<codoacodo/>

Agencia de Aprendizaje a lo largo de la vida

# FULL STACK PYTHON
# Clase 12

## GIT

BA

# GIT

# Les damos la bienvenida

Vamos a comenzar a grabar la clase

## Clase 11

### Bootstrap

- ¿Qué es un framework?
- Frameworks CSS.
- Conceptos básicos de Bootstrap.
- Componentes y funcionalidades.
- Grid Layout.
- Bootstrap Themes.

## Clase 12

### GIT

- Introducción a GIT y GitHub.
- Comandos básicos.
- Creación de repositorios y ramas.

## Clase 13

### Introducción a Javascript

- ¿Qué es y para qué se usa?
- Conceptos generales. Sintaxis básica.
- Variable, ¿qué es y cómo declararla? Tipos.
- Asignación y cambio del valor.
- Operadores aritméticos.
- Conversión a entero y flotante.

# ¿Qué es GIT?

**Git** es un sistema de control de versiones distribuido, diseñado por Linus Torvalds. Está optimizado para guardar cambios de forma incremental. Permite contar con un historial, regresar a una versión anterior y agregar funcionalidades.

Además, es capaz de llevar un registro de los cambios que otras personas realicen en los archivos, gracias a **GitHub**, donde podemos almacenar y compartir nuestros proyectos de forma gratuita. **Es multiplataforma**, es decir, es compatible con Linux, MacOS y Windows. En la máquina local se encuentra Git, se utiliza bajo la terminal o línea de comandos y tiene comandos como **merge**, **pull**, **add**, **commit** y **rebase**, entre otros.

Con **Git** se obtiene una mayor eficiencia usando archivos de texto plano, ya que con archivos binarios no puede guardar solo los cambios, sino que debe volver a grabar el archivo completo ante cada modificación, por mínima que sea, lo que hace que incremente el tamaño del repositorio.

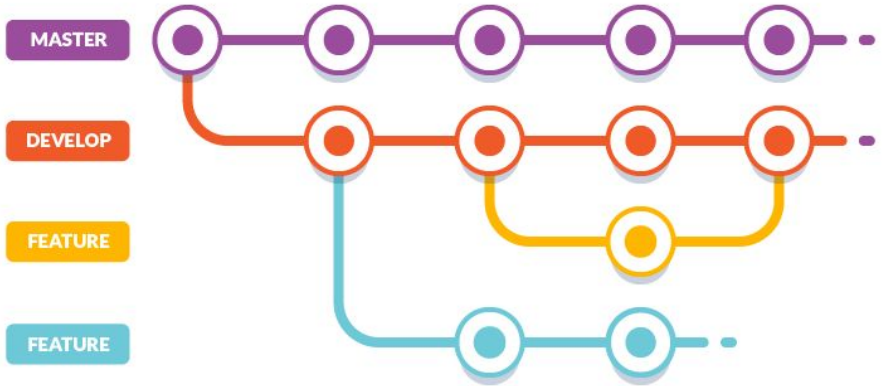
# GIT | Definición

Es 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. 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.

# GIT | ¿Cómo funciona?

Git almacena instantáneas de un mini sistema de archivos. Cada vez que confirmamos un cambio, Git toma una "foto" del aspecto del proyecto en ese momento y crea una referencia a esa instantánea. Si un archivo no cambió Git sólo crea un enlace a la imagen anterior idéntica que tiene almacenada.



# GIT | Terminología

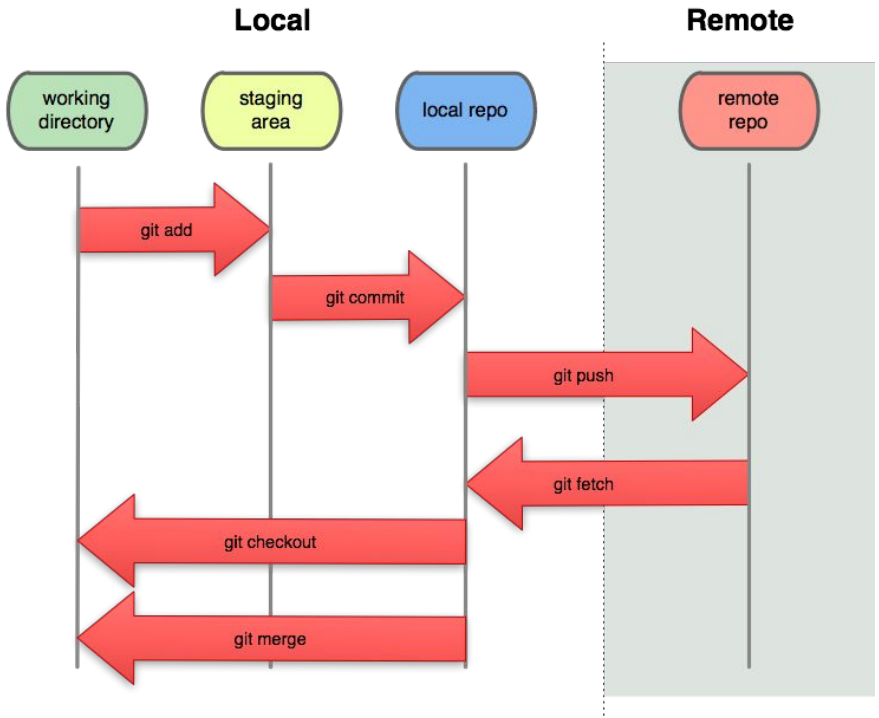
- **Repositorio:** es la carpeta principal donde se encuentran almacenados los archivos que componen el proyecto. El directorio contiene *metadatos* gestionados por **Git**, de manera que el proyecto es configurado como un repositorio local.
- **Commit:** un *commit* es el estado de un proyecto en un 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**.

# GIT | Terminología

- **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 comunicamos el código que incluye los cambios, es revisado, comentado y aprobado para darle *merge*. En el contexto de GIT, *merge* significa unir dos trabajos, en este caso la rama *branch* con la rama *master*.

# GIT | Flujo de trabajo

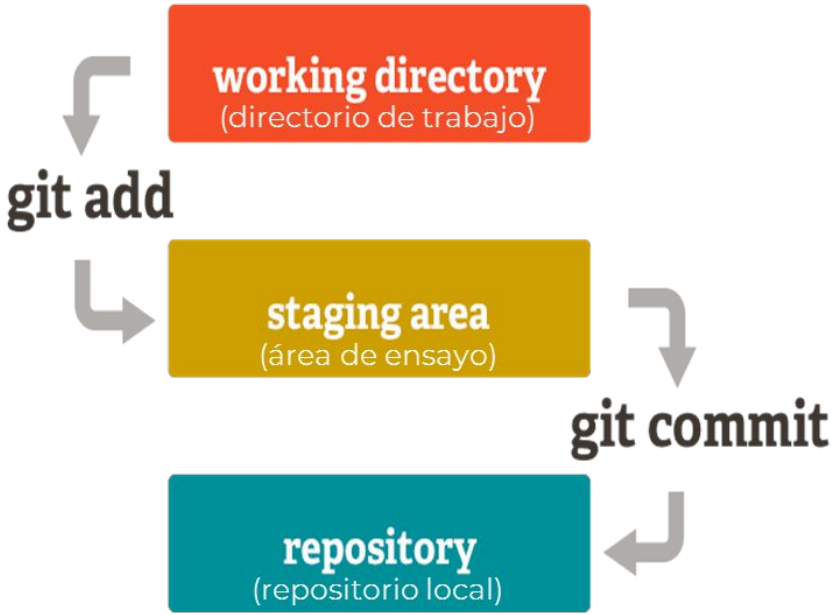
Git registra en nuestro directorio local los cambios que se produzcan en los archivos o código, cada vez que se lo indiquemos. De esta forma podemos “viajar en el tiempo” revirtiendo cambios o restaurando versiones de código. Esto puede hacerse localmente o de forma remota (servidor externo).



# GIT | Estados

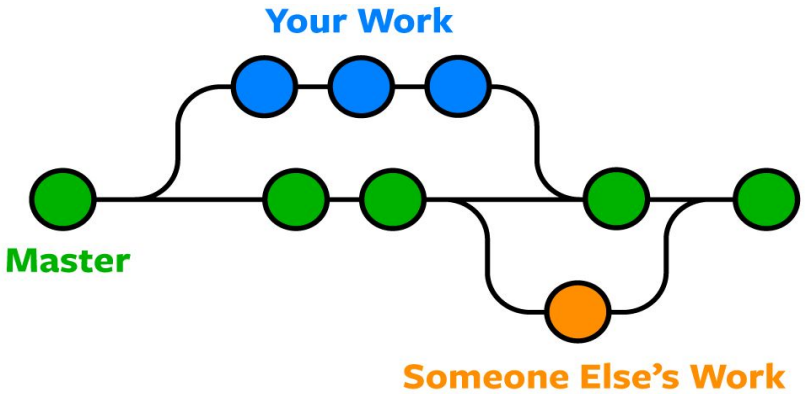
**Working directory** es nuestro directorio de trabajo. Cada vez que queremos agregar un archivo al **staging area**, usamos **git add**.

Luego, cuando queremos establecer un punto de restauración, ejecutamos **git commit**, y los archivos son actualizados en el repositorio (**repository**).



# 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***



# Comandos básicos del sistema operativo

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

| Comando | Descripción |
|---|---|
| **pwd** | Ver en qué carpeta estamos ubicados. |
| **ls** | Mostrar el contenido de la carpeta. Agregando **-l** o **-lh** obtenemos más detalle y con **-la** muestra los archivos ocultos. |
| **clear o CTRL+L** | Limpiar la pantalla. |
| **cd** carpeta | Ingresar en la subcarpeta (*cd: change directory*). |

# Comandos básicos del sistema operativo

| Comando | Descripción |
|---|---|
| **mkdir** *nombrecarpeta* | Nos permite crear una carpeta. |
| **rm** *archivo.ext* | Elimina el archivo, donde ***ext*** es la extensión. |
| **rm** *–r nombredecarpeta/* | Elimina la carpeta. |
| **mv** *nombreoriginal.ext nombrenuevo.ext* | Cambia el nombre a un archivo (agregando la extensión) o a una carpeta. |
| **exit** | Salimos de la terminal. |

# GIT | Registrar mis datos en git

Antes de realizar algunas de las operaciones más importantes de **git**, necesitamos indicar cuál es nuestra dirección de 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"`

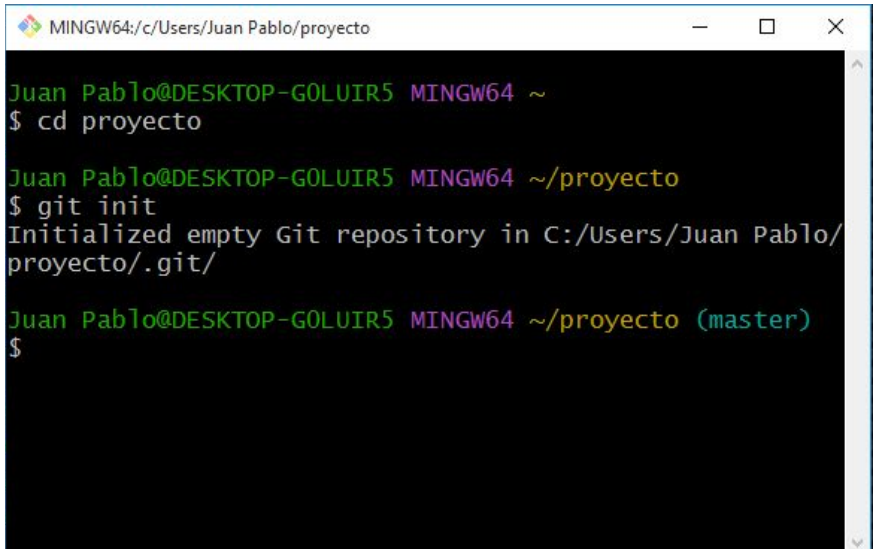
- Proporcionar el nombre del propietario:
  `git config --global user.name "NombreDelUsuario"`

- Consultar los datos que tenemos registrados:
  `git config --global -e`

# 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`.

Esto genera por defecto una rama llamada “master”.



# GIT | Ver y cambiar el nombre a la rama actual

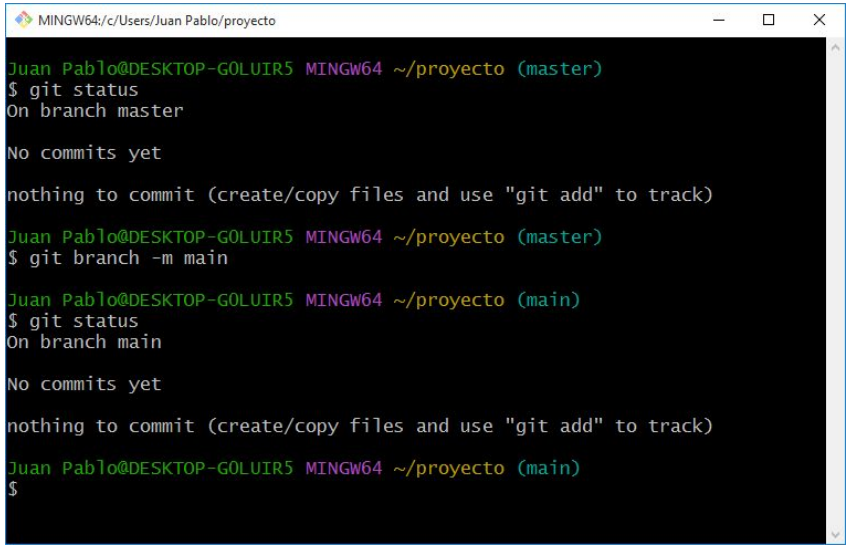
Para ver el estado de la rama actual y su contenido utilizamos:

```
git status
```

Es posible cambiar el nombre a la rama actual con el comando:

```
git branch -m <nombre>
```

Actualmente se está realizando una campaña para utilizar como rama principal ***main*** en lugar de ***master***.

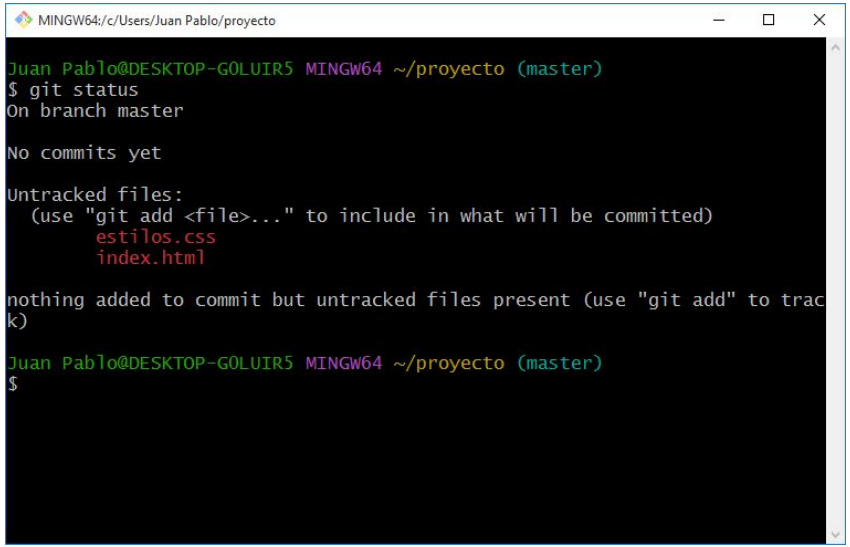


# GIT | Ver el estado de los archivos

Podemos ver el estado de los archivos que contiene la carpeta controlada por git usando

`git status`.

En color rojo vemos archivos agregados/modificados, que aún no se han incluido en el **staging area**.



# GIT | Agregar archivos al staging area

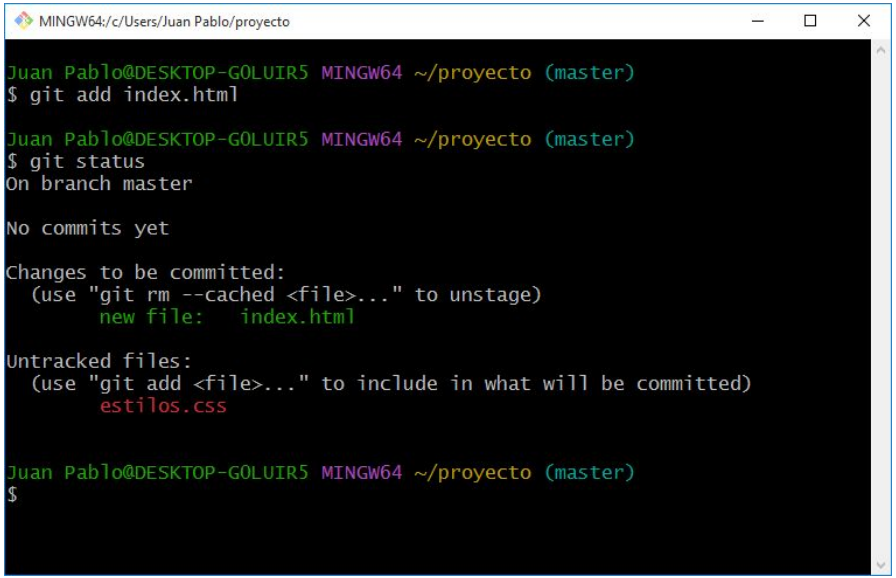
`git add` incorpora los archivos que han sido creados o modificados recientemente al **staging area.** Se puede agregar un archivo con:

`git add <archivo>`

O todos con:

`git add .`

Los archivos listos para ser *commiteados* aparecen ahora en verde.



# GIT | Agregar archivos al repositorio local

Los archivos del ***staging area*** se envían al repositorio local utilizando el comando `git commit -m"comentario"`, donde "comentario" es una explicación de los cambios implicados. Es importante incluir una descripción relevante en cada commit, ya que será lo que git nos mostrará cuando veamos el "historial" de cambios realizados.

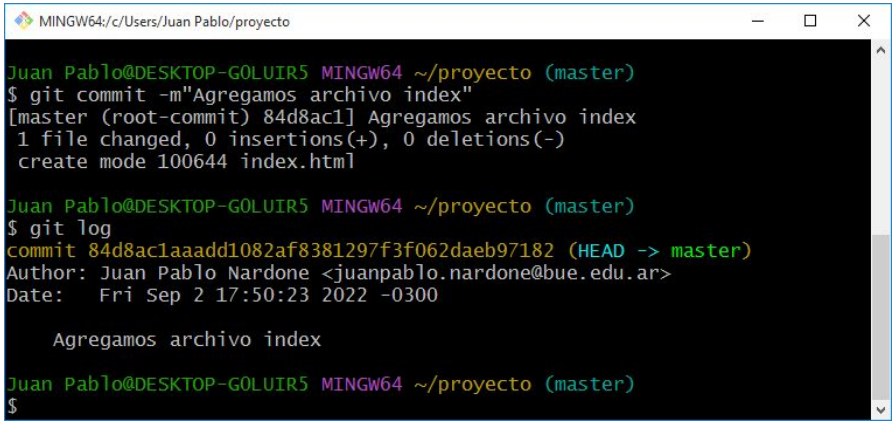
Cada vez que realizamos un `commit`, git genera un punto de restauración al cual es posible volver en cualquier momento.

Si no incluimos el comentario (`git commit`), y hemos configurado un editor de texto, git abre una ventana para que lo hagamos. Grabamos, cerramos, y el commit se habrá realizado.

# GIT | Ver snapshots creados

Luego de hacer el commit, si queremos obtener un historial de los cambios realizados en los archivos que integran nuestro repositorio usamos `git log`

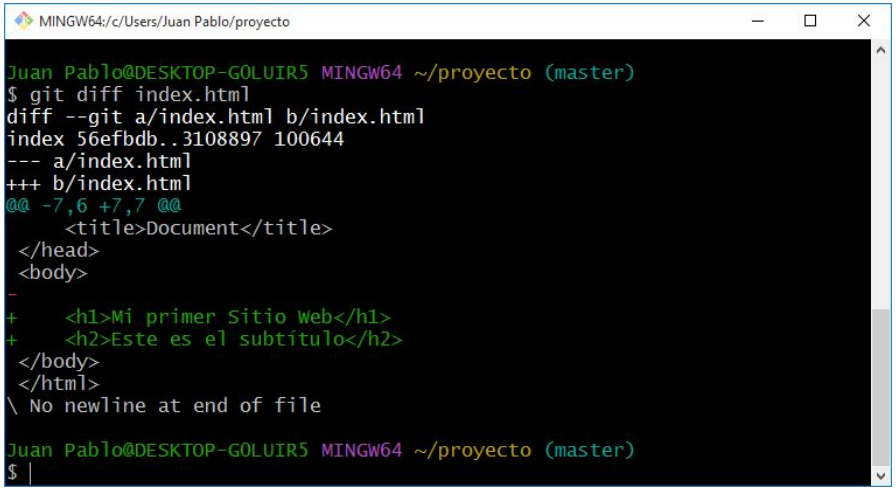
El ciclo de trabajo, entonces, consiste en editar los archivos, enviarlos al ***staging area***, y cuando estamos listos, hacemos un commit. Repetimos este proceso las veces que sea necesario.



# 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 (verde), eliminaron (rojo) o modificaron (amarillo) entre la versión actual del mismo y la del último commit:

```
git diff <archivo>
```



# GIT | Descartar cambios

Existen tres maneras de descartar cambios que hayamos realizado:

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

**git reset  --hard**: descarta **todos** los cambios no commiteados, *sin guardarlos*. Vuelve a las versiones del último commit realizado.

**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 con “stash”. El más reciente tiene el índice 0 (cero).

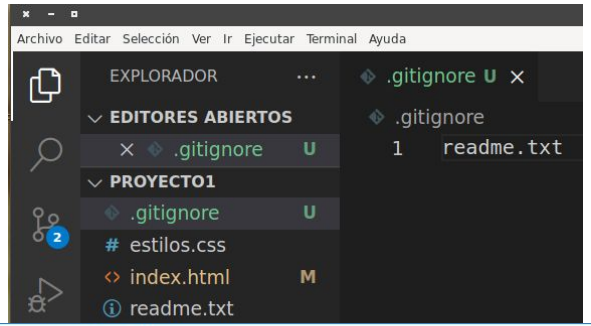
**git stash show –p <stash-name>**: Muestra los cambios que se encuentran guardados en un stash en particular.

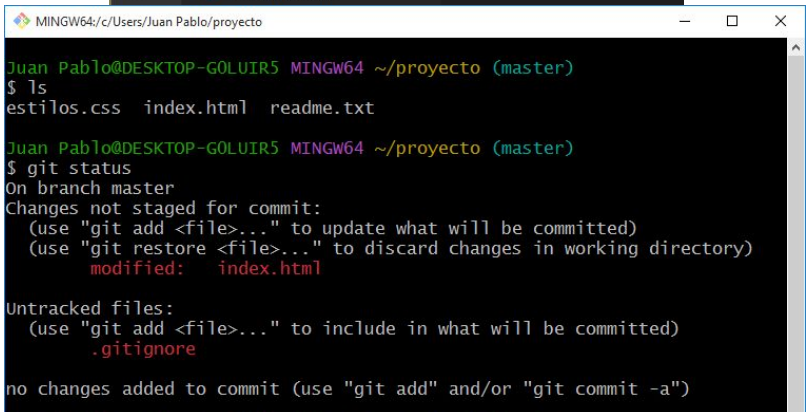
**git stash apply <stash-name>**: Recupera los cambios desde un stash en particular (no se elimina el punto de restauración).

**git stash drop <stash-name>**: Elimina un “punto de restauración” de forma definitiva, y la pila de cambios stasheados se reordenará. Esta acción es irreversible.

# GIT | Ignorar archivos o carpetas

Cuando no necesitamos que todos los archivos de nuestro proyecto sean gestionados por git 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 línea, y todos los archivos allí listados serán ignorados por git.



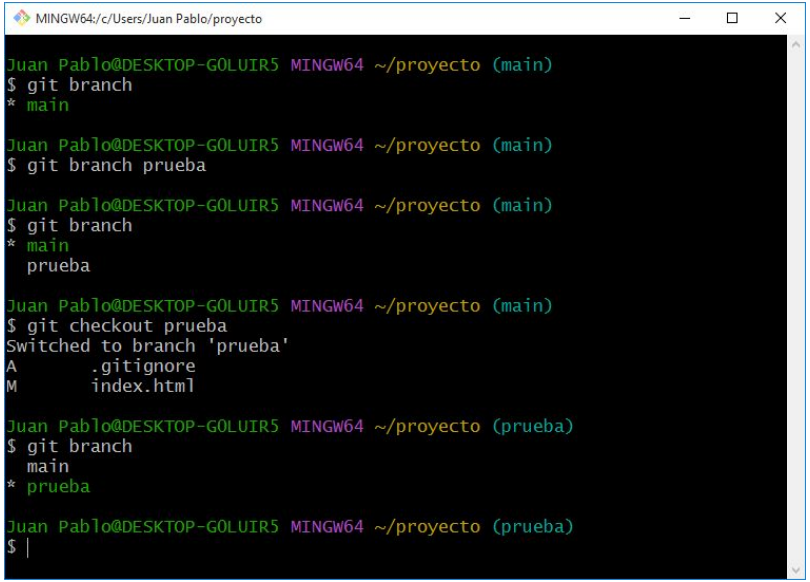
# GIT | Ramas (branch)

Podemos crear una nueva rama en nuestro proyecto, mediante estos comandos:

**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.



# GIT | GitHub

GitHub es una plataforma de repositorios remotos. Además de permitir ver el código y descargar diferentes versiones de una aplicación, la plataforma también conecta desarrolladores para que puedan colaborar en un mismo proyecto.

Podemos sincronizar repositorios locales con repositorios remotos, clonar en nuestra PC repositorios públicos de terceros, utilizar la plataforma como un mecanismo de backup de nuestros repositorios locales. Para poder subir gratis los proyectos deberán ser de código abierto. Ofrece también una herramienta de revisión de código, en la que se pueden dejar anotaciones para mejorar y optimizar el código.

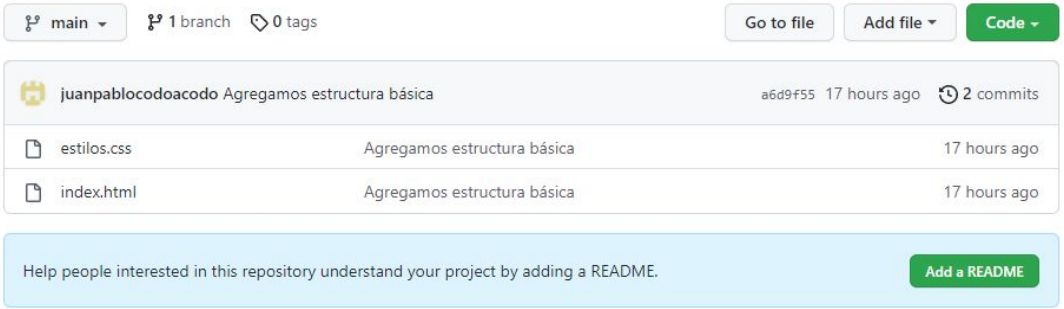
Ingresar

# GIT | Crear y vincular repositorio remoto

Creamos un repositorio **(1)**, le damos un nombre **(2)** y lo enlazamos con nuestro repositorio local mediante el comando que nos muestra la plataforma **(3)**.

# GIT | Sincronizando con GitHub (push)

El comando **git push -u origin <rama>** sincroniza una rama del repositorio local con el repositorio externo. Necesitamos el nombre de usuario en GitHub y un token que se obtiene desde “Usuario -> Settings -> Developer settings -> Personal tokens” (cómo crear un token personal).
Ejecutado este comando, el repositorio se sincroniza y podemos verlo en GitHub:



# GIT | Sincronizando con GitHub (push)

**git push** es un comando de carga que permite subir los commits realizados en nuestro repositorio local a GitHub. Una vez allí, estos pueden ser descargados por el resto del equipo de trabajo.
Para crear una rama local en el repositorio de destino utilizamos:

```
git push <remote> <branch>
```

Si queremos enviar todas las ramas locales a una rama remota especificada.

```
git push <remote> --all
```

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

# GIT | Sincronizando con GitHub (push)

En caso de querer sincronizar nuestro trabajo con el de otro usuario, en forma local, podemos clonar su repositorio:

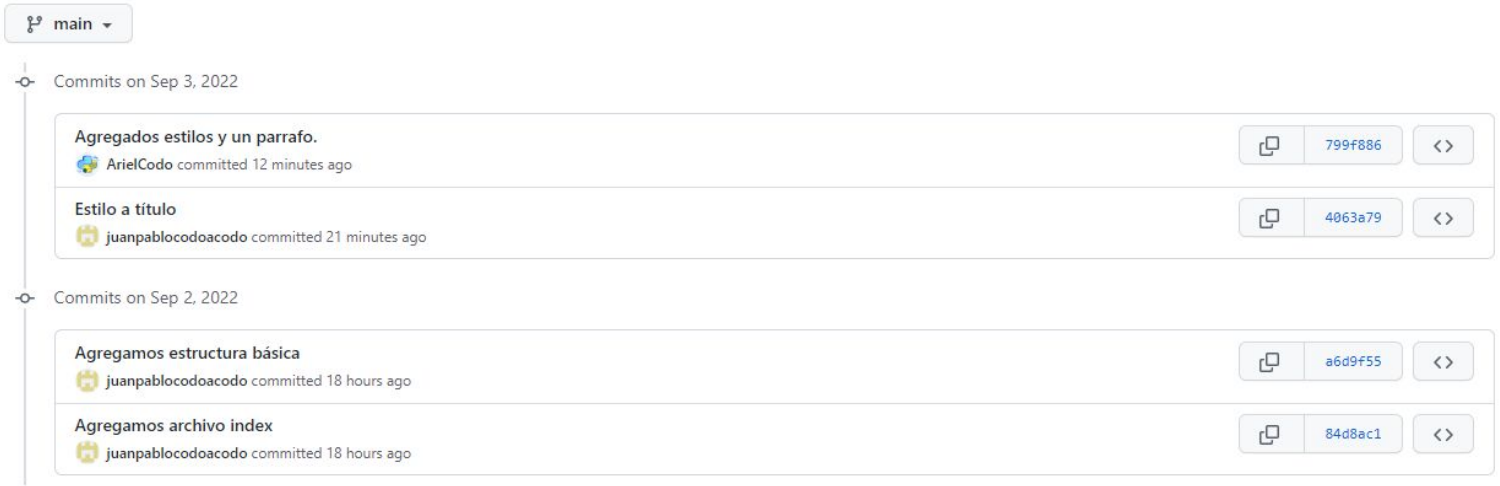
**`git clone <url repositorio externo>`**

Hacer los cambios necesarios, *commitearlos*, y luego, con *push*, enviarlos nuevamente al repositorio remoto. En este caso, en el *push* usaremos nuestro usuario y el token del propietario del repositorio.

```
ariel@AMD:~$ git clone https://github.com/JuanPabloNardone/proyectoWeb.git
Clonando en 'proyectoWeb'...
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 11 (delta 1), reused 11 (delta 1), pack-reused 0
Recibiendo objetos: 100% (11/11), listo.
Resolviendo deltas: 100% (1/1), listo.
```

# GIT | Sincronizando con GitHub (push)

De esta forma vemos en GitHub los commits hechos hasta el momento, tanto del propietario del repositorio como del colaborador:



# 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.

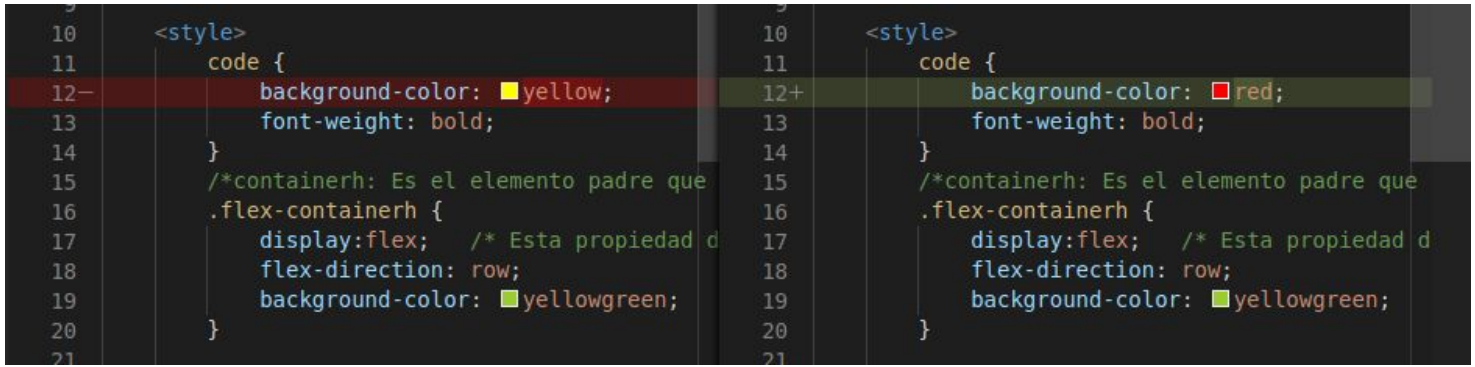
**`git pull <remote>`** Recupera la copia del origen remoto especificado de la rama actual y la fusiona de inmediato en la copia local.

**`git pull --no-commit <remote>`** Recupera la copia del origen remoto, pero no crea una nueva conformación de fusión.

# GIT | Conflictos

El mecanismo provisto por git no está exento de posibles conflictos.

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.



# GIT | Conflictos

En el ejemplo de la diapositiva anterior, en otra rama el color de fondo se había 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 {
```

# GIT | Conflictos

Lo resolvemos eliminando la(s) línea(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
```

# Material extra

# Comandos básicos de Git


Ver más grande


Ver más grande

# Artículos de interés

**Guía rápida sobre GitHub:** https://docs.github.com/en/get-started/quickstart/set-up-git

**GIT y GitHub | Tutoriales:** Fundamentos de GIT GitHub Comandos explicados

**Videos del Profesor Alejandro Zapata (Coordinador y Docente de Codo a Codo):** link

**GIT y GitHub (tutorial en español). Inicio Rápido para Principiantes** link

**¿Cómo trabajar con Git desde Visual Studio Code?** video

**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. Por ese motivo antes, hay que aprender los fundamentos de GIT.

# GIT | Instalación

1. Descargar Git.
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 no tener éxito, ver instrucciones aquí

# Tarea para el Proyecto:

- Crear un repositorio en GitHub y utilizar los comandos básicos de Git para subir el proyecto, agregar colaboradores y comenzar a trabajar en conjunto.

# No te olvides de dar el presente

# Recordá:

- Revisar la Cartelera de Novedades.
- Hacer tus consultas en el Foro.
- Realizar los Ejercicios obligatorios.

**Todo en el Aula Virtual.**

# Muchas gracias por tu atención.

# Nos vemos pronto