Introducción a Git

> Manual redactado por Hugo Ruiz Sánchez. Documento organizado por secciones, vista recomendada en procesadores de documentos que admitan esquemas.

¿Qué es GIT?

Git es un sistema de control de versiones: nos permite disponer de un historial completo de todo el código desarrollado en nuestros programas.Git permite realizar el control sobre el código de manera descentralizada, permitiendo así la modificación grupal del mismo.

Puede usarse a modo de:

- Historial.
- Almacenar código.
- Trabajar en equipo.
- Indentificar errore

Debe ejecutarse en un entorno bash, disponible en windows mediante GIT BASH o nativamente en equipos GNU/LINUX.

```
git --version
```

Configuración global

Configuración de usuario.

Las configuraciones globales aplican a todos los documentos con los que GIT trabajará.

Configuración de nombre de usuario y correo electrónico:

```
git config --global user.name "Hugo Ruiz Sanchez"
git config --global user.email hugoruizschz@gmail.com
```

Configurar editor de código predeterminado.

El ajuste de VCS -Version Control System - está implementado para los editores de código más populares, que normalmente proporcionarán la instrucción recomendada.

Para Visual Studio Code es:

```
git config --global core.editor "(editor) --wait"
```

En el caso de Geany es:

```
git config --global core.editor "geany -imnst"
```

Para verificar el correcto funcionamiento, puede realizarse una consulta del fichero de configuración:

```
git config --global -e
```

Prevención de errores de compatibilidad

Al pulsar la tecla INTRO, Windows introduce el retorno de carro (CR -> ir a la primera columna de la línea) junto con el salto de línea (LF -> desciende una fila), empleándose dos caracteres en la codificación ASCII.

Los sistemas operativos UNIX (Linux / Mac) emplean solo el carácter LF .

El formato de Windows debe adaptarse en todos los casos al de un único caracter; y aunque en Linux no es necesario, es posible que el uso de ciertos editores añada el doble carácter, por lo que debe ajustarse solo al input.

Si es windows:

```
git config --global core.autocrlf true
```

Si es Linux / Mac:

```
git config --global core.autocrlf input
```

Ver más apartados de configuración

```
git config -h
```

Uso del GIT

El uso del GIT se resume en cinco etapas.

1. **Init (Inicializar)** Inicializa el GIT en un determinado directorio.
2. **Add (Añadir)**: Se indica a GIT qué archivos desea realizar para el próximo commit (envío).
3. **Stage (Perparar)** Los archivos pasan al staging (área de preparación). En este área intermedia, pueden revisarse los cambios antes de conformarlos.
4. **Commit (Comprometer)** Al confirmar los cambios, se adjunta un mensaje informándolos.
5. **Push (Enviar)** Efectuado el commit, por lo general se realiza envío del archivo al servidor (por ejemplo, github)

FASE INIT: Inicializar GIT en un directorio

Para inicializar git se usa:

```
git init
```

Este comando crea la carpeta oculta *.git* dentro del directorio en que se ejecute. Dicha carpeta se denomina "Detalle de implementación" y almacena todo lo necesario que usa git para respaldar nuestros proyectos.

En las empresas, se suele ignorar el directorio *.git* al momento de transmitir archivos.

FASE ADD: Selección de archivos.

La fase ADD consiste en la selección de los archivos a los que queremos hacer commit (enviar), esto se realiza mediante:

```
git add (archivo)
```

Admite expresiones regulares, como "*", por ejemplo:

```
git add *.txt
```

O simplemente seleccionar la totalidad de los archivos - una mala práctica, dado que es susceptible de introducir datos irrelevantes que ocupan espacio, o archivos ocultos -:

```
git add .
```

Borrar los archivos.

Al borrar un archivo en el directorio, el **git status** notificará que dicho archivo ha sido borrado. Estos archivos borrados deben añadirse igualmente, para que también se efectúe el borrado dentro del repositorio:

```
rm archivo
git add archivo
```

O simplemente, puede ejecutarse el eliminado y el commit al mismo tiempo de la manera siguiente:

```
git rm archivo
```

Renombrar los archivos

Si un archivo ha cambiado de nombre, reconocerá que el archivo con anterior nombre ha sido eliminado. Si se hace un commit de ambos, reconocerá el cambio de nombre:

```
mv antiguo_nombre nuevo_nombre
git add antiguo_nombre  nuevo_nombre
```

O simplemente, puede ejecutarse el cambio de nombre y el correspondiente commit al mismo tiempo de la manera siguiente:

```
git mv antiguo_nombre nuevo_nombre
```

Modificar el contenido de los archivos

Si durante la fase de stage un archivo es modificado, la versión previa a la modificación seguirá presente hasta que se ejecute un nuevo add sobre el archivo. Mientras tanto, el **git status** advertirá de los archivos *modificados* en rojo.

Ignorar archivos

La naturaleza de algunos archivos puede ser delicada, por ejemplo, variables de entorno, accesos a bases de datos, contraseñas... Es necesario proteger esos archivos del añadido, por

lo que debe crearse un fichero de texto en el directorio llamado *.gitignore* , introduciendo los nombres de los archivos y carpetas separados por un salto de línea:

```
telefonos.txt
documentos
fotodelicada.png
.env
```

Este fichero debe añadirse mediante un **git add** al repositorio

FASE STAGE: Preparación de los archivos

Para consultar el área de preparación, y saber qué archivos se encuentran en ella, debemos utilizar:

```
git status
```

Pero también puede usarse una forma simplificada, más legible:

```
git status -s
```

Consultar los cambios realizados sobre archivos fuera de staged.

Podemos comprobar qué modificaciones se ha realizado sobre un archivo hallando la diferencia (diff) entre ambos; Los números iniciales (-X, Y +Z,W) indican cada unod e los rangos de línea cambiados

```
git diff
```

Consultar los cambios realizados sobre archivos dentro de staged

```
git diff --staged
```

Retirar archivos de la fase stage

Si una operación está en el área de staging, todavía puede revertirse usando:

```
git restore --staged archivo
```

Esto anulará el add realizado anteriormente, pero únicamente del área del stage; si se quiere revertir un **git remove**, debe ejecutarse:

```
git restore archivo
```

FASE COMMIT: Confirmar los archivos

El commit confirmará que los archivos guardados en la fase de stage se quieren registrar. Siempre que se realiza una actualización así, debe informarse de lo que se ha hecho mediante un comentario orientativo:

```
git commit -m "Commit inicial - dos ficheros de texto plano"
```

Si se efectúa el comando sin argumentos, desplegará el editor de texto por defecto, facilitando la escritura, y solicitará que se describa el cambio realizado

FASE PUSH: Enviar repositorios a servidor (Github)

En la página principal de github, creamos un repositorio nuevo. Para añadir dicho repositorio, debemos ordenar:

```
git remote add origin https://github.com/hugoruizsanchez/prueba.git
```

Y para cada rama que queramos subir, debemos hacer:

```
git push -u origin nombre_rama
```

Para ejecutar este comando, nos solicitará una contraseña; dicha contraseña NO es la de nuestra cuenta de github, sino que debe corresponder a un token facilitado en:

Perfil (esquina superior derecha) > Settings > Developer settings > Personal access token > Tokens (Classic) > Generate new token (classic).

En **note** introducimos el nombre del equipo. Permitimos todos los permisos de **repo** y clickamos sobre **generate token**

A partir de ahora, para cada cambio realizado después de cada commit y subir al servidor, solo deberá realizarse un:

```
git push
```

Y para recibir las informaciones del servidor, y actualizar el repositorio:

```
git pull
```

Ver los cambios

Git guarda el seguimiento de las modificaciones sobre nuestro repositorio, hechas a través de los commits. Para verlo, debemos usar:

```
git log
```

Puesto que el comando no facilita la legibilidad, es más conveniente utilizar:

```
git log --oneline
```

Master, Branch y Merge

Definición

Master

Un proyecto puede ser **lineal**, es decir, los cambios realizados en el repositorio se suceden el uno al otro.

Puede ilustrarse así:

```
o-o-o-o-o-o
```

Esta línea se denomina *rama master* o *rama main*.

Branch

Pero para proteger la integridad de la rama master, git permite la posibilidad de realizar **bifurcaciones** o *branch* (rama), para poder trabajar en ellas sin afectar a la principal.

Puede ilustrarse así:

```
o-o-o-o-o-o
    |-o-o
```

Merge

Después de cada bifurcación, eventualmente el trabajo terminado puede volver a incorporarse a la *rama master*, esto se denomina *merge* (fusión).

Puede ilustrarse así

```
o-o-o-o-o-o-o-o
    |-o-o-o-|
```

Realizar branch

Es conveniente mantener un área de staging limpia antes de empezar con una nueva rama.

Para saber en qué rama nos encontramos, debemos ejecutar:

```
git branch
```

Para crear una rama debemos hacer:

```
git checkout -b nombre
```

Y para moverse entre ramas:

```
git checkout rama
```

Eliminar un branch

Para eliminar un branch creado por error, solo es necesario:

```
git branch -d nombre
```

Y en el caso de que git impida el borrado, debe aplicarse:

```
git branch -D nombre
```

Hacer un merge

Para fusionar una rama con la principal, solo es necesario introducir

git merge nombre_rama