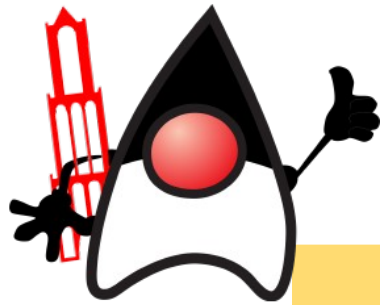
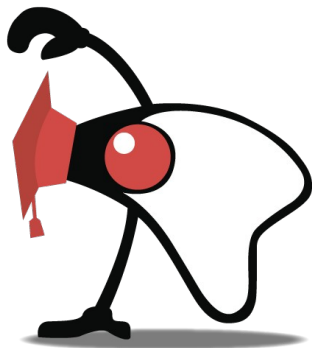
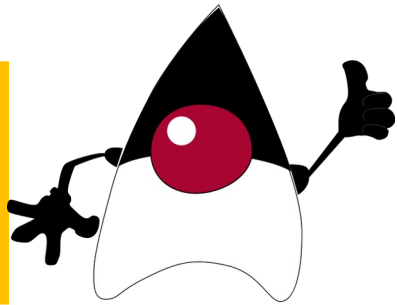




# Curso FullStack Python

Codo a Codo 4.0



# GIT

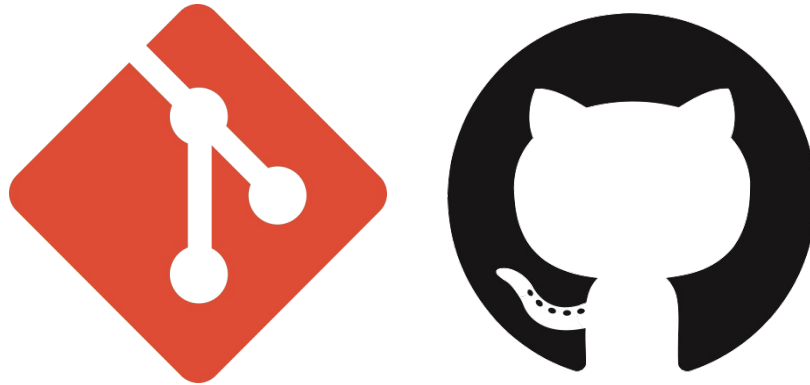
***GIT y GitHub***



# ¿Qué es GIT?

Es un software de control de versiones, su propósito es llevar registro de los cambios en archivos de computadora y coordinar el trabajo que varias personas realizan sobre archivos compartidos.

Existe la posibilidad de trabajar de forma remota y una opción es **GitHub**.



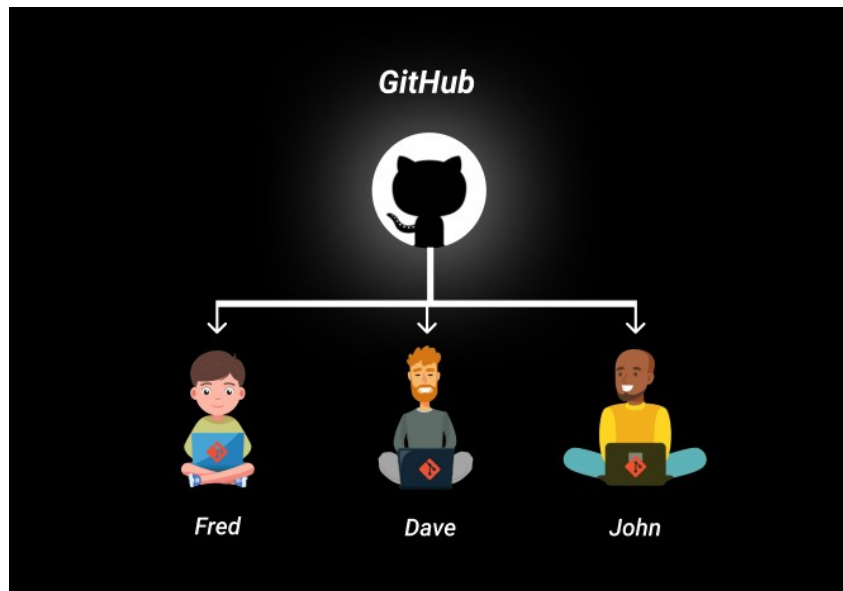
# ¿Qué es GitHub?

Es una plataforma de desarrollo colaborativo para alojar proyectos (en la “nube”) utilizando el sistema de control de versiones Git.

Además cuenta con una herramienta muy útil que es GitHub Pages donde podemos publicar nuestros proyectos estáticos (HTML, CSS y JS) de forma gratuita.

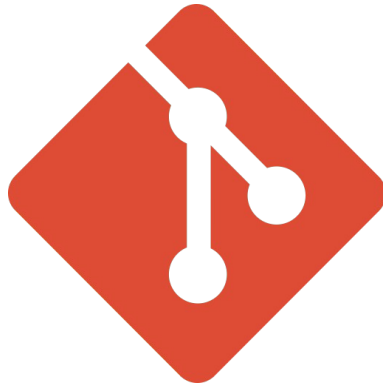
No es la única plataforma, pero si la más popular.

Fue creada por Linus Torvalds para facilitar el desarrollo del núcleo (kernel) de Linux, en el que trabajaban cientos de programadores.



# GIT | Definición

Un sistema que ayuda a organizar el código, el historial y su evolución, funciona como una máquina del tiempo que permite navegar a diferentes versiones del proyecto y si queremos agregar una funcionalidad nueva nos permite crear una rama (branch) para dejar intacta la versión estable y crear un ambiente de trabajo en el cual podemos trabajar en una nueva funcionalidad sin afectar la versión original.



# GIT | ¿Qué permite?

Mediante el control de versiones distribuido permite:

- Manejar distintas versiones del proyecto.
- Guardar el historial o guardar todas las versiones de los archivos del proyecto.
- Trabajar simultáneamente sobre un mismo proyecto.

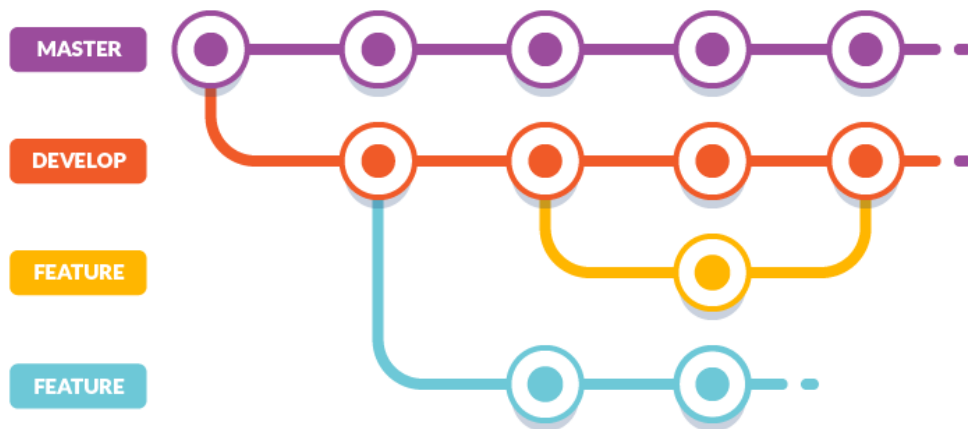
**Para ampliar:**

Guía rápida:

<https://docs.github.com/en/get-started/quickstart/set-up-git>

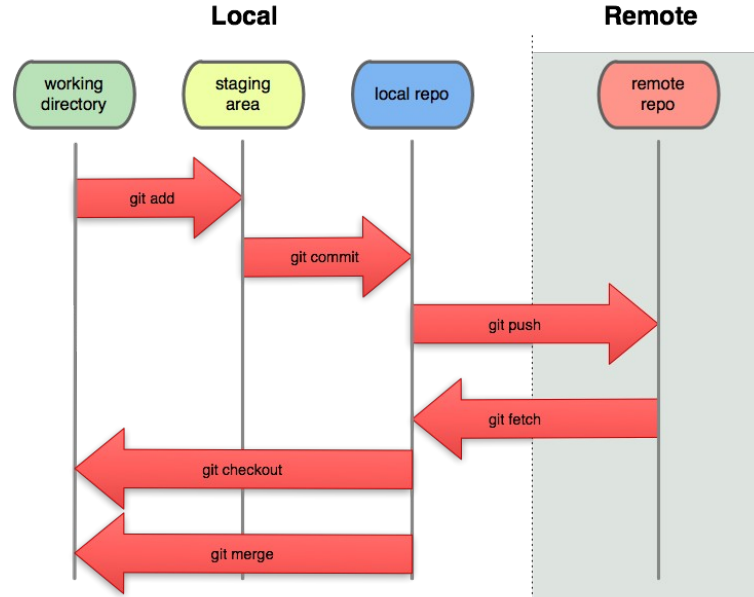
# GIT | ¿Cómo funciona?

Git almacena instantáneas de un mini sistema de archivos, cada vez que confirmamos un cambio lo que Git hace es tomar una "foto" del aspecto del proyecto en ese momento y crea una referencia a esa instantánea, si un archivo no cambió Git no almacena el nuevo archivo sino que crea un enlace a la imagen anterior idéntica que ya tiene almacenada.



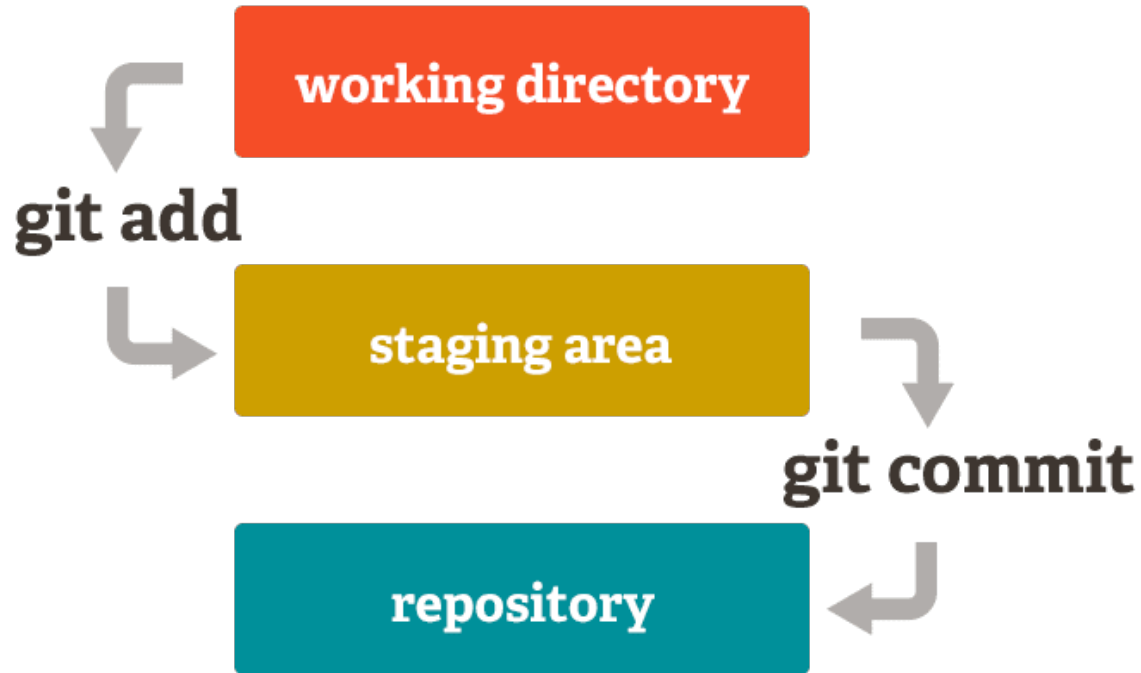
# GIT | Flujo de trabajo

Tenemos nuestro directorio local (una carpeta en nuestra PC) con muchos archivos, Git nos irá registrando los cambios de archivos o códigos cuando nosotros le indiquemos, así podremos viajar en el tiempo retrocediendo cambios o restaurando versiones de código, ya sea en Local o de forma Remota (servidor externo).



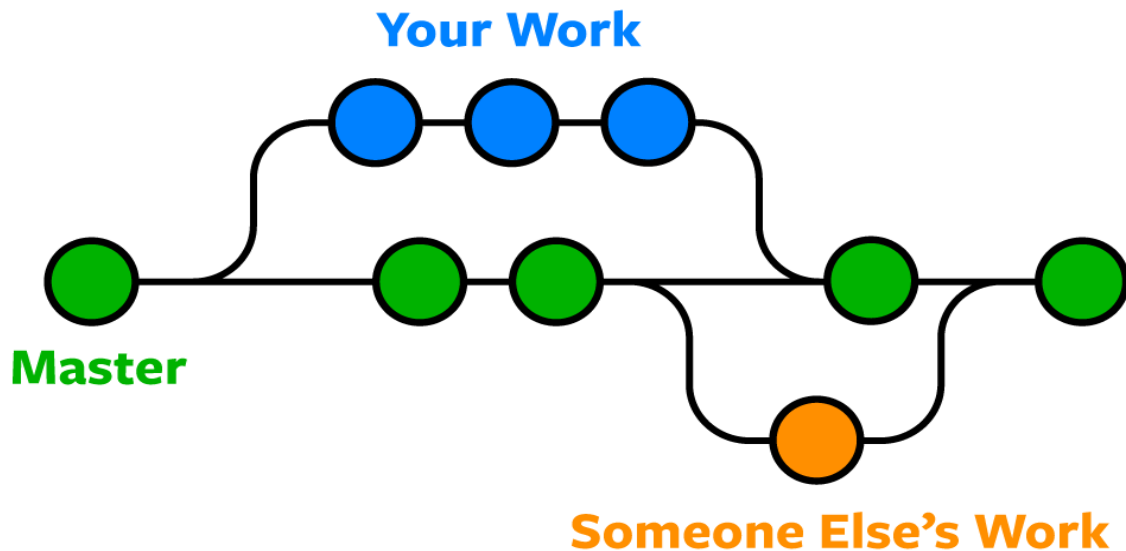


## GIT | Estados



# GIT | Ramas (*branch y merge*)

- **Crear** una rama: `git branch nombreBranch`
- **Unir** la rama a Master: `git merge nombreBranch`
- **Mostrar** en qué rama nos encontramos: `git branch`
- **Cambiar** a una rama determinada: `git checkout nombreBranch`



# GIT | Terminología

- **Repositorio:** es la carpeta principal donde se encuentran almacenados los archivos que componen el proyecto. El directorio contiene metadatos gestionados por Git, de manera que el proyecto es configurado como un repositorio local.
- **Commit:** un commit es el estado de un proyecto en un determinado momento de la historia del mismo, imaginemos esto como punto por punto cada uno de los cambios que van pasando. Depende de nosotros determinar cuántos y cuales archivos incluirá cada commit.
- **Rama (branch):** una rama es una línea alterna del tiempo, en la historia de nuestro repositorio. Funciona para crear features, arreglar bugs, experimentar, sin afectar la versión estable o principal del proyecto. La rama principal por defecto es **master**.
- **Pull Request:** en proyectos con un equipo de trabajo, cada persona puede trabajar en una rama distinta pero llegado el momento puede pasar que dicha rama se tenga que unir a la rama principal, para eso se crea un **pull request** donde comunicas el código que incluye tu cambio y usualmente revisa tu código, se agregan comentarios y por último lo aprueban para darle **merge**. En el contexto de GIT, merge significa unir dos trabajos, en este caso tu **branch** con **master**.

# GIT | Instalación

- 1) Descargar desde: <https://git-scm.com/downloads>.
- 2) Una vez descargado, se emplea la interfaz de línea de comando del sistema operativo para interactuar con GIT:
  - En Windows: abrir la aplicación Git Bash que se instaló junto con GIT.
  - En Mac: abrir la terminal mediante el finder.
  - En Linux: abrir la consola bash.
- 3) Para verificar si está instalado, podemos ejecutar el comando: `git --version`.
  - Si obtenemos respuesta nos indicará la versión de Git que tenemos instalada.
  - En caso de que no poder, ver las instrucciones acá:  
<https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Instalaci%C3%B3n-de-Git>

# GIT | Comandos básicos

- Una vez realizada la configuración básica, hay que utilizar la línea de comando para ubicarnos en la carpeta que queremos hacer control de versión.
  - **Ejemplo:** `cd /Users/Alumno/Desktop/proyecto` y una vez parados en esa carpeta, inicializamos git mediante el comando **git init** que también va a crear un archivo oculto llamado `.git` que se va a encargar de llevar el control de todas las modificaciones que se hagan en esa carpeta.
- Con el comando **git status** se van a mostrar en rojo todos los archivos que git sabe que existen en la carpeta pero que todavía no están registrados.
- Para que esos cambios queden registrados tenemos que añadirlos con el comando **git add** . el punto indica que queremos añadir todos los archivos.
- Si hacemos **git status** todos los archivos van a aparecer en verde.
- (Veremos esto con más detalle)

# Comandos básicos del sistema operativo

- Una vez instalado **git** y con la consola de comandos abierta, podemos utilizar comandos para movernos por el arbol de directorios (carpetas), ver su contenido, crear carpetas nuevas, etcétera.

<b>ls</b>	-> listado de los archivos contenidos en el directorio
<b>ls -lh</b>	-> idem a ls, más detallado
<b>ls -lha</b>	-> idem, pero incluye archivos ocultos
<b>cd</b>	-> Nos permite cambiar a un directorio/carpeta
<b>mkdir</b>	-> Crear directorio/carpeta a partir de la carpeta actual
<b>clear</b>	-> Limpia la pantalla de la terminal de trabajo.



# GIT | Registrar mis datos en git

Antes de realizar lagunas de las operaciones más importantes de git, necesitamos indicarle cual es nuestro correo y cuál es nuestro nombre. Esto se hace con los comandos siguientes:

Proporcionar la dirección de correo:

```
git config --global user.email "correodelusuario@dominio.com"
```

Proporcionar el nombre del propietario:

```
git config --global user.name "NombreDelUsuario"
```



# GIT | Inicializar una carpeta

El primer paso para utilizar **git** en un proyecto consiste en inicializar la carpeta que lo contiene, convirtiéndola en un repositorio local. Para ello, utilizando los comandos provistos por el sistema operativo debemos ubicarnos en ella, y utilizar el comando **git init**:

Si el procedimiento tuvo éxito, veremos al final el mensaje *"Inicializado el repositorio...."*.

Si la carpeta ya habia sido inicializada y usamos **git init**, la misma será reinicializada.

**Nota:** los mensajes de ayuda pueden evitarse asignando un nombre por defecto a la rama inicial de los repositorios con:

**git config --global init.defaultBranch <nombre>**

A screenshot of a terminal window titled 'ariel@AMD: ~/Escritorio/proyecto1'. The terminal shows the following commands and output: 'ariel@AMD:~\$ cd Escritorio', 'ariel@AMD:~/Escritorio\$ cd proyecto1', 'ariel@AMD:~/Escritorio/proyecto1\$ git init'. This is followed by several lines of help text from git, including 'ayuda: Usando 'master' como el nombre de la rama inicial. Este nombre de rama pre-determinado', 'ayuda: está sujeto a cambios. Para configurar el nombre de la rama inicial para usar en todos', 'ayuda: de sus nuevos repositorios, que suprimirán esta advertencia, llame a:', 'ayuda: git config --global init.defaultBranch <nombre>', 'ayuda: Los nombres comúnmente elegidos en lugar de 'maestro' son 'principal', 'truncal' y', 'ayuda: 'desarrollo'. Se puede cambiar el nombre de la rama recién creada mediante este comando:', 'ayuda: git branch -m <nombre>'. The final output is 'Inicializado repositorio Git vacío en /home/ariel/Escritorio/proyecto1/.git/' followed by the prompt 'ariel@AMD:~/Escritorio/proyecto1\$'.

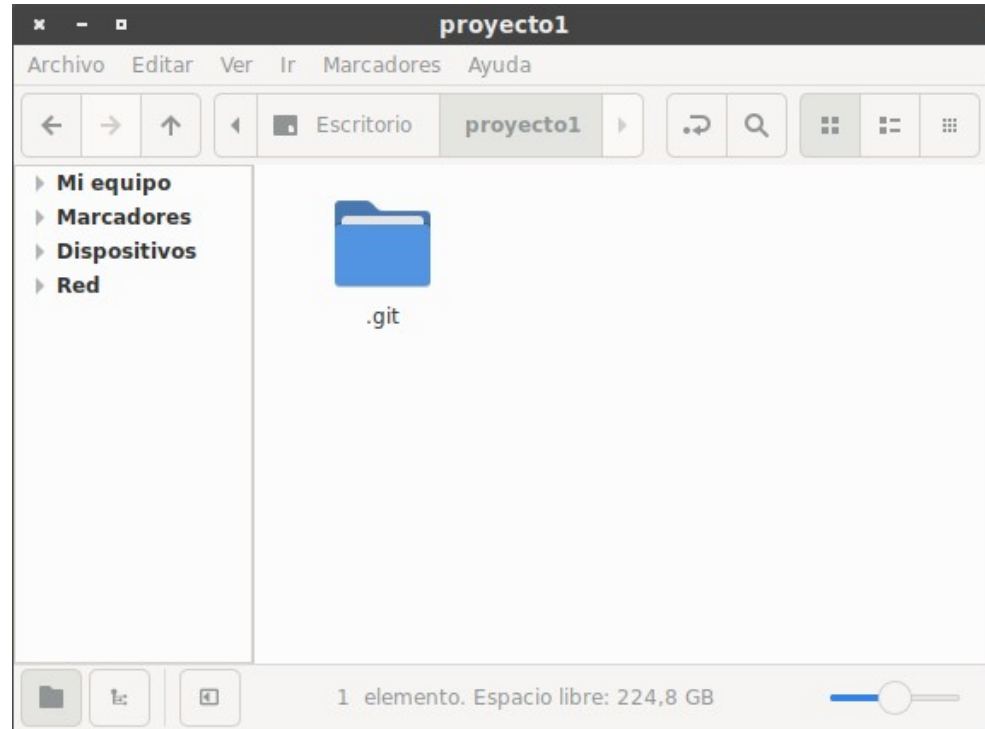
```
ariel@AMD:~$ cd Escritorio
ariel@AMD:~/Escritorio$ cd proyecto1
ariel@AMD:~/Escritorio/proyecto1$ git init
ayuda: Usando 'master' como el nombre de la rama inicial. Este nombre de rama pre-determinado
ayuda: está sujeto a cambios. Para configurar el nombre de la rama inicial para usar en todos
ayuda: de sus nuevos repositorios, que suprimirán esta advertencia, llame a:
ayuda:
ayuda: git config --global init.defaultBranch <nombre>
ayuda:
ayuda: Los nombres comúnmente elegidos en lugar de 'maestro' son 'principal', 'truncal' y
ayuda: 'desarrollo'. Se puede cambiar el nombre de la rama recién creada mediante este comando:
ayuda:
ayuda: git branch -m <nombre>
Inicializado repositorio Git vacío en /home/ariel/Escritorio/proyecto1/.git/
ariel@AMD:~/Escritorio/proyecto1$
```



# GIT | Inicializar una carpeta

Este proceso crea dentro de la carpeta una nueva carpeta oculta, con el nombre **.git** , que será el sitio en el que la herramienta almacenará toda la información necesaria para funcionar.

Si por algún motivo queremos dejar de utilizar git en ella, bastará con eliminar esa carpeta oculta.



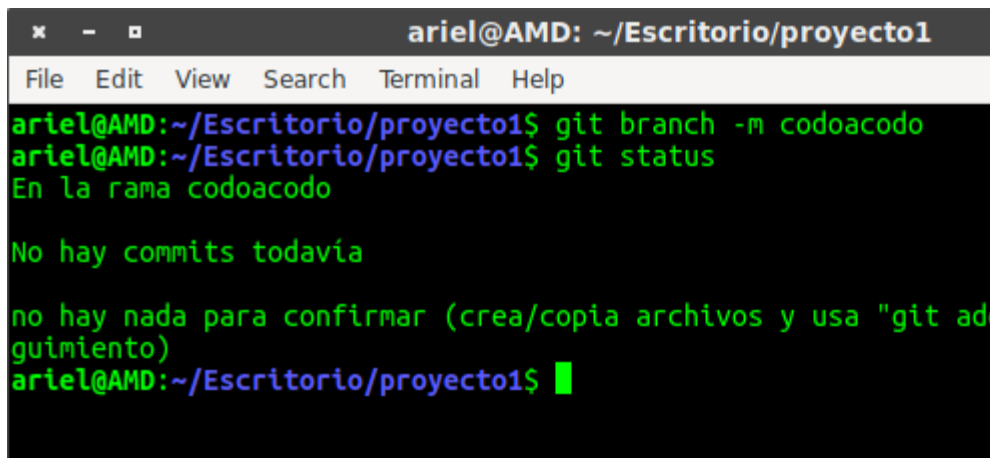
# GIT | Cambiar el nombre a la rama creada

Se puede cambiar el nombre de la rama recién creada mediante este comando:

**git branch -m <nombre\_nuevo>**

Para ver el nuevo nombre de la rama se puede utilizar

**git status**



```
ariel@AMD: ~/Escritorio/proyecto1
File Edit View Search Terminal Help
ariel@AMD:~/Escritorio/proyecto1$ git branch -m codoacodo
ariel@AMD:~/Escritorio/proyecto1$ git status
En la rama codoacodo

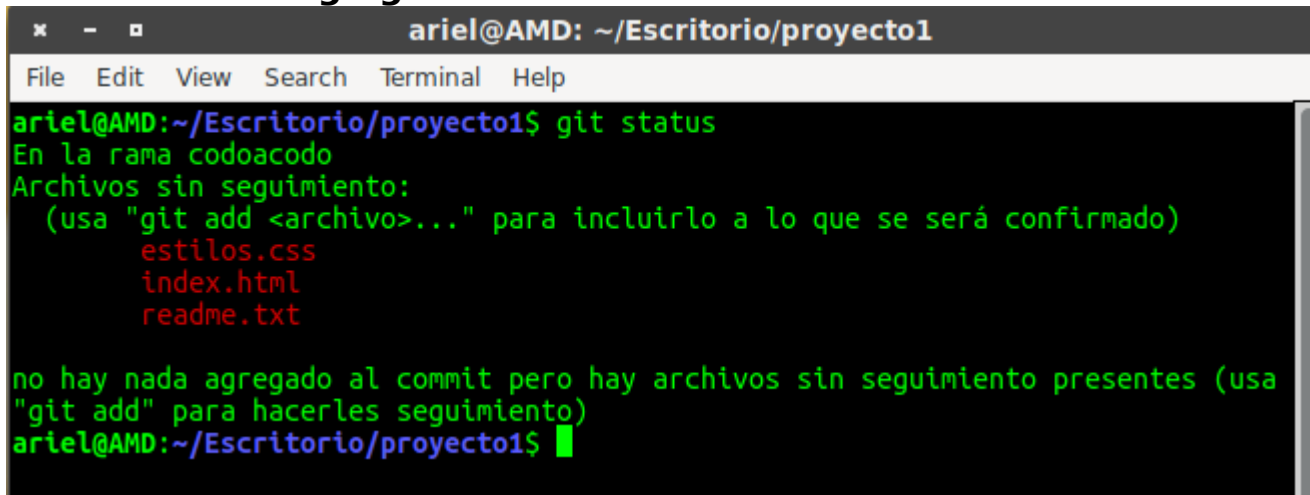
No hay commits todavía

no hay nada para confirmar (crea/copia archivos y usa "git add" para
guimientos)
ariel@AMD:~/Escritorio/proyecto1$
```

# GIT | Ver el estado de los archivos

Con **git status**, además del nombre de la rama, puedo consultar que archivos tengo en el repositorio, y cuales no están en el **staging area**

git status

A screenshot of a terminal window titled 'ariel@AMD: ~/Escritorio/proyecto1'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows the command 'ariel@AMD:~/Escritorio/proyecto1\$ git status' and its output. The output indicates the current branch is 'codoacodo' and lists three files not being tracked: 'estilos.css', 'index.html', and 'readme.txt', which are shown in red text. A message explains that these files need to be added with 'git add' to be included in the next commit. The prompt returns to 'ariel@AMD:~/Escritorio/proyecto1\$' with a cursor.

```
ariel@AMD:~/Escritorio/proyecto1$ git status
En la rama codoacodo
Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlo a lo que se será confirmado)
    estilos.css
    index.html
    readme.txt

no hay nada agregado al commit pero hay archivos sin seguimiento presentes (usa
"git add" para hacerles seguimiento)
ariel@AMD:~/Escritorio/proyecto1$
```

Los archivos que aparecen en rojo están listos para ser agregados al staging area.

# GIT | Agregar archivos al staging area

Podemos incorporar archivos que han sido creados o modificados recientemente al staging area utilizando el comando `git add`. Se puede utilizar el comando para un archivo individual, o para todos los que estén en condiciones de ser agregados:

`git add <archivo>`

Para agregarlos a todos usamos:

`git add .`

```
ariel@AMD: ~/Escritorio/proyecto1
File Edit View Search Terminal Help
ariel@AMD:~/Escritorio/proyecto1$ git add estilos.css
ariel@AMD:~/Escritorio/proyecto1$ git add index.html
ariel@AMD:~/Escritorio/proyecto1$ git add readme.txt
ariel@AMD:~/Escritorio/proyecto1$ git status
En la rama codoacodo
Cambios a ser confirmados:
  (usa "git restore --staged <archivo>..." para sacar del área de stage)
    nuevo archivo:  estilos.css
    nuevo archivo:  index.html
    nuevo archivo:  readme.txt

ariel@AMD:~/Escritorio/proyecto1$
```

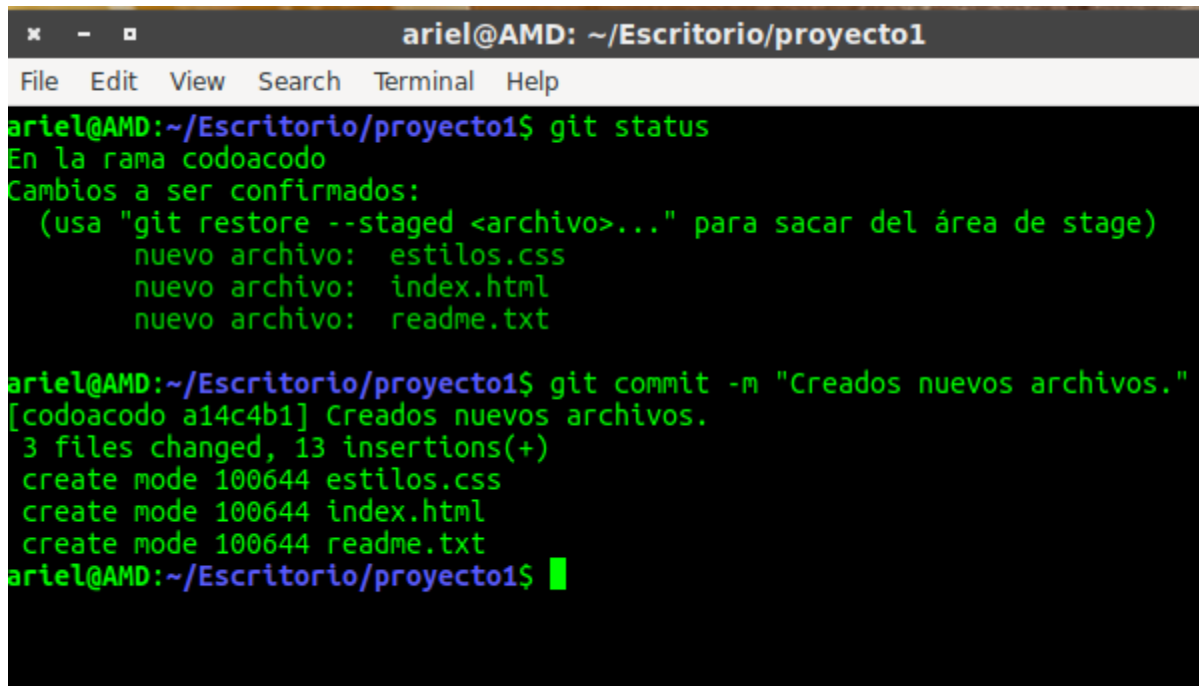
Si uso **git status** nuevamente, veo el cambio realizado.

# GIT | Crear un snapshot

Una vez indicados esos datos, ya estamos en condiciones de hacer nuestro primer **commit**, es decir, pasar archivos que hemos almacenado previamente en el staging area al repositorio local. Usamos el siguiente comando:

**git commit -m "Descripción"**

Es importante incluir una descripción relevante en cada commit, ya que será lo que git nos mostrará cuando veamos el *"historia"* de cambios realizados.

A screenshot of a terminal window titled 'ariel@AMD: ~/Escritorio/proyecto1'. The terminal shows the output of 'git status' and the execution of 'git commit -m "Creados nuevos archivos."'. The status output indicates three new files are staged: estilos.css, index.html, and readme.txt. The commit output shows the commit was successful, creating three files with 13 insertions.

```
ariel@AMD: ~/Escritorio/proyecto1
File Edit View Search Terminal Help

ariel@AMD:~/Escritorio/proyecto1$ git status
En la rama codoacodo
Cambios a ser confirmados:
  (usa "git restore --staged <archivo>..." para sacar del área de stage)
nuevo archivo:  estilos.css
nuevo archivo:  index.html
nuevo archivo:  readme.txt

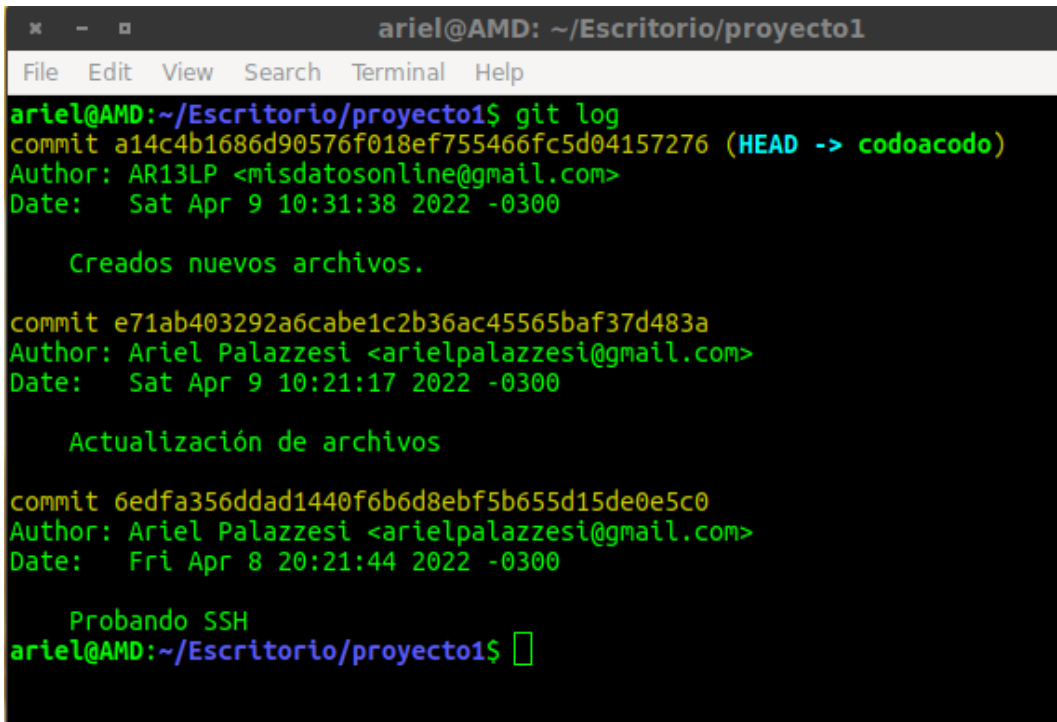
ariel@AMD:~/Escritorio/proyecto1$ git commit -m "Creados nuevos archivos."
[codoacodo a14c4b1] Creados nuevos archivos.
3 files changed, 13 insertions(+)
create mode 100644 estilos.css
create mode 100644 index.html
create mode 100644 readme.txt
ariel@AMD:~/Escritorio/proyecto1$
```

# GIT | Ver snapshots creados

Puede realizarse una consulta para obtener un historial de los cambios que se van realizando en los archivos que integran nuestro repositorio. Para ello, usamos **git log**:

## git log

Y, como vimos antes, si hago cambios en algun archivo, con **git status** veré los que han cambiado y están Listos para ser enviados al staging area o *commiteados*.



```
ariel@AMD: ~/Escritorio/proyecto1
File Edit View Search Terminal Help

ariel@AMD:~/Escritorio/proyecto1$ git log
commit a14c4b1686d90576f018ef755466fc5d04157276 (HEAD -> codoacodo)
Author: AR13LP <misdatosonline@gmail.com>
Date: Sat Apr 9 10:31:38 2022 -0300

    Creados nuevos archivos.

commit e71ab403292a6cabe1c2b36ac45565baf37d483a
Author: Ariel Palazzesi <arielpalazzesi@gmail.com>
Date: Sat Apr 9 10:21:17 2022 -0300

    Actualización de archivos

commit 6edfa356ddad1440f6b6d8ebf5b655d15de0e5c0
Author: Ariel Palazzesi <arielpalazzesi@gmail.com>
Date: Fri Apr 8 20:21:44 2022 -0300

    Probando SSH
ariel@AMD:~/Escritorio/proyecto1$
```

# GIT | Ver cambios en un archivo

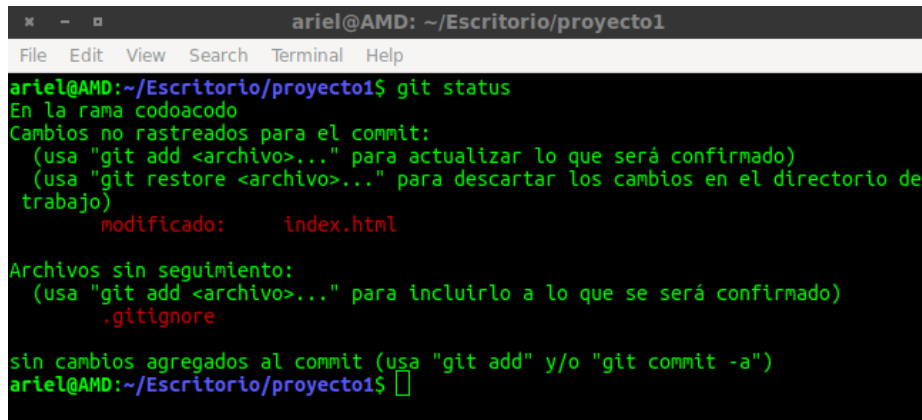
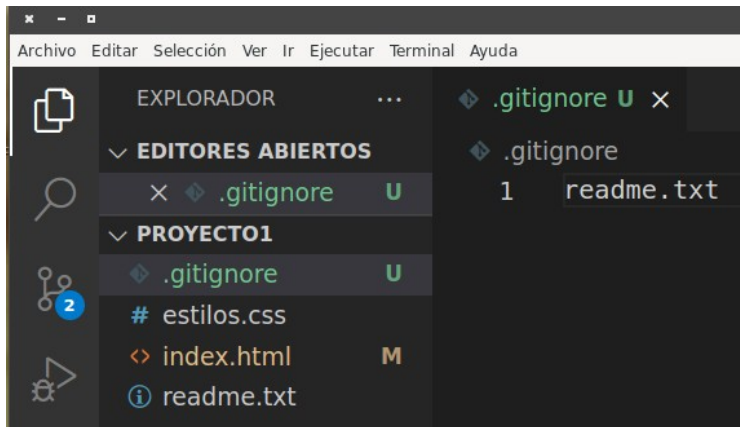
Una característica muy potente de git es la posibilidad de visualizar los cambios que se han producido en un archivo. Con **git diff** podemos ver que líneas se agregaron, eliminaron o modificaron entre la versión actual del mismo y la que tenemos almacenada:

**git diff <archivo>**

```
ariel@AMD: ~/Escritorio/proyecto1
File Edit View Search Terminal Help
ariel@AMD:~/Escritorio/proyecto1$ git diff index.html
diff --git a/index.html b/index.html
index cee901b..5ede1ae 100644
--- a/index.html
+++ b/index.html
@@ -11,6 +11,7 @@
     <p>Sistema de control de versiones</p>
     <br>
     <p>Hemos agregado un nuevo parrafo y usado <b>git commit</b></p>
-    <p>Hemos agregado otro parrafo y usado <b>git commit</b></p>
+    <p>Hemos agregado otro parrafo y usado <b>git commit</b></p>
+    <p>Hemos agregado otro parrafo, pero no hecho el <b>git commit</b></p>
   </body>
 </html>
\ No newline at end of file
ariel@AMD:~/Escritorio/proyecto1$
```

# GIT | Ignorar archivos o carpetas

En ocasiones no necesitamos que la totalidad de los archivos que forman parte de nuestro proyecto sean gestionados por **git**. En estos casos, podemos hacer una lista con los archivos y/o carpetas a excluir, y guardarla en un archivo de texto que tenga el nombre **.gitignore**. Se debe poner un nombre por linea, y todos los archivos alli listados seran ignorados por git.



Si ejecutamos un **git status**, los archivos de la lista ya no aparecen como pendientes de ser almacenados.





# GIT | Descartar cambios

Puede darse el caso que interese descartar cambios que hayamos realizado sobre uno o más archivos. Esto puede lograrse de tres formas diferentes:

1. Descartar cambios sobre un archivo en particular, sin guardarlos:


**git checkout:** descarta los cambios sobre el archivo y vuelve a la versión que esté en el último commit del repositorio.

2. Descartar todos los cambios, sin guardarlos:

**git reset --hard:** descarta todos los cambios no commiteados, sin guardarlos. Vuelve a las versiones del último commit descargado desde el repositorio remoto.

3. Descartar todos los cambios, guardándolos:

**git stash:** descarta todos los cambios no commiteados, guardándolos para poder recuperarlos en un futuro.



# GIT | Recuperar cambios que han sido descartados

Los cambios que han sido descartados con **git stash** pueden ser recuperados.

**git stash list:** lista todos los “puntos de restauración” que hemos generado al utilizar el comando “stash” para descartar los cambios y guardarlos. El más reciente tienen el índice 0 (cero).

**git stash show -p <stash-name>:** Permite ver los cambios que se encuentran guardados en un stash en particular.

**git stash apply <stash-name>:** Permite recuperar los cambios desde un stash en particular (no se elimina el punto de restauración).

**git stash drop <stash-name>:** Permite eliminar un “punto de restauración” de forma definitiva, y la pila de cambios stasheados se reordenará. Hay que ser cuidadosos, ya que ésta acción es irreversible y no se solicita confirmación por parte del usuario para llevarla a cabo.

# GIT | Ramas (branch)

Cómo vimos, en cualquier momento podemos crear una nueva rama en nuestro proyecto. Los comandos implicados en esta operación son:

**git branch** : muestra la(s) ramas que componen el proyecto.

**git branch <nombre de la rama>** : crea una nueva rama con el nombre indicado

**git checkout <nombre de la rama>** : cambio a otra rama para trabajar en ella.

Al cambiar de una rama a otra, si miramos el contenido de la carpeta de nuestro proyecto veremos que los archivos que no pertenecen a la misma están ocultos, y el contenido de los que han sido modificados es el correspondiente a la rama actual. Si volvemos a cambiar de rama, el contenido visible de la carpeta volverá a modificarse para que dispongamos de las versiones correctas.



# GIT | Sincronizando con GitHub (push)

A **git push** se le puede considerar un comando de carga, ya que permite subir los **commits** realizados en nuestro repositorio local un repositorio remoto como **GitHub**. Una vez allí, estos pueden ser descargados por el resto del equipo de trabajo.

**git push <remote> <branch>**

De este modo se crea una rama local en el repositorio de destino.

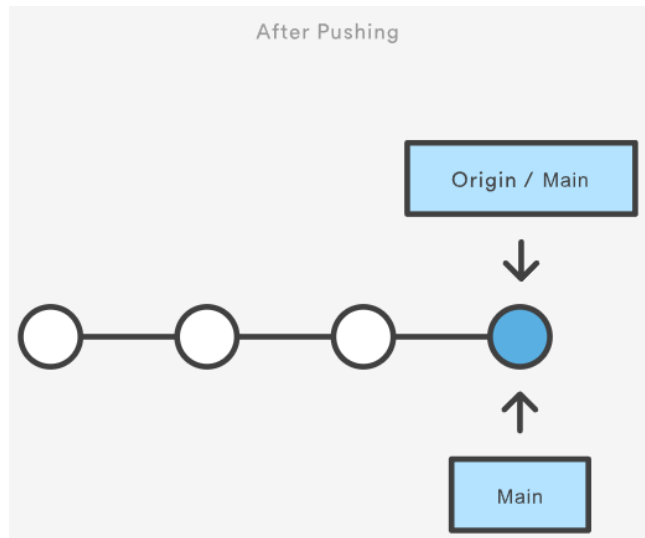
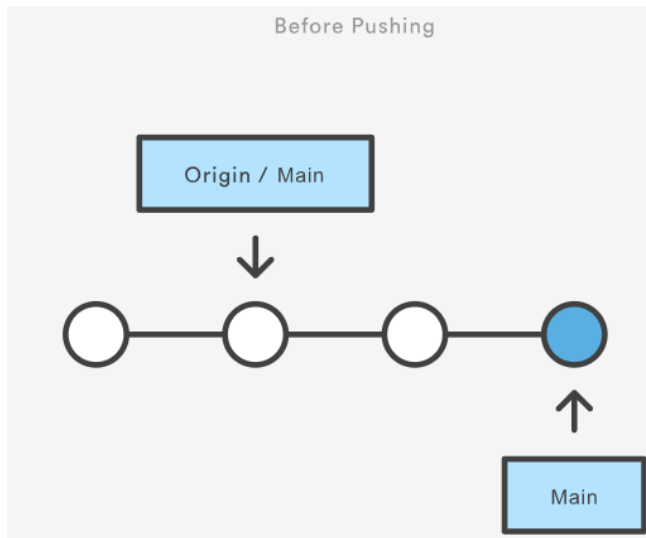
**git push <remote> --all**

Envía todas las ramas locales a una rama remota especificada.

Una vez movidos los conjuntos de cambios se puede ejecutar un comando **git merge** en el destino para integrarlos.

(Aprende más en <https://www.atlassian.com/es/git/tutorials/syncing/git-push> )

# GIT | Sincronizando con GitHub (push)



# GIT | Sincronizando con GitHub (pull)

El comando **git pull** se emplea para extraer y descargar contenido desde un repositorio remoto y actualizar al instante el repositorio local para reflejar ese contenido. El comando **git pull** es, en realidad, una combinación de dos comandos, **git fetch** seguido de **git merge**.

## **git pull <remote>**

Recupera la copia del origen remoto especificado de la rama actual y fusi3nala de inmediato en la copia local.

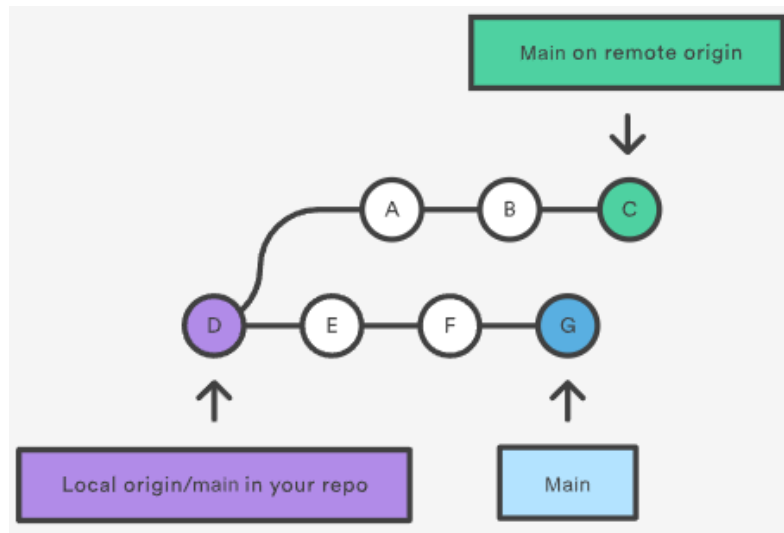
## **git pull --no-commit <remote>**

Recupera la copia del origen remoto, pero no crea una nueva conformaci3n de fusi3n.

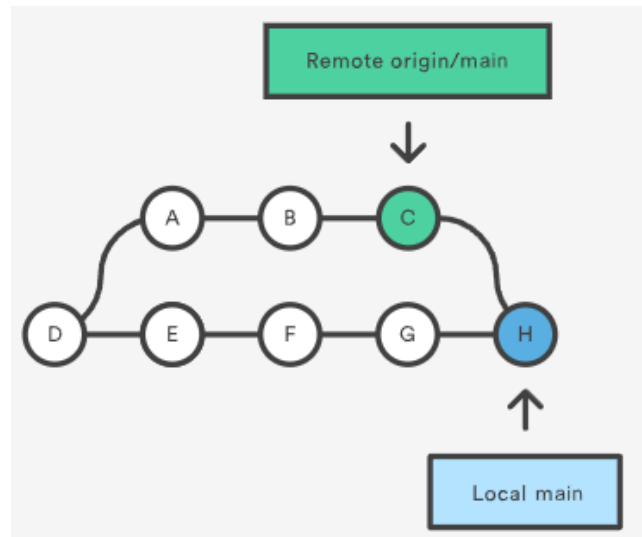
(Aprende m1s en <https://www.atlassian.com/es/git/tutorials/syncing/git-pull> )

# GIT | Sincronizando con GitHub (pull)

Supongamos que tenemos un repositorio con una rama principal y un origen remoto:



El proceso de incorporación de cambios creará otra confirmación de fusión local que incluya el contenido de las nuevas confirmaciones remotas divergentes.



# GIT | Conflictos

El mecanismo provisto por git no está exento de posibles conflictos. Puede darse el caso de que estemos trabajando sobre mismo archivo en el cual que está trabajando otra persona, y si está hace un **push** antes que nosotros, cuando intentemos realizar nuestro **push** aparecerá un conflicto (merge).

Un merge se genera cuando dos o más commits contienen cambios sobre las mismas líneas de código de los mismos archivos. En ocasiones, git no puede resolver la situación automáticamente.

10	<style>	10	<style>
11	code {	11	code {
12-	background-color: yellow;	12+	background-color: red;
13	font-weight: bold;	13	font-weight: bold;
14	}	14	}
15	/*containerh: Es el elemento padre que	15	/*containerh: Es el elemento padre que
16	.flex-containerh {	16	.flex-containerh {
17	display: flex; /* Esta propiedad d	17	display: flex; /* Esta propiedad d
18	flex-direction: row;	18	flex-direction: row;
19	background-color: yellowgreen;	19	background-color: yellowgreen;
20	}	20	}
21		21	



# GIT | Conflictos

En el ejemplo de la diapositiva anterior, en otra rama el color de fondo se habia fijado en “green”. Al realizar el pull request aparece el conflicto, y se nos pide que lo solucionemos manualmente:

```
9
10     <style>
11         code {
12     <<<<<<< pruebaFile9
13             background-color: green;
14     =====
15             background-color: red;
16     >>>>>>> main
17             font-weight: bold;
18         }
19     /*containerh: Es el elemento pa
20     .flex-containerh {
```

Lo resolvemos eliminando la(s) linea(s) que no sean pertinentes y lo marcamos como “resolved”. Hacemos el **commit merge**, y el archivo finalmente quedará con los cambios elegidos:

```
10     <style>
11         code {
12             background-color: green;
13             font-weight: bold;
14         }
15     /*containerh: Es el elemento pa
16     .flex-containerh {
17         display:flex; /* Esta pro
```

# GIT | Resumen comandos básicos

## COMANDOS BÁSICOS DE git



**GIT es un sistema de control de versiones** que nos ayuda a llevar el historial completo de modificaciones de un proyecto.

Todo desarrollador sin importar el lenguaje **debe dominar Git**.  
**Prof. Beto Quiroga**

### **GIT INIT**

Inicia un nuevo repositorio.

### **GIT ADD**

Añade un archivo a la zona de montaje. **git add \*** añade uno o más archivos a la zona de montaje.

### **GIT LOG**

Se utiliza para listar el historial de versiones de la rama actual.

### **GIT RESET**

Descompone el archivo, pero conserva el contenido del mismo.

### **GIT CLONE**

Clona un repositorio existente.

### **GIT CONFIG**

Establece el nombre del autor, el correo y demás **parámetros** que Git utiliza por defecto.

### **GIT STATUS**

Enumera todos los archivos que deben ser confirmados.

### **GIT DIFF**

Muestra las diferencias de archivo que aún no se ponen en escena.

¿Qué comandos faltó? Haremos la 2da parte con tus sugerencias

 [ed.team/cursos/git](https://ed.team/cursos/git)



## COMANDOS BÁSICOS DE GIT

HECHO CON MUCHO AMOR POR @AMADOONLINE ★

### GIT PUSH

CON ESTE COMANDO FINALMENTE DESPACHAMOS NUESTROS CAMBIOS Y LOS MANDAMOS AL REPOSITORIO REMOTO, DONDE TODAS LAS PERSONAS QUE COLABORAN PUEDEN ACCEDER.

EN UN SOLO PUSH SE PUEDEN ENVIAR VARIOS COMMITS



### GIT PULL

ASÍ COMO "git push" SIRVE PARA ENVIAR CAMBIOS AL REMOTO, "git pull" NOS PERMITE TRAER CAMBIOS A NUESTRO REPOSITORIO LOCAL.

¡BIENVENIDO A CASA, GATTITO!

### GIT BRANCH Y GIT MERGE

"git branch" NOS PERMITE CREAR RAMAS (BRANCHES) QUE NOS PERMITE TRABAJAR EN UN CONTEXTO SEPARADO DEL ORIGINAL, COMO SI MANDÁRAMOS A NUESTRO GATTITO A OTRA LINEA TEMPORAL PARALELA.

ESTO NOS PERMITE TRABAJAR CON LIBERTAD SABRIENDO QUE NO VAMOS A ROMPER NADA. CUANDO EL GATTITO ESTÁ LISTO PARA SER COMPARTIDO NUEVAMENTE PODEMOS "UNIRLO" A LA RAMA ORIGINAL UTILIZANDO "git merge"



### CONFLICTOS

GIT SUELE SER MUY BUENO INTERPRETANDO NUEVOS CAMBIOS, PERO ¿QUE PASA SI DOS PERSONAS PARTEN DE LA MISMA VERSIÓN Y HACEN CAMBIOS EN EL MISMO ARCHIVO? EN ESTE CASO SE PUEDE GENERAR UN "CONFLICTO".



CUANDO ESTO OCURRE DEBEMOS INDICARLES MANUALMENTE A QUE DARLE PRIORIDAD. PUEDE SER NUESTRO CAMBIO, EL OTRO O UNA MEZCLA DE AMBOS.



# GitLab vs. GitHub

- Actividad Práctica: investigar las diferencias.
  - Referencia: <https://www.redeszone.net/2019/01/10/github-vs-gitlab-diferencias/>



# GIT y GitHub | Tutoriales

- **Fundamentos de GIT:** <https://bluuweb.github.io/tutorial-github/guia/fundamentos.html>
- **GitHub:** <https://bluuweb.github.io/tutorial-github/guia/github.html>
- **Comandos explicados:** <https://gist.github.com/dasdo/9ff71c5c0efa037441b6>



# GIT y GitHub | Material multimedia

- **Videos del Profesor Alejandro Zapata (Coordinador y Docente de Codo a Codo):**  
<https://www.youtube.com/watch?v=ptXiQwE535s&list=PLoCpUTIZIYORkDzYwdunkVf-KlqGjyoot>
- **GIT y GitHub (tutorial en español). Inicio Rápido para Principiantes:**  
[https://www.youtube.com/watch?v=hWglK8nWh60&list=PLPl81lqbj-4l8i-x2b5\\_MG58tZfgKmJls](https://www.youtube.com/watch?v=hWglK8nWh60&list=PLPl81lqbj-4l8i-x2b5_MG58tZfgKmJls)
- **¿Cómo trabajar con Git desde Visual Studio Code?:** <https://youtu.be/AYbgqmyg7dk>
  - Nota: con Visual Studio Code también se puede hacer commits, push, resolver conflictos, crear ramas y mucho más. Prácticamente todo lo que se hace desde la línea de comandos lo podés hacer desde una interfaz gráfica. En el video se indica cómo hacerlo. Al final se recomienda un plugin que hay que instalar si se quiere trabajar con Git desde Visual Studio Code.
  - *Importante: se puede utilizar una interfaz gráfica para trabajar con Git, pero es importante saber qué es lo que pasa detrás de cada clic que uno hace. Antes, hay que aprender los fundamentos de GIT.*

