

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Sistemas de control de versiones

Introducción

En la vida real, muchas veces hemos tenido la sensación de lo bueno que sería poder volver atrás en el tiempo para cambiar alguna cosa de nuestra vida... Bueno, pues básicamente eso es lo que permite un buen sistema de control de versiones: repensar lo que hemos hecho para hacerlo mejor... incluso quedándonos con ciertos cambios que hayamos hecho en una "vida paralela"

Cuando se trabaja en un proyecto está claro que conviene ir realizando backups de las distintas etapas del desarrollo de la aplicación. Pero esto no suele llegar. Puede suceder que me interese realizar alguna de las siguientes tareas:

- Cerrar versiones para distribución.
- Querer abrir distintas variantes de una misma versión para ver cual funciona mejor al final.
- Hacer pruebas en versiones paralelas y si funcionan bien juntar dicho código con el de la versión principal.
- Trabajar en varios equipos o querer que varias personas trabajen sobre el mismo proyecto o incluso dentro del proyecto sobre el mismo archivo y todo bien coordinado.
- Recuperar una versión válida si se localiza un problema durante la depuración. Es decir, puedo ir buscando hacia atrás en el histórico de versiones hasta que el problema desaparezca lo que me puede ayudar en la localización del mismo.

Para todas estas tareas sirve un sistema de control de versiones como Git que es el que usaremos en este curso por estar muy extendido. Se puede usar desde VSCode, VS2022 y desde otros IDEs sin ningún problema. Independientemente que se use Git u otro sistema (CVS, Subversion, Mercurial,...) todos usan conceptos similares del control de versiones.

Los SCV deben permitir:

- Mantener un histórico de los distintos recursos de un programa (código, imágenes, documentos, etc...).

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

- Actualizar distintas partes del código o archivos de recursos incluso por distintos autores.
- Cerrar una fase de desarrollo (sacar una versión) de forma que no pueda modificarse más (lo que se modifica sería una nueva versión).
- Realizar pruebas distintas dentro de una versión sin perjudicar a las demás (desarrollo de ramas).

Inicialmente se puede pensar que a golpe de backups se puede hacer todo esto y efectivamente es así, pero un SCV es un sistema de backup muy potente y versátil a la hora de recuperar ciertos escenarios, versiones o sincronizar distintos programadores sobre el mismo proyecto o incluso archivo.

También hay que tener en cuenta que un sistema de este tipo no es exclusivo de programadores. Por ejemplo se puede usar para realizar versiones de documentación interna de una empresa o incluso a la hora de escribir un libro o componer una canción o cualquier ámbito creativo donde implique posibilidad de querer volver a una versión anterior del documento en un momento dado.

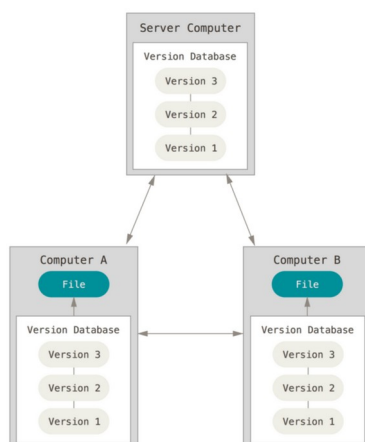
COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Tipos de control de versiones

Sistema local: Es el siguiente paso a simplemente renombrar los archivos con distintos nombres según la versión. Se mantiene una base de datos del archivo original y los cambios en el tiempo guardando exclusivamente las diferencias (esas diferencias se llaman "parches" o "patches" en inglés). De esta forma se puede llegar a cualquier versión del archivo. Se hace, por supuesto, de forma local cada uno en su ordenador (Sería similar a un history de cualquier programa).

Sistema Centralizado: Similar a lo anterior pero centralizado en un único equipo de forma que pueden acceder distintos usuarios. De esta forma cada uno de los parches está guardado en un ordenador central con múltiple acceso y se incluye en dichos archivos la autoría del parche.

Sistema distribuido: El sistema anterior por tanto gana al local en cuanto a capacidad colaborativa, sin embargo, tiene el gran problema de que si se corrompe por lo que sea dicho servidor, se pierde todo el trabajo. El sistema distribuido es un híbrido de ambos de forma que combina lo mejor de los dos anteriores.



Los cambios se guardan de forma local evitando conexiones continuas pero al mismo tiempo se mantiene en cada ordenador de cada usuario y en el servidor una copia completa de todo el historial de cambios. Cuando el usuario lo desea los cambios locales son sincronizados con los del servidor.

Si en el servidor hubiera una copia corrupta no pasaría nada pues disponemos del histórico en local.

<div>COLEXIO</div> <div>VIVAS S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO						CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR	Francisco Bellas Aláez (Curro)						

GIT

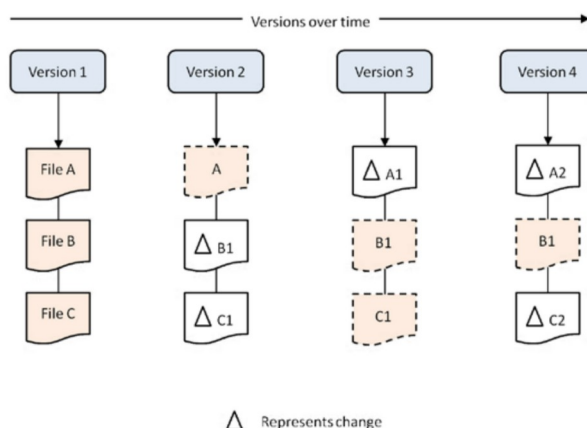
Descripción e instalación

Es un sistema de control de versiones distribuido diseñado por Linus Torvalds con el objetivo entre otros de coordinar el desarrollo del kernel de Linux en el que participan varios centenares de programadores distribuidos por todo el mundo. Dicho kernel está compuesto actualmente por más de 9 millones de líneas de código y más de 25000 archivos. Todo esto es una prueba clara de que Git es un buen sistema para controlar versiones de software de forma distribuida.

Si quieres profundizar existe un libro gratuito a través de web:

<https://git-scm.com/book/es/v2>

Una característica de git que lo hace rápido es que por cada cambio guarda una "fotografía" de todos los archivos de un proyecto y esto hace que sea veloz a la hora de recuperar archivos de versiones previas:



Git es un scm multiplataforma y lo aprenderemos a usar desde el terminal, pues es la base común que encontramos independientemente que trabajemos en un sistema operativo o en otro. Por supuesto existen plugins o extensiones para los diferentes IDEs o editores como VSCode o Visual Studio que comentaremos más adelante.

La versión más reciente en el momento de actualizar este documento es la 2.42. Hay más información sobre la misma en este enlace:

<https://git-scm.com>

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM					
	MÓDULO							CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:				
	AUTOR	Francisco Bellas Aláez (Curro)							

Windows

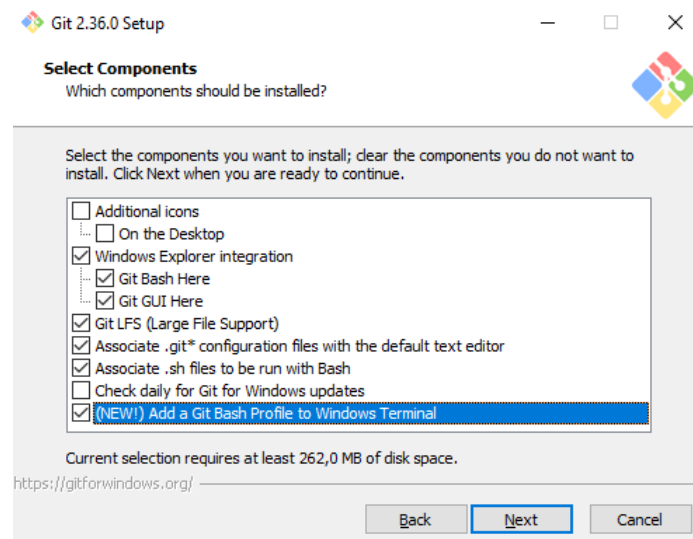
En la web de descargas del proyecto git:

<https://git-scm.com/download/win>

se descarga la versión adecuada para el Windows donde se desea instalar (32 o 64 bits).

Se ejecuta el instalador y se recomienda no pulsar "siguiente" sin pararse a leer los distintos puntos pues existen elementos de configuración importantes que conviene entender.

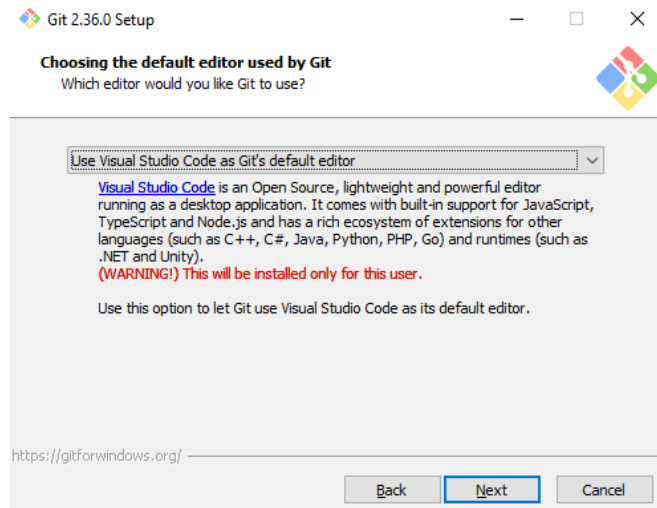
Sí vamos a dejar por el momento los componentes por defecto pero añadiéndole el Git Bash como se ve en la imagen:



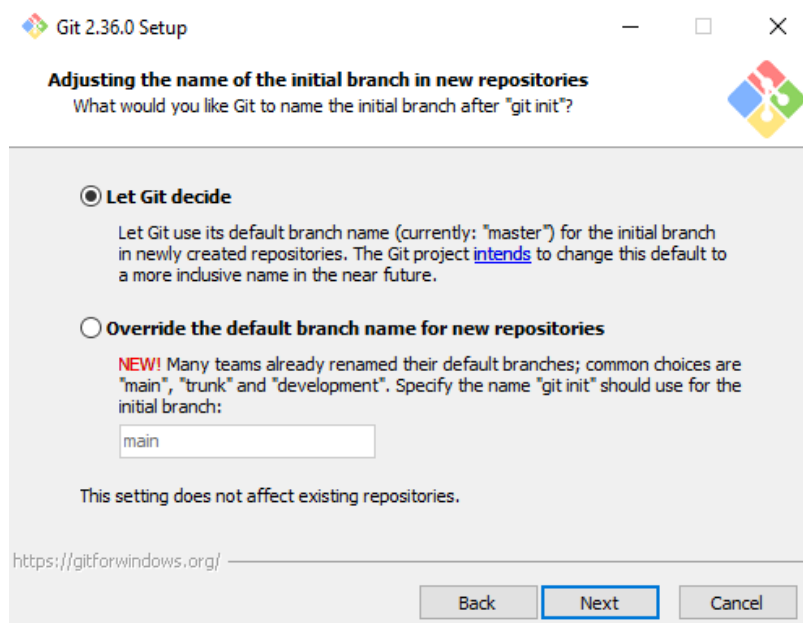
Es simplemente una consola tipo unix que podemos usar.

En la selección del editor por defecto vamos a poner el vscode (si alguien prefiere puede poner otro).

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				



A continuación se indica el nombre que se le da a la "línea" principal de trabajo. Históricamente en git se llamó "master" pero muchos lo cambian a opciones como main o development. Lo dejamos por defecto y ya veremos como cambiarlo.



COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

El ajuste del path y el conexionado remoto (SSH) dejamos el recomendado.

En proyectos donde está metido Windows, es recomendable que git se encargue de cambiar los retornos de carro según vengan de un unix o no. Dejamos la primera opción que ya viene marcada.

Las siguientes opciones (MinTTY, git pull, Credential helper, Extra Options, Experimental options) también las dejamos como vienen por defecto. Se irán entendiendo a medida que trabajemos con esta herramienta.

Linux Mint

La instalación en Linux es simple mediante repositorios. Accede a la consola y ejecuta:

```
$ sudo apt update
$ sudo apt install git-all
```

Esto instala tanto git como otras herramientas que nos harán falta. Debes tener en cuenta que para usar todo el potencial de git, este debe ser usado desde línea de comando.

macOS

En una consola en línea de comando escribe:

```
$ git
```

si no están instalado previamente el sistema te guiará para instalar las herramientas de desarrollo necesarias.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Configuración inicial

Tras la instalación podemos comprobar que git está instalado ejecutando desde una consola windows o desde git bash el comando:

```
$ git --version
```

o

```
$ git -v
```

El símbolo \$ indica el prompt de la consola bash, en otra consola puede cambiar dicho símbolo.

Identidad

Lo primero que haremos es identificarnos de forma local como usuarios del sistema git. Esto es necesario porque por cada actualización que se envía de un archivo (commit) se guarda la identidad. Lo más sencillo es establecer una identidad global para el equipo en el que te encuentres y que la cojan todos los proyectos.

Esto se hace cambiando las "variables de entorno" de git de la siguiente forma:

```
$ git config --global user.name "Aloy"
$ git config --global user.email "aloy@zerodawn.com"
```

En este caso se puede ver que dichas variables son user.name y user.email. Por supuesto debes inicializarlo con tus datos. Salvo que seas Aloy, entonces usa estos.

Puedes ver las variables inicializadas mediante:

```
$ git config --list
```

De esta forma se pueden configurar elementos como el editor que usará git que sería así (para poner el nano en vez de vscode):

```
$ git config --global core.editor nano
```

Si quisieras quitar alguna variable (por ejemplo que añadiste por error) escribe:

```
$ git config --global --unset
```


COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Podéis ver más variables de configuración en:

https://git-scm.com/docs/git-config#_variables

Si en un momento dado para un proyecto particular en lugar de para todo el usuario actual se quieren cambiar las credenciales simplemente se ejecuta el comando anterior sin el modificador `--global`.

El total de variables junto con el archivo donde están inicializadas (archivo global, de usuario o de proyecto) lo puedes ver mediante:

```
$ git config --list --show-origin
```

Repositorio de proyecto e inicio.

A continuación se le indica a git que proyecto me puede interesar tener gestionado. Para ello simplemente me voy al directorio del proyecto y ejecuto el comando:

```
$ git init
```

Esto crea un **directorio .git** en la propia carpeta donde se va a almacenar toda la información necesaria para el control de versiones de dicho proyecto. Por cada proyecto que quiera gestionar con git tengo que hacer un **init**. Esto englobaría todas las subcarpetas del mismo.

Cuando trabajamos en git se permite tener varias ramas de desarrollo. Siempre hay una que es la principal donde desembocarán las pruebas de las secundarias. El nombre de esta rama está indicado por el archivo `.git/HEAD`.

Es probable que el nombre actual sea el especificado por git de forma estándar: `master`. Puedes cambiar el nombre mediante el comando:

```
$ git branch -m main
```

Podrás ver que el contenido del archivo `HEAD` pasa a ser:

```
ref: refs/heads/main
```

pues en ese directorio git guarda el nombre de las ramas y `HEAD` es el puntero a la rama en uso actual. Lo iremos viendo en la práctica posterior.

Se puede usar también git desde modo gráfico con alguna herramienta como GitGUI que ya viene con Git, otra herramienta como GitKraken o una extensión de VSCode.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Práctica Guiada

Una vez que tenemos creado el repositorio, para ver como funciona git, vamos a ir haciendo pruebas con elementos muy sencillos y explicándolos al mismo tiempo.

Veremos sobre todo que git nos permite gestionar versiones de cualquier documento, no se limita exclusivamente a archivos de programación.

Los comandos que no son propios de git están pensados para usar el git bash, pero se podría hacer desde un cmd o Powershell usando los comandos típicos de consola de Windows. Es decir, si hay que hacer un mkdir o un ls, en el cmd de windows se realizaría un MD o un DIR respectivamente.

El resto de comandos de git funcionan de forma indistinta en cualquier consola.

Sigue los siguientes pasos tratando de entender todo lo que haces y preguntando las dudas que te surjan al profesor:

Crea un directorio denominado **prueba_git** y accede a él.

```
$ mkdir prueba_git
$ cd prueba_git
```

Crea mediante un editor de texto dos archivos dentro de dicho directorio. Uno denominado **texto.txt** y otro **script.sh**.

En el primero (*texto.txt*) introduce simplemente las palabras (Puedes usar nano en git bash o un editor cualquiera)

```
uno
dos
tres
```

Como se ve, cada una en una línea.

El segundo (*script.sh*) será un script con estas dos líneas:

```
echo Listado completo
ls -l
```

Si prefieres que funcione en consola de Windows el archivo denomínalo script.bat y

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

usa dir en lugar de ls.

Si quieres pásalo a ejecutable y Pruébalo mediante los comandos:

```
$ chmod a+x script.sh
$ ./script.sh
```

1. Inicialización (init y status)

Añádelo como **nuevo repositorio local (debes estar dentro del directorio prueba_git)**

```
$ git init
```

Te debe salir un mensaje similar a este:

```
Initialized empty Git repository in /Users/curro/temp/prueba_git/.git/
```

Por supuesto con el directorio que has usado tú.

Como se comentó, esto **crea** un directorio denominado **.git** donde se guarda toda la información de histórico del repositorio. Por cada versión de cada elemento que haya en el repositorio, aquí se hará una copia con una estructura determinada.

Por tanto, en este momento tenemos un repositorio git con dos archivos sin rastreo (**untracked**). Es decir, que git no los tiene en cuenta para ser parte del repositorio.

Puedes verlo con el comando:

```
$ git status
```

Debe salir un mensaje similar a este:

```
On branch master
```

```
No commits yet
```


```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
script.sh
```

```
texto.txt
```

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

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Esto significa que aunque estos archivos estén en el directorio, no están gestionados por git. Esto es importante pues puedo tener un directorio con elementos gestionados y no gestionados por git. Los no gestionados no se replicarán en otras máquinas con las que estemos compartiendo.

Si te fijas en la primera línea habla de la rama master. Este es el nombre que le damos y que se podría cambiar, por ejemplo a main, que es muy común.

Cambialo a main con el comando:

```
$ git branch -m main
```

Junto al prompt (si estas en git bash) ahora ya indica el nuevo nombre. Deberías ver algo así (salvo lo correspondiente al usuario curro, por supuesto)

```
curro@CURROBELLAS0741 CLANGARM64 ~/prueba_git (master)
$ git branch -m main
curro@CURROBELLAS0741 CLANGARM64 ~/prueba_git (main)
```

También si haces un status o miras el archivo .git/HEAD verás el nombre de la rama en la que estamos con el nuevo nombre.

Restáuralo a master:

```
$ git branch -m master
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

2. Añadiendo archivos (add y commit)

Prueba a ejecutar el siguiente comando para **añadirlo** a archivos rastreado:

```
$ git add script.sh
$ git status
```

Ahora verás que `texto.txt` sigue sin estar rastreado pero `script.sh` pasa a ser un archivo rastreado por git (**tracked**).

On branch master

No commits yet

Changes to be committed:
 (use "git rm --cached <file>..." to unstage)
 new file: script.sh

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

Esto significa que git comprobará si hay cambios desde la última versión. Si embargo aún no está confirmado. Esto es, no se sincronizará con los equipos remotos.

Por tanto **confirmamos o hacemos un commit** para establecer una versión sincronizada del archivo. Además se guarda con la identidad que se configuró al principio (user.name y user.mail).

```
$ git commit -m "Confirmación inicial"
[master (root-commit) 972303a] Confirmación inicial
 1 file changed, 2 insertions(+)
 create mode 100644 script.sh
$ git status
```

Aparece solo el aviso de `texto.txt` que no está en seguimiento. El resto esta guardado como versión.

Al confirmar se añade un mensaje que queda guardado con cada commit. Ahí se suelen indicar los cambios desde la última versión. Si hubiera muchos cambios o el mensaje fuera muy largo se puede hacer (no hace falta que lo hagas ahora):

```
$ git commit -F mensaje.txt
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

En el archivo mensaje.txt, lógicamente, iría todo el texto asociado al commit.

Por tanto con **add** indicamos que un archivo va a ser gestionado por git y con **commit** hacemos en git una foto (una versión) de dicho archivo para que sea sincronizado entre distintos equipos.

Cuando hablamos de versiones no os planteéis versiones comerciales de un producto. Es decir, de una aplicación puede salir al mercado la versión 1.0 y cinco meses más tarde la 1.1. Pero no penséis que en git hay solo esas dos. Entre ellas a lo mejor hay mil o diez mil versiones más, pues por cada pocos cambio o pruebas que se hacen en un programa se hace un commit lo que lleva a una nueva versión. Las que vemos en el mercado son solo las versiones comerciales.

Concretamente puedes ver en el directorio .git/objects que aparece un directorio con un número y dentro un número más grande. Eso es un código único donde guarda git de forma comprimida cada uno de los objetos que está manejando. Como curiosidad puedes ver lo que hay guardado (está codificado) en alguno de los hash mediante el comando:

```
$ git cat-file has_code -t
```

Sustituye hash_code por las 4 o 6 primeras cifras del número y ya lo detecta git. Hay tres tipos de objetos: blob (archivos), tree (directorios), commit. También hay tags que veremos más adelante.


Añadimos también texto.txt:

```
$ git add texto.txt
$ git commit -m "Añadido archivo texto.txt"
$ git status
```

Tras esto nos dice que está todo en el directorio confirmado. Algo así:

```
On branch master
nothing to commit, working tree clean
```

Puedes ver que el árbol de objetos ha crecido notablemente.

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

3. Modificando archivos

Ahora abre el archivo **texto.txt** y añádele otra línea con la palabra cuatro. Tras ello ejecuta:

```
$ git status
```

Verás que git lo toma como "modificado", pero indica que no hay cambios agregados al commit.

```
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:   texto.txt
```

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

Para prepararlo **para el siguiente commit hay que ejecutar de nuevo add**. Add es un comando que además de añadir a rastreo nuevos archivos, los prepara para el siguiente commit (puede no interesar actualizar en versión algún archivo que está en prueba).

```
$ git add texto.txt
$ git status
```

Resulta:

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

No hagas todavía commit. Si ahora modificas de nuevo texto.txt (añádele cinco) y miras el estado de git verás que aparece en stage (preparado para commit) y en modificado sin preparar:

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

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:   texto.txt
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Si se hace ahora el commit, confirmaría la versión anterior (la que aparece en verde), no la última modificación. Lo que implicaría que al sincronizarlo en otro equipo (aun no hemos visto como se hace) sincronizaría dicha versión (la verde).

Debes de hacer add si quieres incluirlo todo:

```
$ git add texto.txt
$ git commit -m "Ampliada la explicación del texto"
$ git status
On branch master
nothing to commit, working tree clean
```

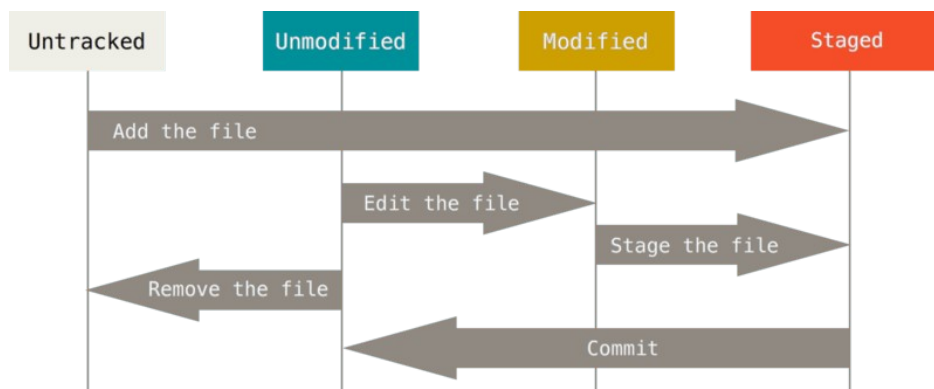
Vuelve a estar todo limpio.

Es posible deshacer cambios pero es la parte más "peligrosa" de git ya que se podrían llegar a perder datos, es por ello que hasta que se coja un poco de práctica es recomendable hacer siempre experimentos con directorios de prueba.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

4. Ciclo de vida de Git e información (git show y git log)

Podemos resumir el ciclo de vida de archivos en el repositorio local con el siguiente esquema:



Un archivo puede estar:

- Untracked: no se gestiona.
- Staged: gestionado y preparado para confirmar en el siguiente commit.
- Unmodified: Estado del archivo tras el último commit, podemos considerarlo una "versión" cerrada del mismo.
- Modified: Un archivo modificado desde el último commit. Es necesario mandarlo mediante add a Stage para indicar que puede confirmarse como válido en el siguiente commit.

Por tanto se mantienen tres árboles de trabajo. Como si fueran tres directorios:

Directorio de trabajo local: tu propio directorio donde tienes tus archivos.

Index o Stage: Es el almacén donde se colocan los archivos preparados.

HEAD: Que apunta a donde se ha hecho el último commit (ultima versión).

Si el flujo de trabajo habitual me está obligando continuamente a pasar de modificados a la de preparación mediante add y luego confirmar mediante commit, puedo saltarme un paso (o mejor dicho, resumirlo) ejecutando el comando

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```
$ git commit -a -m "Nueva versión"
```

Ojo, porque **si hay un nuevo archivo estoy obligado a hacer add**. Esto es válido solo para modificaciones. Así en un único comando realizas add y commit.

También es cómodo hacerse un script con las acciones repetitivas que hacemos.

Si quieres ver **información sobre la última versión confirmada**:

```
$ git show
```

Si hay mucho que mostrar puedes verlo con los cursores y usar *q* para finalizar. Sale un montón de información que iremos entendiendo poco a poco. Dicha información es sobre los últimos cambios en el/los archivos del último commit. Por ahora no te preocupes.

Por otro lado, si queremos ver los **cambios que se han ido realizando en forma de histórico** puedes ejecutar el comando:

```
$ git log
```

En este caso lo mismo, sale bastante información pero en este caso es bastante más comprensible. Veamos, debe salir algo similar a esto:

```
commit b34f694ad7dd653ea8c05c2645a045e4a444c06e (HEAD -> master)
Author: Aloy <aloy@zerodawn.com>
Date:   Wed Apr 15 12:31:34 2020 +0200
```

Ampliada la explicación del texto

```
commit 7a50da6b7213540b50ad492c62ded95d30c34b7d
Author: Aloy <aloy@zerodawn.com>
Date:   Wed Apr 15 12:01:52 2020 +0200
```

Añadido archivo de texto

```
commit b034e2966c9c1c1cc5547293591da777e189f543
Author: Aloy <aloy@zerodawn.com>
Date:   Wed Apr 15 11:34:01 2020 +0200
```

Confirmación inicial

Estamos viendo un log de los 3 commits que hemos hecho (si has hecho alguno más también aparecerá).

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Las líneas que empiezan por commit, indican al lado una clave única de versión (hash), más adelante veremos como personalizarla. Además indica que se está trabajando sobre la rama maestra (HEAD→master). En git se pueden tener ramas paralelas de trabajo sin conflicto entre ellas para hacer pruebas. Veremos algo más adelante.

Luego indica por cada commit autor, fecha y hora. Lógicamente aquí os pueden aparecer distintos datos dependiendo de cómo hayáis configurado git al principio, no pasa nada.

Por último aparece el mensaje que escribimos en cada commit.

Este comando muestra solo los objetos commit indicando además el hash asociado. Podría verse por separado con más información:

```
$ git cat-file b34f -p
```

Con información sobre el directorio (tree) y el commit previo (parent) además del resto de datos del commit.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

5. Diferencias (git diff)

Coge ahora el archivo texto.txt, añádele una línea (seis) y elimina otra (la tres, por ejemplo).

Para conocer las diferencias entre un archivo modificado y el que se ha confirmado en la última versión se puede usar el comando

```
$ git diff
```

Esto devuelve una respuesta en "Formato Unificado" (Unified format) el cual es un formato específico para ver diferencias entre archivos. Sin pasar a explicarlo todo, si que podemos entender la parte final donde se relatan las diferencias:

```
diff --git a/texto.txt b/texto.txt
index 23c83ff..bf98941 100644
--- a/texto.txt
+++ b/texto.txt
@@ -1,5 +1,5 @@
 uno
 dos
-tres
 cuatro
 cinco
+seis
```

En la línea que empieza por doble arroba (@@) el - se lee como "archivo previo" y el + como "archivo final". Es decir, aquí nos indica que se están mostrando las filas de la 1 a la 5 en el archivo origen y en el final, pues el primer número indica fila inicial y el segundo cantidad de líneas.

En las otras líneas el - es línea en el archivo origen que no existe en el archivo final y el + indica línea insertada en el archivo final y que no estaba en el origen.

Ejecuta:

```
$ git commit -a -m "Probando diff"
```

da información sobre lo añadido y eliminado:

```
[master 6292380] Probando diff
1 file changed, 1 insertion(+), 1 deletion(-)
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Ahora en texto.txt añade línea (siete); luego ejecuta de nuevo

```
$ git diff
```

y obtendrás:

```
diff --git a/texto.txt b/texto.txt
index bf98941..430cdae 100644
--- a/texto.txt
+++ b/texto.txt
@@ -3,3 +3,4 @@ dos
    cuatro
    cinco
    seis
+   siete
```

Ahora la @@ indica que en el archivo previo (-) se presenta a partir de la línea 3 solo 3 líneas (3,3). Y en el modificado se presenta a partir de la línea 3, 4 líneas más (3,4).

Todo esto se complica con archivos grandes y muchos cambios. Con el tiempo lo irás entendiendo mejor. Haz pruebas eliminado añadiendo y ejecutando diff para entender esto.

Más información leyendo este enlace:

<https://stackoverflow.com/a/25931860>

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

6. Ignorar archivos (.gitignore)

Cuando empiezo a tener una cantidad de archivos importantes suele suceder que hay algunos que no me interesa gestionar a nivel de versiones. Por ejemplo, si estoy programando en Java, los archivos .class no tiene sentido gestionarlos, pues solo me interesa el código fuente no el compilado. O archivos de texto o directorios temporales (muchas veces se marcan con extensión tmp o acabados en ~).

Para evitar estos archivos se debe crear un archivo dentro del propio directorio de trabajo (en nuestro caso en prueba_git) con el nombre

`.gitignore`

Y para el caso que se habló en el primer párrafo podría contener las siguientes líneas:

```
# ignora los archivos terminados en .class
*.class

# ignora los archivos terminados en ~
*~

# pero no importante~, aun cuando había ignorado los archivos
terminados en ~ en la linea anterior
!importante~
```

Observa que el # indica línea de comentario, es como el // en Java.

Veamos esto en funcionamiento: crea el archivo .gitignore y añádele lo anterior.

Añade a tu directorio (o en un subdirectorio) el archivo **Hola.java**. Por si no lo tienes cópialo de aquí y compílalo.

```
class Hola {
    public static void main(String[] args){
        System.out.println("Welcome to the Java World");
    }
}
```

También crea uno denominado **temporal~** aunque no tenga nada y también uno denominado **importante~** (puedes dejarlo también vacío en principio).

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Para confirmar que has seguido adecuadamente los pasos hasta aquí, si ejecutas

```
$ ls -a
```

debes obtener este listado:

```
.      .git      Hola.java  script.sh  texto.txt
..     .gitignore importante~  temporal~
```

Si tuvieras algunos archivos más porque estuviste haciendo pruebas no pasa nada, pero al menos los anteriores deben estar.

Ahora ejecuta:

```
$ git status
```

Obtendrás:

```
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:   texto.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    Hola.java
    importante~
```

Verás que hay archivos que se ignoran. Puedes como prueba quitar el archivo .gitignore y hacer de nuevo un status para ver como tiene en cuenta temporal~

Haz lo siguiente:

```
$ git add .
$ git commit -m "Añadidos fuentes en java y archivos importantes"
```

Con el . le indicamos a add que añada todo lo que está pendiente. También

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

subdirectorios si hubiera. Aparece:

```
[master 83eb263] Añadidos fuentes en java y archivos importantes
4 files changed, 15 insertions(+)
create mode 100644 .gitignore
create mode 100644 Hola.java
create mode 100644 importante~
```

Con un git status verás que está todo en regla.

Puedes probar a compilar Hola.java y verás que el class no lo incluye.

Dependiendo del lenguaje, framework o proyecto con el que estés pueden interesar distintos tipos de .gitignore, para no tener que hacerlos a mano dispones de algunas plantillas ya realizadas en :

<https://github.com/github/gitignore>

Para ver los archivos que hay en el directorio tras el último commit en la rama principal (apuntada por HEAD):

```
$ git ls-tree --name-only -r HEAD
```

deberías obtener:

```
.gitignore
Hola.java
importante~
script.sh
texto.txt
```


COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

7. Tags y gestión de versiones (git tag y git checkout)

Cuando ejecutas un **git log**, ves como ya se comentó, el historial de versiones. El número identificativo de la versión es un código único que aparece al lado de la palabra *commit* de cada versión. Por ejemplo:

```
commit b34f694ad7dd653ea8c05c2645a045e4a444c06e
```

Este es un número bastante complejo sobre todo si se quiere referenciar en alguna forma, por ejemplo con un show o para recuperar cierta versión en un futuro. Veamos el primer ejemplo, tendríamos que hacer:

```
$ git show b34f694ad7dd653ea8c05c2645a045e4a444c06e
```

pero evidentemente coge uno de los número que te haya salido a ti en el log.

El resultado es información sobre el commit y la comparación de dicha versión con la previa. Algo similar a esto (siempre dependiendo de lo que hayáis puesto en el show):

```
commit b34f694ad7dd653ea8c05c2645a045e4a444c06e
Author: Aloy <aloy@zerodawn.com>
Date: Wed Apr 15 12:31:34 2020 +0200
```

Ampliada la explicación del texto

```
diff --git a/texto.txt b/texto.txt
index 15bf608..23c83ff 100644
--- a/texto.txt
+++ b/texto.txt
@@ -1,3 +1,5 @@
 uno
 dos
 tres
+cuatro
+cinco
```

Como facilidad permite meter las primeras cifras de forma que no haya que meterlo todo, pero sería mejor si pudiéramos darle un nombre más claro y sencillo. Esto se hace con los tags y se usan normalmente para nombrar versiones según nos interese.

Para añadir un tag es muy simple. Si se quiere añadir a la versión actual simplemente se escribe:

```
$ git tag v0.7
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Si quieres añadir un tag a una versión anterior, por ejemplo la que se comentó al principio del apartado, simplemente se añade el código hash al final del comando:

```
$ git tag v0.3 b34f694ad7dd653ea8c05c2645a045e4a444c06e
```

Si ahora haces:

```
$ git log --oneline
```

Es una forma resumida (en una línea) de escribir el log. Verás salen las versiones abreviadas y los tags. (en las versiones antiguas de git no colorea nada por lo que había que añadirle el modificador --decorate).

```
83eb263 (HEAD -> master, tag: v0.7) Añadidos fuentes en java y archivos importantes
6292380 Probando diff
b34f694 (tag: v0.3) Ampliada la explicación del texto
7a50da6 Añadido archivo de texto
b034e29 Confirmación inicial
```

Si prefieres prueba el git log simple y verás la información más extendida (con fechas, autor, etc.)

También puedes hacer:

```
$ git show v0.3
```


Para ver una versión determinada sin tener que usar la clave compleja.

La numeración de versiones puede depender de la empresa, pero es muy habitual una numeración estándar como la que aparece en el siguiente enlace:

https://en.wikipedia.org/wiki/Software_versioning

4.2.1

MAJOR Minor patch

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Si en un momento dado quieres ir a una versión antigua para revisar algo del proyecto puedes hacer:

```
$ git checkout v0.3
```

Por supuesto en vez de v0.3 puedes ir al tag o al código hash que quieras.

Luego para volver a la versión más reciente harías:

```
$ git checkout master
```

Si en vez de master has usado otro nombre (como main) pues deberías usarlo. También valdría para cambiar de rama (las ramas las vemos más abajo).

Ojo porque al ir a una versión más antigua, si deseas modificarla deberías hacer primero una nueva rama como se explica más abajo, ya que si no corres el riesgo de perder dichas modificaciones al volver a la versión más reciente.

Los tags se almacenan en el directorio **.git/refs/tags**

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

8. Ramas

Una gran utilidad de los sistemas de control de versiones es la realización de ramas, es decir, archivos paralelos a los principales con el objetivo de hacer pruebas o poner en práctica ideas. Luego la rama se puede abandonar, eliminar o mezclar con la rama principal.

Este es un punto complejo, sobre todo por la solución de conflictos que pueden aparecer, y no vamos a profundizar en él hasta haber trabajado de forma simple con git.

Otra ventaja de las ramas en git es que pueden mantenerse solo en el sistema local sin necesidad de llevarlas al remoto. Esto es muy práctico para que programadores en equipo hagan sus pruebas en sus equipos locales sin tener que ramificar el proyecto general de todo el equipo.

En git bash junto al prompt aparece en todo momento en la rama en la que estamos trabajando, Por ejemplo:

```
curro@win10vm MINGW64 ~/pruebas_git (master)
```

Nos dice que estoy en la rama master.

Ejecuta el siguiente comando para **crear una rama** con el nombre Prueba:

```
$ git branch Prueba
$ git log --oneline
```

Verás que ahora en la versión más reciente aparecen dos ramas, la principal **master**, y la que has creado (Prueba).

```
83eb263 (HEAD -> master, tag: v0.7, Prueba) Añadidos fuentes en java y archivos importantes
6292380 Probando diff
b34f694 (tag: v0.3) Ampliada la explicación del texto
7a50da6 Añadido archivo de texto
b034e29 Confirmación inicial
```

Ahora si sigues cambiando cosas y haces commits lo haces en master, porque es donde apunta HEAD que es el puntero de trabajo.

Si ejecutas:

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```
$ git branch
```

```
Prueba
```

```
* master
```

Sale el listado de ramas y marcada con asterisco la rama en la que te encuentras trabajando (coincide con la del prompt).

Las ramas se almacenan en el directorio .git/refs/heads

Para **cambiar a la rama de prueba**:

```
$ git switch Prueba
```

También se puede ejecutar **git checkout Prueba**, haría lo mismo, pues checkout es un comando que vale para varias cosas.

Sale el siguiente mensaje:

```
Switched to branch 'Prueba'
```

Y en el prompt te indica la nueva rama.

```
curro@win10vm MINGW64 ~/pruebas_git (Prueba)
```

Haz de nuevo:

```
$ git branch
```

```
* Prueba
```


```
master
```

Como ves el asterisco ha cambiado de rama.

Prueba también:

```
$ git log --oneline
```

```
83eb263 (HEAD -> Prueba, tag: v0.7, master) Añadidos fuentes en java y archivos importantes
6292380 Probando diff
b34f694 (tag: v0.3) Ampliada la explicación del texto
7a50da6 Añadido archivo de texto
b034e29 Confirmación inicial
```

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Ahora HEAD, que recuerda que es el puntero de trabajo, apunta a Prueba, por tanto todo commit que hagas se hará en dicha rama, dejando intacta la principal.

Puedes ver el archivo .git/HEAD y verás que efectivamente ahora apunta a la nueva rama.

```
$ git commit -a -m "Rama para pruebas de código"
$ git log --oneline
```

Puedes ver en el log que ahora HEAD apunta a Prueba por lo que todos los cambios será en paralelo con la master no afectando a esta. Es decir los cambios que hagas en cualquier archivo no afectan a la rama principal.

Prueba a cambiar el Hola.java de añadiéndole la línea:

```
System.out.println("I have no more branches to commit, said the Ent");
```

A continuación haz el commit que le corresponda:

```
$ git commit -a -m "Llegan los Ents a Java"
$ git tag v0.7-Ent-Release
```

Haz un log y verás:

```
c862211 (HEAD -> Prueba, tag: v0.7-Ent-Release) Llegan los Ents a Java
83eb263 (tag: v0.7, master) Añadidos fuentes en java y archivos importantes
6292380 Probando diff
b34f694 (tag: v0.3) Ampliada la explicación del texto
7a50da6 Añadido archivo de texto
b034e29 Confirmación inicial
```

Vuelve a la rama principal para ver que ahí sigue la versión previa:

```
$ git switch master
$ cat Hola.java
```

Y sale sin la línea nueva añadida porque lo hicimos en la rama.

Si el experimento convence, se puede **juntar nuevamente con la rama principal** o con otra cualquiera con

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

\$ git merge Prueba

Y nos informa de lo sucedido:

```
Updating 83eb263..c862211
Fast-forward
 Hola.java | 1 +
 1 file changed, 1 insertion(+)
```

Esto no siempre es tan bonito y en ocasiones informa de conflictos o posibles problemas.

Evidentemente este ejemplo es muy simple pero da una idea de la potencia del uso de ramas. Si quieres profundizar en este apartado se recomienda leer este enlace:

<https://git-scm.com/book/es/v2/Ramificaciones-en-Git-¿Qué-es-una-rama%3F>

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

9. Eliminar y quitar de seguimiento

Si deseas eliminar archivos lo mejor es hacerlo a través del comando:

```
$ git rm importante~
```

Y luego un **commit**.

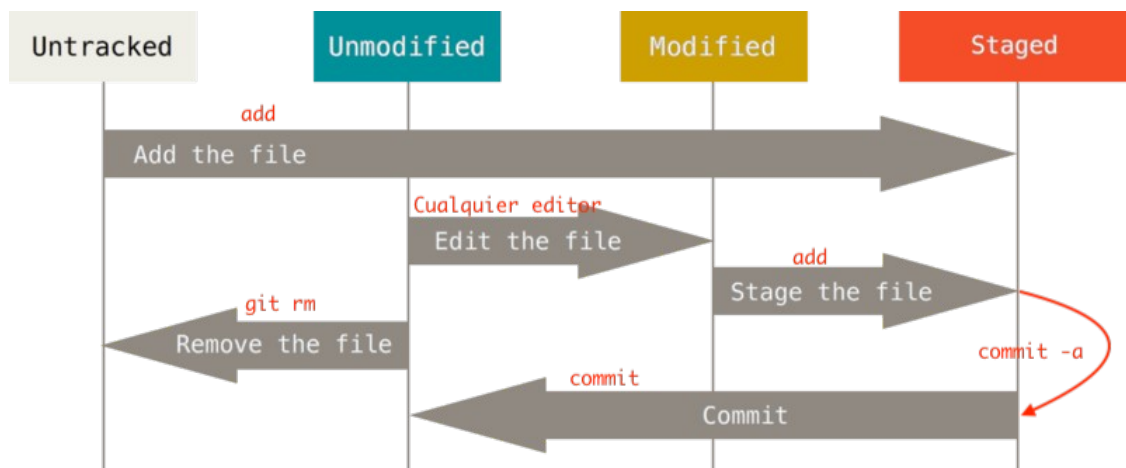
Si lo has eliminado previamente mediante un *rm* normal, tendrás que ejecutar también **git rm** o **git rm -f** si ya está confirmado el archivo.

Si había algún archivo ya en seguimiento que deseas quitar (por ejemplo los .class), para sacarlos de la zona de seguimiento (HEAD) ejecuta:

```
$ git reset HEAD *.class
```

Realmente HEAD es un puntero a la versión de trabajo del repositorio (la última versión confirmada).

Con esto podemos ver de nuevo el ciclo de vida asociado a los distintos comandos vistos:



COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

10. Repositorios remotos: GitHub

Uno de los puntos interesantes de Git y cualquier control de versiones, es la posibilidad de trabajo en equipo teniendo un repositorio al que pueden acceder varios programadores desde su equipo. De esta forma un programador tiene su repositorio local, tal cual lo hemos usado hasta ahora, y luego tiene un repositorio remoto al que vuelca sus modificaciones y desde el que vuelca al suyo modificaciones que hayan hecho terceros si están incluidos en el proyecto.

GitHub es precisamente un proveedor de repositorios Git para que distintos usuarios puedan compartir sus proyectos. El uso es gratuito y recientemente han dado la posibilidad de crear repositorios públicos o privados. La web es:

<https://github.com>

También existe para estudiantes un pack especial que permite repositorios privados y otras herramientas: <https://education.github.com/pack>

Conviene añadir además que aunque sea el más conocido, github no es el único servidor de repositorios git. Otros de uso habitual son:

<https://bitbucket.org>

<https://gitlab.com/>

Si lo único que necesitas es **clonar un proyecto** de GitHub (o de cualquier otro repositorio remoto), no es necesario tener cuenta. Simplemente ve a tu directorio de usuario y ejecuta:

```
$ git clone https://github.com/ColegioVivasCurro/HolaED
```

verás:

```
Cloning into 'HolaED'...
remote: Enumerating objects: 20, done.
remote: Total 20 (delta 0), reused 0 (delta 0), pack-reused 20
Unpacking objects: 100% (20/20), done.
```

Puedes comprobar que se te ha creado un directorio denominado **HolaED**.

Esto hace una réplica exacta del repositorio remoto en el directorio actual. Esta acción es muy habitual para coger proyectos libres que hay en github y hacer pruebas en tu ordenador local.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Ojo, con este comando no estás sincronizando nada, simplemente te descargas en tu equipo un proyecto. Sin permiso del propietario del repositorio no podrías subir nada al mismo. Por supuesto si el repositorio es tuyo sí vas a tener permisos para realizar la sincronización.

Cuando quieres gestionar tus propios proyecto o colaborar con otro (hacer un fork) es cuando sí debes crearte una cuenta. Conéctate y crea una.

Puedes probar el enlace: <https://try.github.io>

Que da varios recursos y cursos de manejo de git/github.


Para disponer de un nuevo repositorio para un proyecto **lo primero que debes hacer es crear el proyecto en tu cuenta de github** para luego sincronizar los archivos locales con el remoto. A continuación un ejemplo:

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner *

Repository name *

 ColegioVivasCurro


CalOfDutyPizzaWars


✓

Nombre del repositorio

Great repository names are short and memorable. Need inspiration? How about **cuddly-spoon**?

Description (optional)

☒  **Public**
 Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**
 You choose who can see and commit to this repository.

Autoexplicativo

Initialize this repository with:
 Skip this step if you're importing an existing repository.

☒ **Add a README file**
 This is where you can write a long description for your project. [Learn more.](#)

El README es para descripciones. Se escribe en lenguaje Markdown, de ahí su extensión md.

Add .gitignore
 Choose which files not to track from a list of templates. [Learn more.](#)


.gitignore template: Java ▼

Plantillas predefinidas para .gitignore. Por supuesto puedes hacer una a medida.

Choose a license
 A license tells others what they can and can't do with your code. [Learn more.](#)

License: None ▼

Si prefieres puedes poner master.

This will set  **main** as the default branch. Change the default name in your [settings](#).

① You are creating a public repository in your personal account.

Create repository

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Suponiendo que tienes un repositorio denominado MiPrueba donde quieres enlazar el directorio actual ya gestionado por git haz:

```
$ git clone https://github.com/ColegioVivasCurro/MiPrueba
```

Como se comentó esto simplemente copia el repositorio remoto al local. Ahora puedes modificar en local con nuevos archivos y ejecutar los add y commit que consideres.

Cuando estás preparado para sincronizar lo que hay que hacer es enlazar la rama que se desea sincronizar de forma local (en nuestro caso master o main) con la remota que normalmente tiene el nombre **origin** (puedes verlo en .git/config).

Para mandar los cambios al repositorio remoto se hará

```
git push -u origin master
```

Este comando lo que hace es volcar tu rama principal (denominada master) local en el repositorio remoto (origin).

El parámetro -u sirve para recordar estos nombres de forma que a partir del primer push simplemente haciendo

```
git push
```

Ya reconoce origin y master.

Lógicamente usa tu repositorio en lugar del que aparece en el ejemplo (puedes ver en la página de github lo que tienes que hacer, está todo explicado).

No es tan habitual pero si lo que quieres es sincronizar un repositorio local que ya estás usando con uno remoto, primero hay que indicarle al local cual es el remoto mediante

```
git remote add origin https://github.com/ColegioVivasCurro/MiPrueba.git
```

A continuación se sincroniza primero descargando lo que pueda haber en el remoto (normalmente solo el readme) mediante pull:

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```
git pull origin main --allow-unrelated-histories
```

Ojo, si el nombre de tu rama es master en vez de main eso debes cambiarlo.

Si deseas aprender más sobre la estructura interna de git, es **muy recomendable** el curso:

<https://app.pluralsight.com/library/courses/how-git-works/table-of-contents>

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO						CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR	Francisco Bellas Aláez (Curro)						

Front-ends gráficos

Por supuesto existen entornos gráficos que nos facilitan toda esta labor que se ha explicado en los puntos anteriores, pero como siempre, es conveniente conocer como funcionan los comandos que se usan para luego poder solventar ciertos problemas que no son triviales desde un modo gráfico.

Por ejemplo con git viene incluido Git GUI que entendiendo todo lo anterior es sencillo de usar.

También hay otros muy sencillos de usar como GitKraken o Github Desktop. Puedes ver una lista de clientes Git aquí:

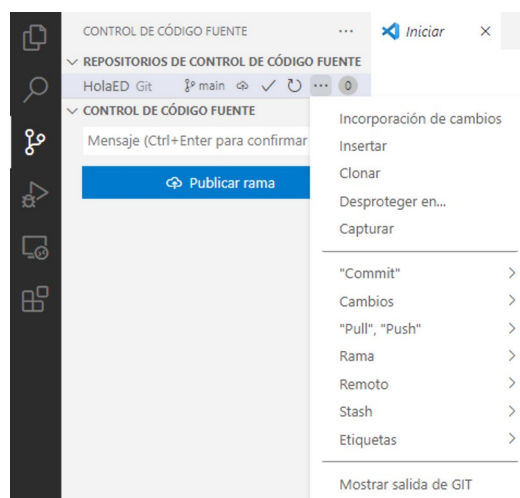
<https://git-scm.com/downloads/guis>

Otra posibilidad si lo único que quieres es experimentar es este proyecto que de forma visual vas viendo como avanza un proyecto a medida que se ejecutan distintas instrucciones git.

<https://github.com/git-school/visualizing-git>

<https://git-school.github.io/visualizing-git/>

También vscode (y otros IDEs) viene con la posibilidad de gestionar git/github. Si has entendido los pasos de la guía no te será difícil hacerte con cualquiera de estos interfaces gráficos.



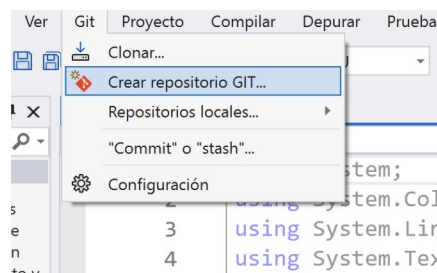
<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM				
	MÓDULO						CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:			
	AUTOR	Francisco Bellas Aláez (Curro)						

Visual Studio 2022 y Git

A continuación vemos como gestionar un proyecto en vs2022 usando git y github. En principio se supone que se entienden los conceptos vistos previamente por lo que simplemente se indica como realizar las tareas desde el entorno gráfico.

Crea un proyecto cualquiera en vs2022. Para esta práctica crearé uno de consola sencillo.

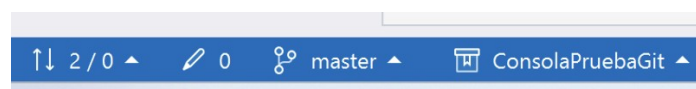
Una vez creado, como ya sabemos, aún no está gestionado por git. Para crear el repositorio nos vamos al menú Git→Crear repositorio Git.



Y ahí ya nos sale como configurar tanto el repositorio remoto como el local. Podrías ya hacer la configuración completa, pero para este ejemplo iremos poco a poco. Seleccionamos Solo local dejando las opciones por defecto, pero marcando crear README.md pues es un archivo muy típico descriptivo del proyecto.



En la barra de estado, a la derecha aparece diversa información sobre el repositorio y estado del proyecto. Los tooltips son autoexplicativos.

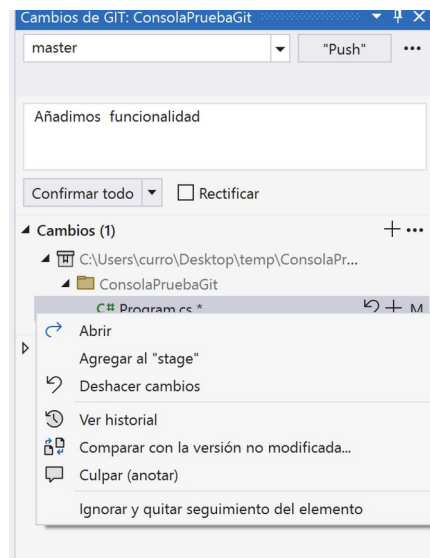


<div>COLEXIO</div> <div>VIVAS S.L.</div>	RAMA:	Informática	CICLO:	DAM					
	MÓDULO							CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:				
	AUTOR	Francisco Bellas Aláez (Curro)							

Con estos elementos del status bar y con el menu git se puede gestionar todo lo que hemos estado viendo desde la consola.

Por ejemplo si hacemos una modificación al archivo (añade por ejemplo un Write). Aparece en la estatus bar el icono lápiz (Cambio) con un 1 al lado indicando modificación. Haciendo doble click sobre dicho lápiz se accede a la ventana de commits.

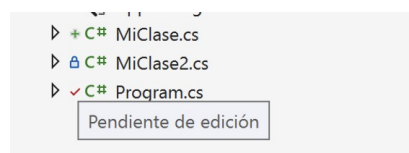
Pulsando sobre el botón derecho en el archivo modificado ofrece diversas opciones.



Aquí puedes ver el historial, ver la diferencia del nuevo archivo con el anterior, etc. Pulsando el + se hace el add del archivo. Luego se haría el commit.

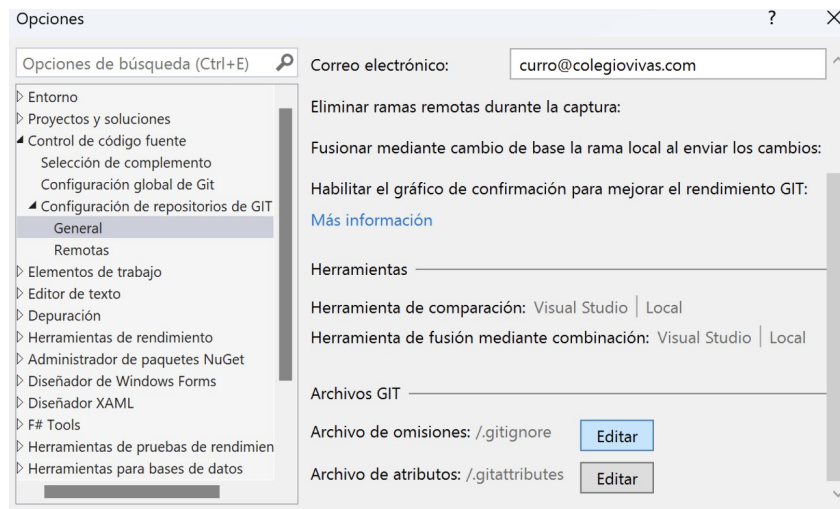
También se puede confirmar todo en el botón correspondiente. Simplemente pregunta, en el caso de que no se han hecho los add, si deseas que se hagan también automáticamente.

En el explorador de soluciones, veras mediante distinta iconografía el estado de cada archivo. Si tienes dudas pon el ratón sobre el icono y te indica es estado. Por ejemplo, el candado indica que está todo hecho y el archivo está, por tanto, protegido.



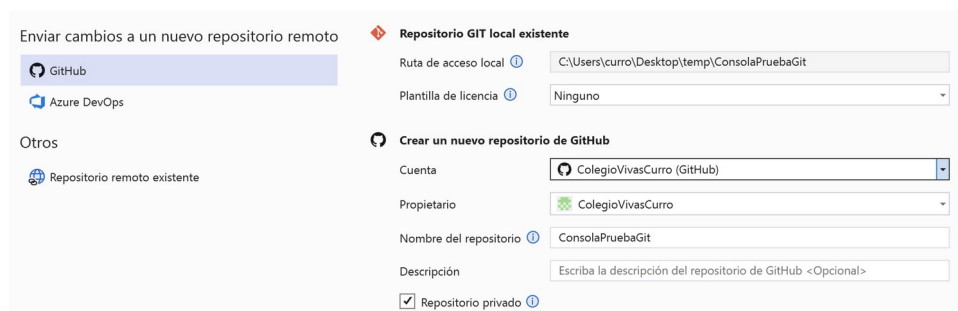
<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	DAM					
	MÓDULO							CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:				
	AUTOR	Francisco Bellas Aláez (Curro)							

En el menú de Git, en Opciones, dispones de varias posibilidades. Una importante es la configuración de gitignore, que puedes hacerla manual o a través de este menú como se ve en la siguiente captura:

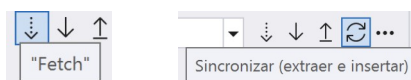


Haz pruebas y observa como se gestiona de forma local.

Cuando quiera enviarlo a un repositorio GitHub (u otro remoto). En el Menú Git pulsas en *Aplicar Push a un servicio Git*. Ahí puedes configurar el repositorio remoto.



Si se realizan cambios en el repositorio remoto, antes de seguir trabajando se recomienda pulsar sincronizar. O si se prefiere confirmar si hay algún cambio dale a Fetch.



COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

En principio con estos elementos ya puedes tener una gestión suficiente de git y github desde vs2022. Si quieres profundizar te recomiendo este enlace de Microsoft:

<https://learn.microsoft.com/es-es/visualstudio/version-control/git-with-visual-studio?view=vs-2022>

Y también estos vídeos (puedes encontrar un montón):

<https://www.youtube.com/watch?app=desktop&v=8zSVvTQXSic>

<https://www.youtube.com/watch?v=uYdfPa1VJ10>

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO					CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Referencias:

<https://git-scm.com/book/es/v2>

<http://rogerdudler.github.io/git-guide/index.es.html>

<https://app.pluralsight.com/library/courses/how-git-works/table-of-contents>

<https://stackoverflow.com/questions/2529441/how-to-read-the-output-from-git-diff/25931860#25931860>

<https://try.github.io/levels/1/challenges/1>

<http://stackoverflow.com/questions/2745076/what-are-the-differences-between-git-commit-and-git-push>

Libros:

- Git: Version control for everyone. Beginners guide. Ravishankar Somasundaram. Pack Publishing 2013.