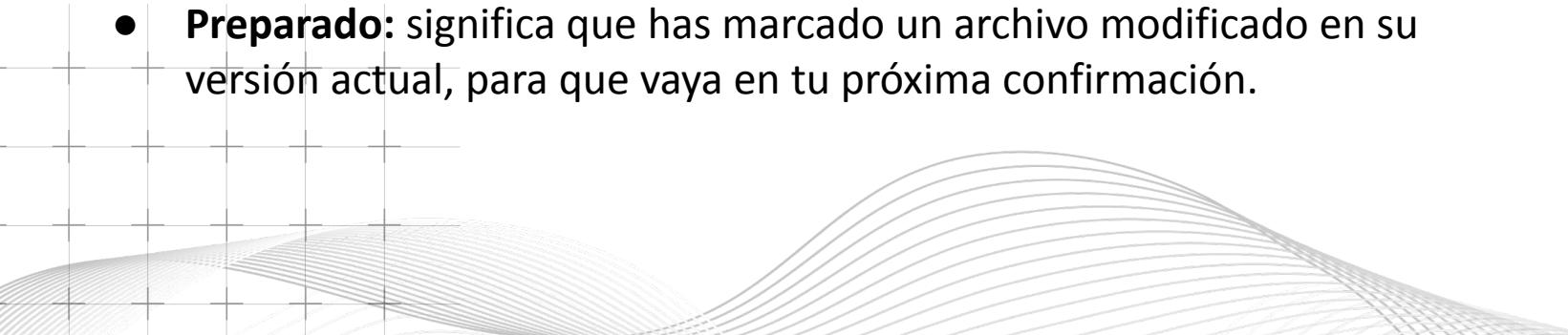


Estados en GIT



Este es uno de los detalles más importantes dentro de Git: el sistema consta de tres estados principales en los que se pueden encontrar un archivo.

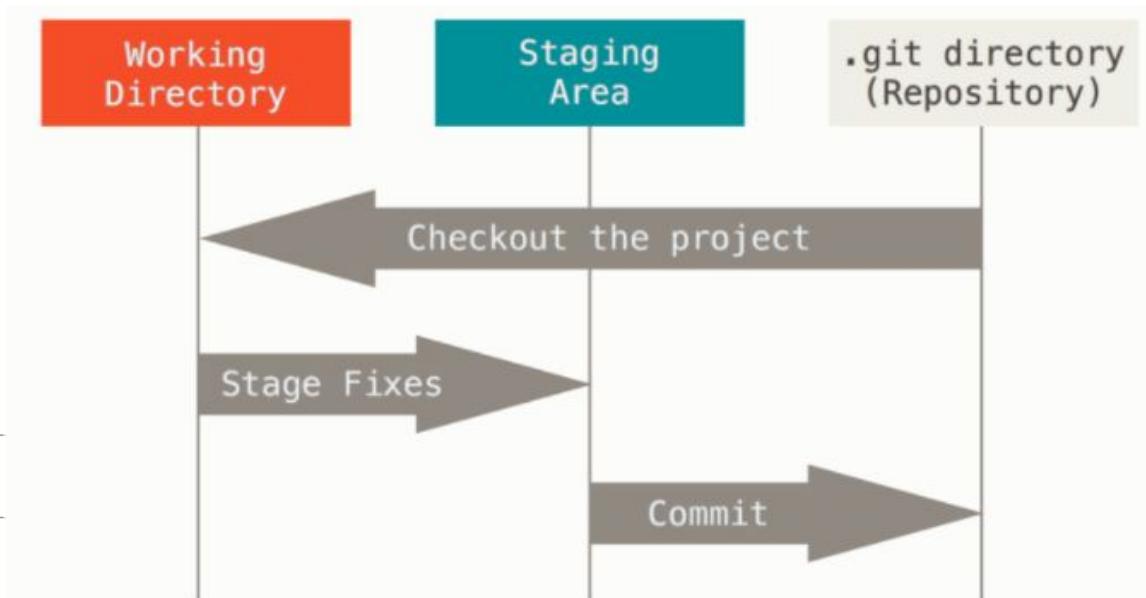
- **Confirmado:** significa que los datos están almacenados de manera segura en tu base de datos local.
- **Modificado:** significa que has modificado el archivo, pero todavía no lo has confirmado a tu base de datos.
- **Preparado:** significa que has marcado un archivo modificado en su versión actual, para que vaya en tu próxima confirmación.



Cambios de estado



Cuando no tenemos cambios instanciados, se dice qué nuestro directorio de trabajo está limpio (working directory clean):



La forma de pasar de un estado al otro, es a través del ciclo descrito en la gráfica. El paso entre una etapa y otra, se da a partir de los distintos comandos Git.

Flujo de trabajo a partir de estados



A partir de nuestro directorio de trabajo, se describe cómo se van dando los procesos previamente mencionados:

- Se modifican archivos en el directorio de trabajo.
- Se preparan los archivos para su confirmación, esto significa agregarlos al “Staging Area” (**git add <fileName>**).
- Se confirman los cambios, pues al momento de confirmar, los archivos se toman tal cual están en el área de preparación, almacenando de manera permanente una copia de estas versiones en nuestro directorio Git. (**git commit**).



Comandos y flujo de trabajo



Veremos en mayor detalle el empleo de comandos para movernos dentro del flujo de trabajo local. A su vez, todos los pasos que damos los podemos controlar con el comando “**git status**”:

- Partimos de un directorio de trabajo limpio.
- Se producen cambios, los cuales deben ser instanciados para su posterior confirmación. Para ello usamos git add (continuado del nombre del archivo a agregar, o un punto, para agregar todos los cambios disponibles).



Comandos y flujo de trabajo



- Otra opción respecto a los cambios es descartarlos; para hacerlo, usaremos git restore (continuado del nombre del archivo a descartar, o un punto, para descartar todos los cambios disponibles).
- Por último, se confirman los cambios, dejando nuevamente limpio el directorio de trabajo. Esto se hace a través de git commit -m “descripción de la confirmación”.





Vamos a practicar lo visto!

- Flujo local de git -



Flujo local Git (1/5)



A continuación, a partir de la creación y posterior modificación de un simple archivo "index.html", estaremos analizando los comandos git a usar, y las respuestas de los mismos. Como pasos iniciales creamos el nombrado archivo, y damos inicio al repositorio local. Luego chequeamos el directorio de trabajo, los comandos y salidas.

```
git init
Initialized empty Git repository in C:/Users.....
-----
git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    index.html
nothing added to commit but untracked files present (use "git add" to track)
```



Flujo local Git (2/5)



Aún ni siquiera hemos confirmado que nuestro archivo existe. Es bastante común cuando se da inicio a un repositorio hacer un primer commit; para lograr esto, según lo visto anteriormente, deberíamos ejecutar los siguientes comandos:

Debemos tener en cuenta qué git add no produce salida, pero podemos chequear estatus para controlar.

```
git add index.html
-----
git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   index.html
-----
git commit -m "primer commit"
[master (root-commit) 9c18736] primer commit
  1 file changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 index.html
```



Flujo local Git (3/5)



```
git status
On branch master
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:   index.html

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

-----
-
git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   index.html
```

Luego de nuestro primer commit, haremos cambios en el archivo `index.html`, y chequeamos el status de nuestro directorio de trabajo. Luego agregamos dicho cambio, esta vez utilizando `"."` en lugar del nombre del archivo. Por último, volvemos a chequear el status.



Flujo local Git (4/5)



Llevamos a cabo el segundo **commit**, y luego corremos **git status** (aunque esto no es necesario, en este momento se hace con fines demostrativos).

Por otro lado, cabe aclarar que haber ejecutado **git add .** seguido de **git commit -m "segundo commit"** sería equivalente a ejecutar **git commit -am "segundo commit"**.

```
git commit -m "segundo commit"
[master 0619123] segundo commit
  1 file changed, 1 insertion(+)

-----
git status
On branch master
nothing to commit, working tree clean
```



Flujo local Git (5/5)



Por último, como para mostrar una variante al flujo efectuaremos un cambio sobre el mismo archivo, sólo que esta vez en lugar de agregarlo al “stage area” nos desharemos de él con “git restore .” (también sirve git restore “nombreDelArchivo”, su lógica es idéntica al comando git add, pero el efecto totalmente inverso).

```
git restore .
-----
git status
On branch master
nothing to commit, working tree clean
```



¿Qué implica un árbol de trabajo limpio?



A lo largo del ejemplo anterior, habremos observado el mensaje "**nothing to commit, working tree clean**", que apareció cuando confirmamos nuestros cambios mediante git commit, o cuando nos deshicimos de ellos mediante git restore.

Pero ¿qué significa?

A través del mensaje y la interpretación de la palabra "limpio", podremos suponer que aparece cuando no hay cambios disponibles, para confirmar o pasar la "**staging área**" mediante git add, ya que los mismos fueron confirmados o descartados. A su vez, esto nos habilita a acciones posteriores, tanto a nivel local como a nivel remoto.





Branches

- (ramas) -

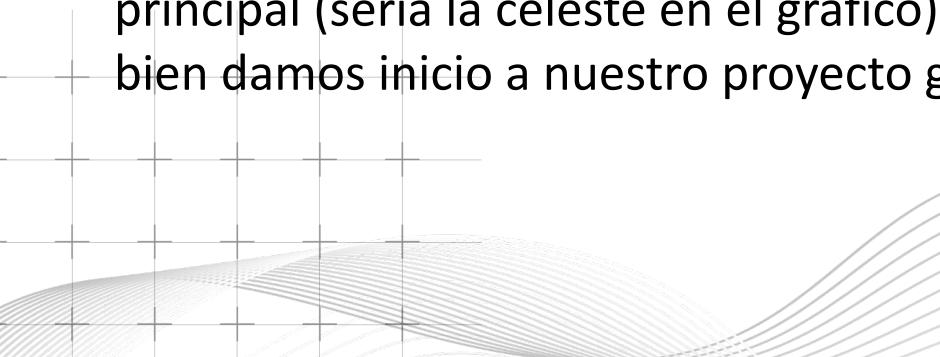
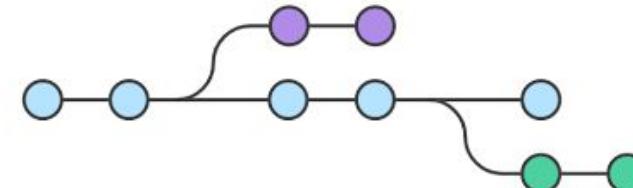


Ramificaciones



Prácticamente todos los sistemas de control de versiones implementan una lógica de ramificaciones. Esto implica poder “separarnos” de la línea principal de desarrollo, y trabajar por fuera de la misma sin que los cambios que podamos introducir la afecten.

En sintonía con lo desarrollado filminas atrás, necesitamos un árbol de trabajo limpio (sin cambios a declarar) para poder movernos entre ramas o crear una nueva, de manera correcta. Debemos saber que la rama principal (sería la celeste en el gráfico) se llama main o master, y se crea ni bien damos inicio a nuestro proyecto git:



Comandos para trabajar ramificaciones



Al momento de trabajar con branches, debemos tener en cuenta los siguientes comandos:

- **git branch “nombre de la rama”**: nos permite crear una nueva rama.
- **git checkout “nombre de la rama”**: se emplea para el “salto” de una rama a otra.
- **git checkout - b “nombre de la rama”**: combina los dos comandos anteriores, creando la nueva rama, y ubicando el desarrollo en la misma.
- **git branch**: lista las ramas disponibles a nivel local.
- **git branch --all (-A)**: lista las ramas disponibles a nivel local y remoto.
- **git merge “nombre de la rama”**: fusiona la rama nombrada a donde estamos ubicados.
- **git branch -D “nombre de la rama”**: elimina la rama.



Ramificaciones Git (1/2)



```
git branch newBranch
-----
-
git checkout newBranch
Switched to branch 'newBranch'
-----
-
git status
On branch newBranch
nothing to commit, working tree clean
-----
-
git branch
  master
* newBranch
```

El terminal, disponible en VSC, nos informa que estamos ubicados en la rama principal (master). Desde este punto crearemos una nueva ramificación llamada “newBrach”, y luego nos ubicamos en ella.



Ramificaciones Git (2/2)



A continuación realizaremos un proceso equivalente al anterior, sólo que mediante el comando checkout -b, lo que nos permitirá crear una nueva rama, y trasladarnos a ella en una misma acción.

```
git checkout -b otherNewBranch
Switched to a new branch 'otherNewBranch'
-----
git status
On branch otherNewBranch
nothing to commit, working tree clean
```





CONSULTAS?

Muchas Gracias!

