Contenido

[1. GIT 2](#_Toc175687717)

[1.1. Las áreas de trabajo de Git 2](#_Toc175687718)

[1.1.1. El área de preparación (área de staging) 3](#_Toc175687719)

[1.2. Estados de los archivos en Git 3](#_Toc175687720)

[2. Gitflow o ciclo de vida de un sistema de control de versiones 4](#_Toc175687721)

[3. Versiones de Git: comandos o gráfica 6](#_Toc175687722)

[4. Instalación de Git 7](#_Toc175687723)

[5. Comandos principales de Git 13](#_Toc175687724)

[5.1. Instrucciones básicas de Linux 13](#_Toc175687725)

[5.2. Ayuda 14](#_Toc175687726)

[5.3. Crear un repositorio local 14](#_Toc175687727)

[5.4. Añadir ficheros al área de trabajo 15](#_Toc175687728)

[5.5. Añadir ficheros al área de preparación 15](#_Toc175687729)

[5.6. Aprobar los cambios 16](#_Toc175687730)

[5.7. Fichero .gitignore para poder excluir ficheros especificos 17](#_Toc175687731)

[5.8. Ver versiones creadas 18](#_Toc175687732)

[5.9. Añadiendo y deshaciendo cambios 18](#_Toc175687733)

[5.10. Deshaciendo cambios 21](#_Toc175687734)

[5.11. Gestión de ramas 25](#_Toc175687735)

[5.12. Guardar los cambios del área de trabajo (Git Stash) 27](#_Toc175687736)

[5.13. Fusionar ramas 28](#_Toc175687737)

[5.14. Comando cherry -pick 29](#_Toc175687738)

# GIT

Git es un sistema a base de comandos que nos permite controlar las versiones generadas en un proyecto. Concretamente, Git es una herramienta de control de versiones de software libre, diseñada por Linus Torvalds que no depende de un repositorio central y que con ella podemos mantener un historial completo de versiones. GIT es el SCV (sistema de control de versiones) de código abierto [más utilizado](https://www.g2.com/categories/version-control-systems?utf8=%E2%9C%93&order=g2_score) que te permite rastrear los cambios realizados en los archivos. Las empresas y los programadores suelen utilizar GIT para colaborar en el desarrollo de software y aplicaciones.

## Las áreas de trabajo de Git

El modo de trabajo estándar de GIT se basa en que los archivos pasarán por varios espacios antes de validarse en el repositorio o base de datos. Estas zonas son:

* **La carpeta de trabajo o área de working**, que es el directorio local de trabajo, donde creamos los archivos, y realizamos todas las modificaciones. Cuando se realiza una modificación de un archivo pasa al área de working.
* **El área de preparación o área de staging**, es un estado intermedio en el que se prepara a los archivos para luego validarlos en el repositorio local. Los cambios y modificaciones de los archivos del área de working, antes de validarse y pasar al área de commit, pasarán por el área de preparación.
* **El respositorio local o área de commit**, este repositorio lo creamos con GIT, y es donde se van a guardar nuestros archivos con las modificaciones y todas las versiones por las que se ha pasado. Normalmente pasaremos los archivos del área de preparación, al área de commit.
* **El repositorio remoto**, este repositorio estará fuera del equipo local, que, en el caso de colaborar con más usuarios, podremos sincronizar nuestros ficheros y cambios realizados con el repositorio remoto. Git no va a permitir acceder a los repositorios remotos desde cualquier sitio y lugar.

Escala de tiempo

Descripción generada automáticamente

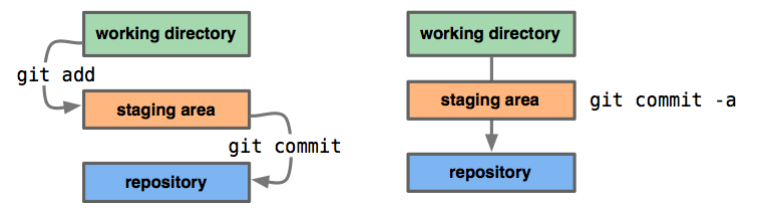
## El área de preparación (área de staging)

Es la zona donde se añaden los cambios que se van a hacer commit.

* NO es necesario añadir todos los archivos del área de trabajo al área de preparación
* Es incluso posible añadir al área de preparación modificaciones concretas dentro de 1

archivo (hunks) - Si hay 2 cambios en el archivo se puede hacer commit de 1 de los cambios en un primer paso y otro segundo commit en un segundo

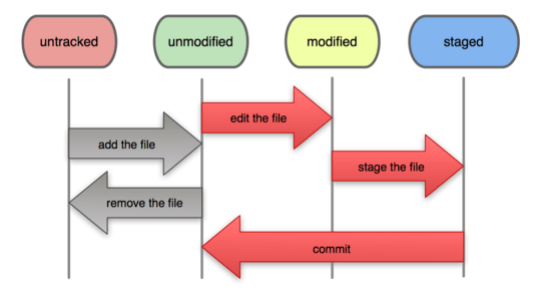
* Promociona una buena práctica de Git: haz commit frecuentemente
  + Que sean pequeños (si es posible)
  + Incluye sólo las modificaciones concretas que resuelvan el problema/tarea.



## Estados de los archivos en Git

Tres estados:

* **Modificado o untracked:** se han modificado los archivos en el área de trabajo, pero los cambios no se han confirmado en el repositorio local. Los archivos estarán en el área de trabajo.
* **Preparado o staged:** los archivos modificados se han marcado y preparado en su versión actual para la próxima confirmación. Estarían en el área de staging o preparación.
* **Confirmado o committed**: los datos del archivo están almacenados de manera segura en la base de datos local o repositorio local. Estaría en el área de commit.



# Gitflow o ciclo de vida de un sistema de control de versiones

El Gitflow es un flujo de trabajo basado en la estructura de Git que brinda un mayor control y organización en el proceso de integración continua dentro del desarrollo del software. Para ello, la herramienta utiliza la creación de un conjunto de ramas (branches) y se establece un convenio de nombres para nuestras ramas que permite a todos los miembros del equipo saber qué es cada cosa. Además, establece unas reglas comunes a la hora de mover la información de una rama a otra.

Podemos separar Gitflow en dos tipos de ramas:

* **Principales**: son las Branch *Master* y otra llamada *Develop*.
* **Soporte**: son las conocidas como *Feature*, *Release*, *Hotfix y Bugfix*.

**Caracteristicas**

Entre otras muchas, las características que lo hacen uno de los sistemas de control de versiones más utilizados a nivel mundial son:

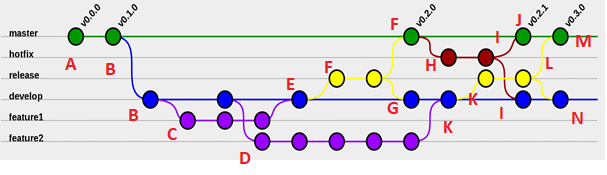
* Aumenta la velocidad de entrega de código terminado al equipo de pruebas.
* Disminuye los errores humanos en la mezcla de las ramas.
* Elimina la dependencia de funcionalidades al momento de entregar código para ser puesto en producción.

**¿Cuándo se recomienda implementarlo?**

* El equipo de trabajo está conformado por más de dos (2) personas.
* Se emplean metodologías ágiles.
* El proyecto tiene cambios frecuentes y se requiere actualizar el ambiente de producción garantizando continuidad en la operación.
* El proyecto tiene un nivel de complejidad considerable.
* Se desea tener un proceso de **soporte a errores efectivo** con actualizaciones rápidas.

**Politicas internacionales en nombre de ramas, Gitflow propone:**

* **Master** como rama principal del proyecto. Contiene el código de producción. NUNCA trabajaremos sobre ella. De esta rama no nace ninguna excepto los hotfix.
* **Develop** como rama de desarrollo. NUNCA trabajaremos sobre ella. De esta rama nacen todas las ramas de feature. La rama develop se fusionará con master cuando vayamos a desplegar el proyecto a través de las releases.
* **Feature** serán las ramas sobre las que trabajaremos normalmente. Llevan por defecto el prefijo feature/ seguido del nombre de la rama (ej: feature/add-event-cpt). Nacen SIEMPRE desde la rama develop y mueren cuando son fusionadas.
* **Hotfix** son las ramas que utilizaremos para corregir errores críticos encontrados en producción. Llevan el prefijo hotfix/ seguido del nombre de la rama (ej: hotfix/fix-form-submit). Nacen SIEMPRE desde la rama master y se fusionan contra master y develop, con el objetivo de poner el hotfix en producción y que también esté disponible para nuevos evolutivos en la rama de desarrollo.
* **Release** son las ramas que utilizaremos para crear nuevas versiones para desplegar a producción. Llevan el prefijo release/ seguido del número de versión. Es el mecanismo a través del cual fusionamos los nuevos desarrollos que tenemos en develop contra master.
* **Bugfix** son ramas que se utilizan para corregir errores que aún no han llegado a producción. Llevan el prefijo bugfix/ seguido del nombre de la rama (ej: bugfix/fix-wrong-logout-link). Nacen SIEMPRE desde la rama develop.



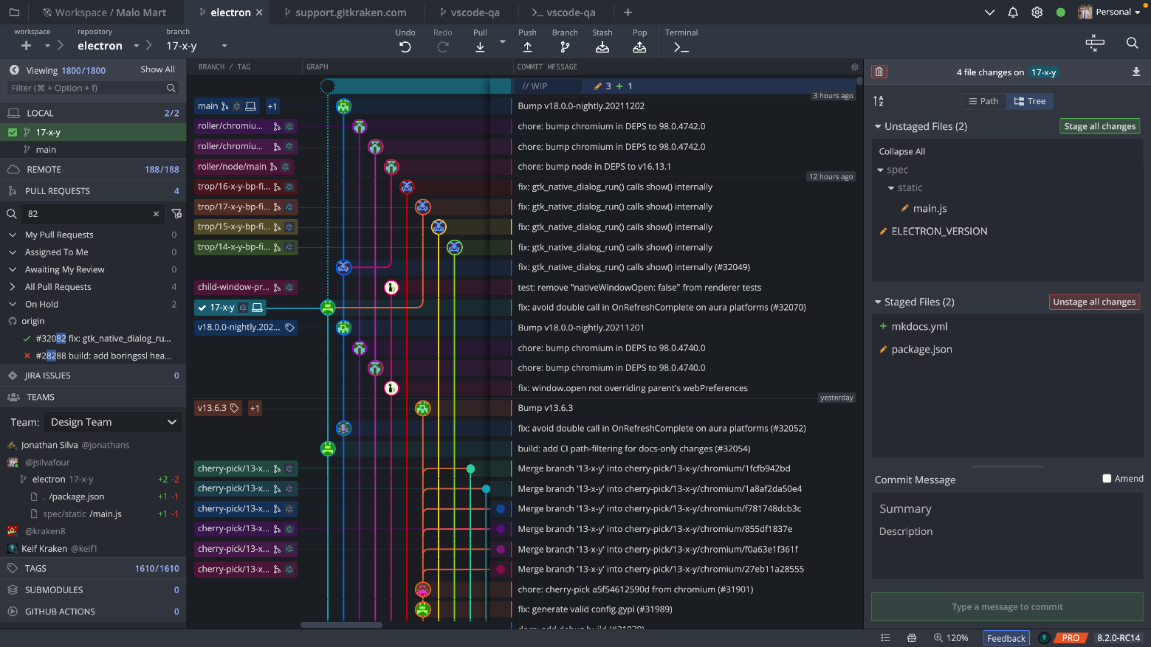
**Pasos de un control de versiones dentro de un proyecto:**

1. Se crea la rama *master* (NUNCA se desarrolla sobre ella). Se crea un TAG (etiqueta que nos ayuda a establecer puntos concretos en el código) que define el punto inicial *v0.0.0*
2. A partir del TAG v0.1.0 se crea la rama estable de *develop*. Se trata de la rama estable de la que se crearán ramas de desarrollo nuevo o correcciones. Unicamente de las fusiones de esta rama se puede subir código a la rama *master*. (NUNCA se desarrolla sobre ella).
3. Se crea una rama nueva (feature1) con el sufijo *feature* porque hay que añadir una nueva funcionalidad. La rama tendrá el término de feature como sufijo de su nombre.
4. Se crea una segunda rama (feature2) con el sufijo *feature* para un segundo desarrollo.
5. El primer desarrollo llega a su fin, se fusiona con la rama *develop* y de ahí se libera una versión *release.*
6. De la *release* se fusiona toda la versión en *master* y se crea un nuevo TAG v0.2.0 para indicar la nueva versión enviada a *producción.*
7. La modificación realizada en la *release* se debe añadir también a la rama *develop* para que no haya diferencia entre *master* y *develop*.
8. De la versión v0.2*.*0 se detecta un error en producción, por lo que se crea una nueva rama esta vez de nombre *hotfix.*
9. Se corrige el error y se fusiona tanto en *master* como en *develop*, para que esté corregida en las dos ramas principales.
10. De esta corrección se crea un TAG v0.2.1, indicando que se trata de una corrección de la versión v0.2.0.
11. El segundo desarrollo de *feature2* ha sido fusionado también en la rama *develop* al finalizarlo. Se crea una nueva rama *Release* para la subida de la siguiente versión.
12. En esta rama se modifican problemas detectados durante las pruebas y se fusiona en *master* la siguiente versión.
13. Se crea un TAG de esa versión nueva v0.3.0.
14. Se fusionan los cambios de la rama *release* en *develop* para que *master* y *develop* contengan el mismo código.

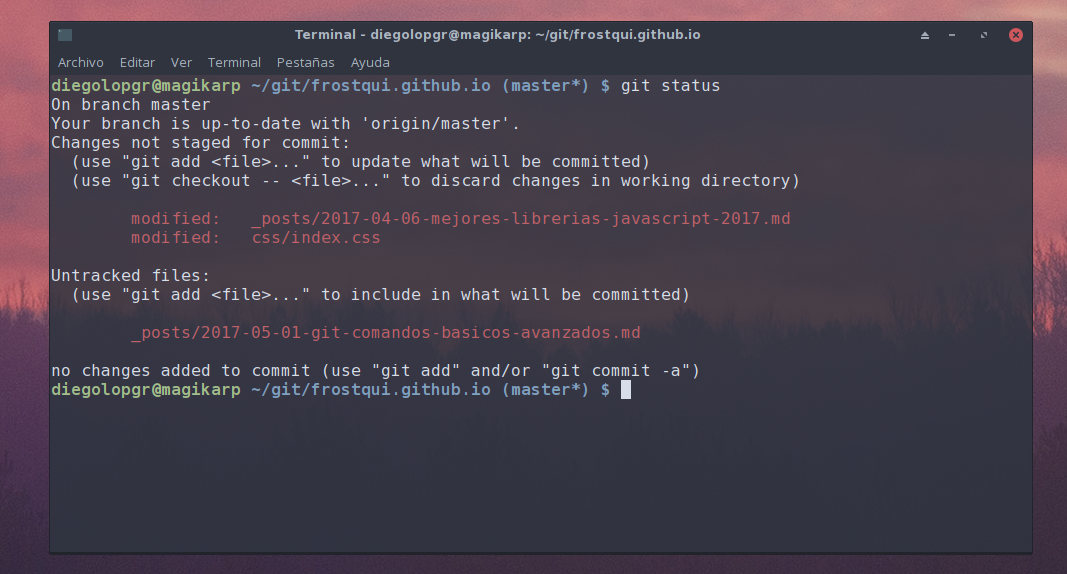
# Versiones de Git: comandos o gráfica

Existen muchas formas de usar Git. Por un lado, tenemos las herramientas originales de línea de comandos, y por otro lado tenemos una gran variedad de interfaces de usuario con distintas capacidades. **En esta unidad didáctica vamos a utilizar Git desde la línea de comandos**. La línea de comandos es el único lugar en donde puedes ejecutar todos los comandos de Git - la mayoría de interfaces gráficas de usuario solo implementan una parte de las características de Git por motivos de simplicidad.

**Herramienta grafica Git (GitKraken)**



**Consola de comandos Git**



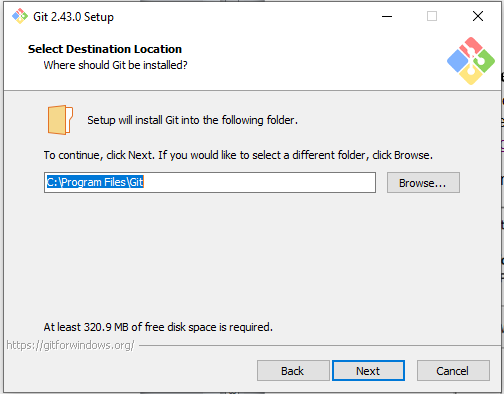
# Instalación de Git

Visitamos la página de Git <https://git-scm.com> y buscamos dónde podemos descargar las versiones para los diferentes sistemas operativos, la versión de Windows se encuentra en <https://gitforwindows.org>. Una vez descargado lo ejecutamos haciendo doble clic.

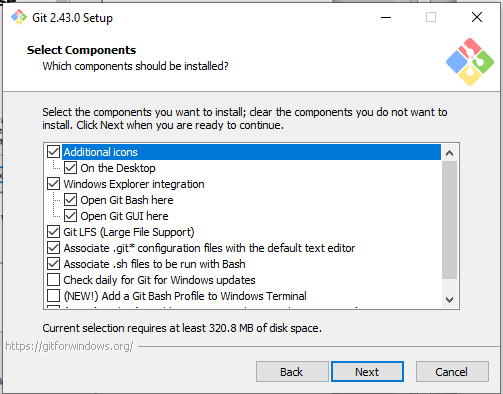
Al iniciar la instalación es preciso aceptar la licencia del programa:



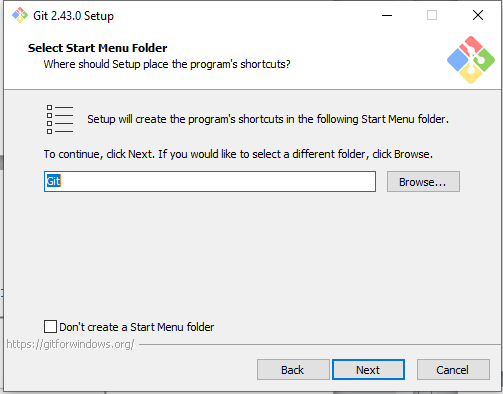
En la siguiente pantalla se define el directorio del equipo en el que se va a instalar la aplicación. Se puede modificar la ubicación por defecto que aparece:



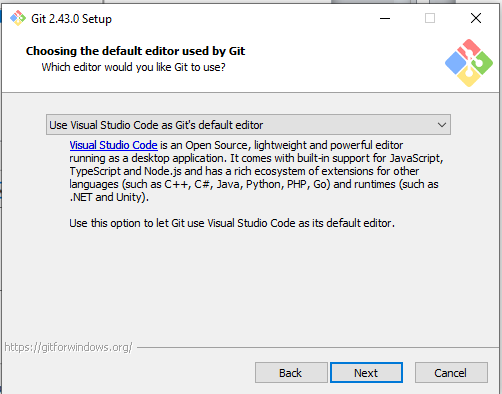
A continuación es preciso seleccionar los componentes que se quieren instalar:



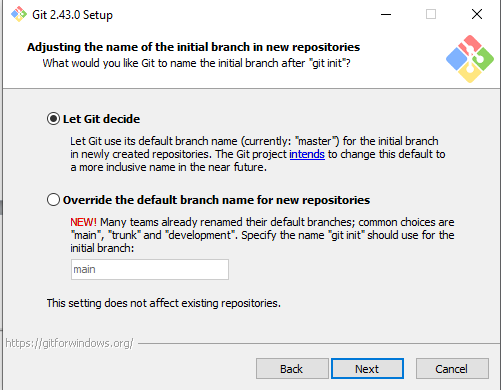
Después es preciso indicar el nombre de la carpeta del Menú de Inicio:



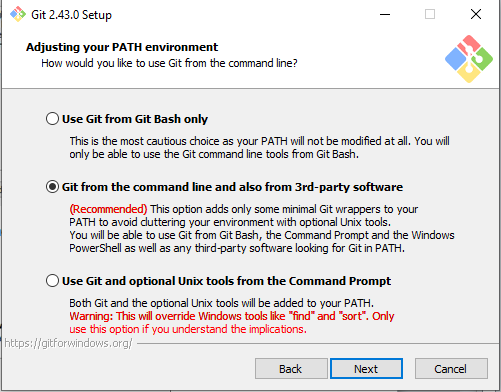
Luego hay que indicar el editor por defecto para Git:



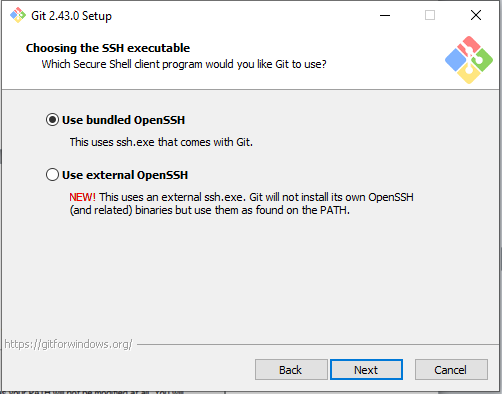
A continuación se ajusta el nombre de la rama inicial en los nuevos repositorios de Git:



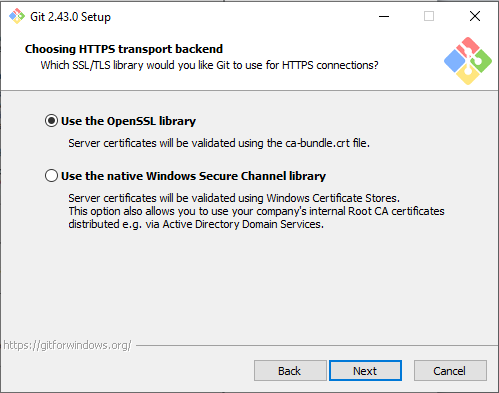
Después se elige la forma de usar Git desde la línea de comandos:



A continuación se elige programa cliente SSH que va a usar Git:

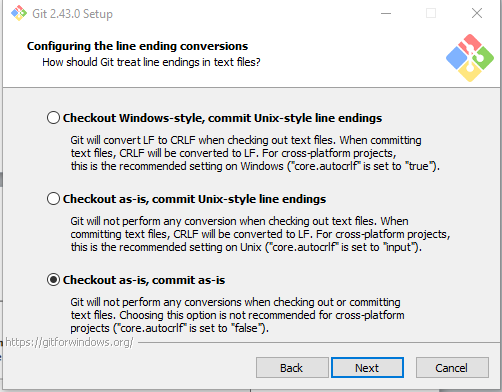


Después se elige la librería SSL/TLS que Git usrá para las conexions HTTPS:

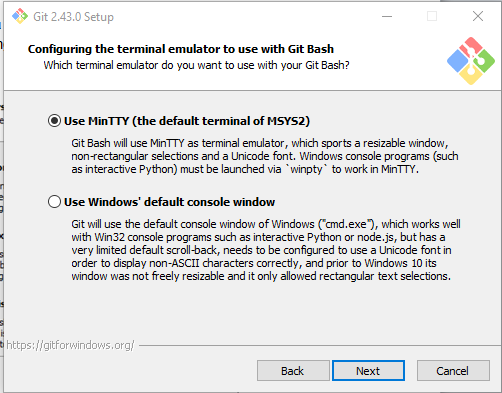


En la siguiente ventana indicamos como serán los saltos de línea, porque ***Linux*** y ***Windows*** tienen diferentes formatos de saltos de línea para los archivos de texto. Es importante indicarlo, porque si se escribe un archivo con saltos de línea de Windows, otra persona puede tener problemas para abrir ese archivo en Linux y viceversa. Si vas a trabajar siempre con el mismo sistema operativo se selecciona ***Checkout as-is, commit as-is***.

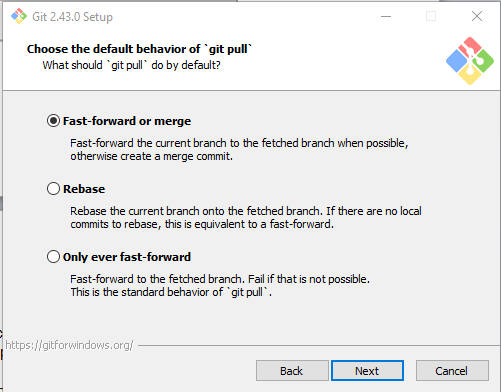
Si seleccionamos la primera opción cada vez que se haga un commit nos visualizará una advertencia que se hace un checkout al estilo Windows y commit al estilo Linux. Evitamos el mensaje seleccionando la tercera opción.



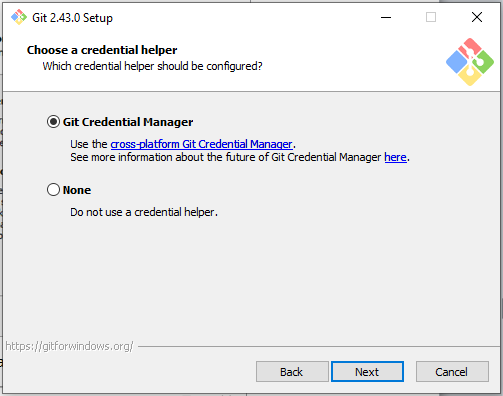
Pulsamos ***Next***, a continuación, pide seleccionar qué consola queremos utilizar, seleccionamos la opción por defecto ***MinTTY***. Pulsamos ***Next***, en la siguiente ventana se dejan habilitadas las opciones que aparecen, y se pulsa el botón Install. Al finalizar la instalación observaremos que Git aparecerá integrado en el explorador de Windows, y lo veremos asociado al menú contextual de las carpetas. Lo que nos permitirá abrir la consola Git en la carpeta.



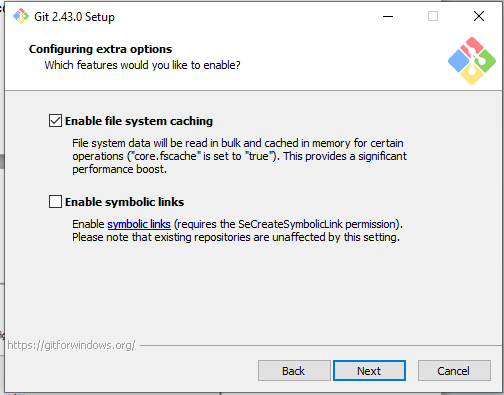
A continuación se selecciona el comportamiento por defecto del comando **git pull**:



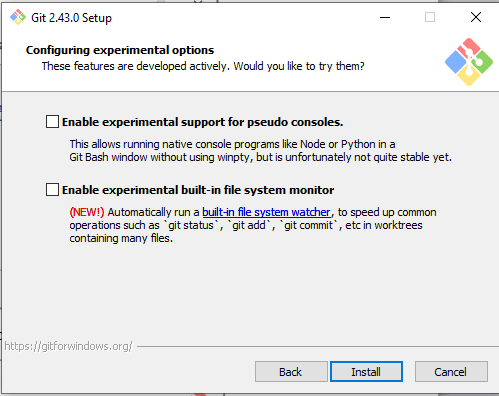
Después se selecciona el gestor de credenciales a configurar:



A continuación se seleccionan unas características a activar:



Luego se seleccionan una serie de opciones de configuración experimentales:



A continuación comienza la instalación de las diferentes opciones seleccionadas y, tras unos minutos, la instalación finaliza.

Para comprobar qué versión de Git está instalada, sigue los siguientes pasos:

* Abre la línea de comandos o cualquier otra interface de línea de comandos.
* Escribe **git --version** y pulsa la tecla Enter para ejecutar el comando.

**$git --version**

Git version 2.30.2.windows.1

# Comandos principales de Git

En este apartado se va a utilizar la consola de Git, la ***Git Bash***. Aunque también se puede trabajar con un interfaz gráfica ***Git GUI*** pero es muy limitada, es más apropiado aprender a utilizar los comandos básicos de Git.

## Instrucciones básicas de Linux

En el modo comando de Git se utilizarán las instrucciones básicas de Linux para crear carpetas, directorios y ficheros.

|  |  |
| --- | --- |
| Comando | Ejemplo |
| **cd** – Navega aun directorio | **cd ..** Retrocede al directorio anterior.  **cd carpeta1** Se mueve a la*carpeta1* |
| **ls** – Muestra el contenido de un directorio | **ls** Visualiza ficheros  **ls -l** Visualiza toda la información: usuario, grupos, fechas.  **ls -la** Visualiza también los ficheros ocultos. |
| **mkdir** - Crea un directorio | **mkdir midirectorio** Crea *midirectorio*. |
| **pwd** – Directorio actual | **pwd** Visualiza el nombre del directorio actual. |
| **rmdir** – Borra directorios | **rmdir direc1** Borra el directorio *direc1* si está vacío.  **rmdir -p direc1** Borra el directorio *direc1*si está vacío, y todos los que tenga dentro, si están vacíos. |
| **rm** – Borra ficheros y directorios | **rm file1.txt** Borra el fichero *file1.txt* y pide confirmación.  **rm \*** Borra todos los ficheros del directorio.  **rm -d direc1** Borra el directorio *direc1* si está vacío.  **rm -r direc2** Borra el directorio *direc2* y el contenido de todos sus subdirectorios recursivamente. |
| **mv** – Renombra y mueve ficheros y directorios de una ubicación a otra | **mv archivo1 archivo2** Renombra *archivo1* como *archivo2* si existen y son ficheros.  **mv dir1 dir2** Mueve el directorio *dir1* dentro de *dir2*. Si *dir2* no existe, *dir1* se renombrará como *dir2*. |
| **cat** – Entre otros, puede usarse para visualizar el contenido de un fichero | **cat fichero.txt** Visualiza el contenido de *fichero.txt* en la consola. |
| **echo** – Permite introducir texto en una salida | **echo Mensaje que va a la consola** Visualiza el mensajes en la consola.  **echo “Añado datos”>fichero.txt** Añade la línea de texto “*Añado datos*” al fichero *fichero.txt*. Si el fichero no existe, se crea.  **echo “Añado otra línea”>>fichero.txt** Añade otra línea de texto al fichero. |

## Ayuda

Si tienes problemas para recordar comandos u opciones para comandos, puedes usar la ayuda de Git.

Hay un par de maneras diferentes de utilizar el comando de ayuda en la línea de comandos:

**git help --all** - Ver todos los posibles comandos

**git command -help** - Ver todas las opciones disponibles para el comando específico (por ejemplo, ayuda para el comando git push)

**$git push -help**

usage: git push [<options>] [<repository> [<refspec>...]]

-v, --verbose be more verbose

-q, --quiet be more quiet

--repo <repository> repository

--all push all branches

--branches alias of --all

--mirror mirror all refs

-d, --delete delete refs

--tags push tags (can't be used with --all or --branches or --mirror)

-n, --dry-run dry run

--porcelain machine-readable output

-f, --force force updates

--force-with-lease[=<refname>:<expect>]

require old value of ref to be at this value

--force-if-includes require remote updates to be integrated locally

--recurse-submodules (check|on-demand|no)

control recursive pushing of submodules

--thin use thin pack

--receive-pack <receive-pack>

receive pack program

--exec <receive-pack> receive pack program

-u, --set-upstream set upstream for git pull/status

--progress force progress reporting

--prune prune locally removed refs

--no-verify bypass pre-push hook

--follow-tags push missing but relevant tags

--signed[=(yes|no|if-asked)]

GPG sign the push

--atomic request atomic transaction on remote side

-o, --push-option <server-specific>

option to transmit

-4, --ipv4 use IPv4 addresses only

-6, --ipv6 use IPv6 addresses only

## Crear un repositorio local

Creamos una carpeta en nuestro disco duro, llamada ***repositorio1***, pulsamos el botón derecho del ratón y elegimos ***Git Bash*** sobre la carpeta, se abrirá la consola de ***Git***.

La orden para crear un repositorio de Git es **git init.** Con esta orden hago que mi carpeta “*repositorio1*” sea capaz de albergar un repositorio de Git. Al ejecutar la orden observaremos que en nuestra carpeta se crea un subdirectorio oculto ***.git***. Este subdirectorio contiene los metadatos que Git utiliza para almacenar su información.

Con la orden **ls -la** veremos el directorio oculto. ***Git status*** nos permitirá visualizar los cambios realizados en el área de trabajo y en el área de preparación comparando con lo que hay en el repositorio local. Si tecleamos ***git status***, se verá el estado actual del área de trabajo, nos indica que estamos en la rama ***master***, y que no hay nada para hacer commit.

**$git status**

On branch master

No commits yet

Nothing to commit (create/copy files and use “git add” to track).

## Añadir ficheros al área de trabajo

A continuación, creamos 3 archivos para hacer pruebas. Utilizamos ***echo***, y almacenamos un mensaje en cada uno. Los visualizamos, y volvemos a mostrar con ***git status*** el estado de los cambios:

**$echo “Fichero prueba 1” > fichero1.txt**

**$echo “Fichero prueba 2” > fichero2.txt**

**$echo “Fichero prueba 3” > fichero3.txt**

**$ ls –la**

Total 19

drwxr-xr-x alumno 197121 0 may. 9 18:07 ./

drwxr-xr-x alumno 197121 0 may. 9 18:07 ../

drwxr-xr-x alumno 197121 0 may. 9 18:07 .git/

-rwxr-xr-x alumno 197121 0 may. 9 18:07 fichero1.txt

-rwxr-xr-x alumno 197121 0 may. 9 18:07 fichero2.txt

-rwxr-xr-x alumno 197121 0 may. 9 18:07 fichero3.txt

alumno@w10virtual MINGW64 /c/repositorio1 (master)

**$cat fichero1.txt**

Fichero prueba 1

alumno@wvirtual MINGW64 /c/repositorio1 (master)

**$git status**

On branch master

No commits yet

Untracked files:

(use “git add <file> …” to include in what will be commited)

fichero1.txt

fichero2.txt

fichero3.txt

nothing added to commit but untracked files present (use “git add” to track)

alumno@wvirtual MINGW64 /c/repositorio1 (master)

Si observas ahora el estado del área de ***trabajo***, se muestran los tres ficheros creados y estos aparecen en rojo, y se indica que se han modificado (***untracked***). En el área de ***trabajo*** aparecerán siempre las modificaciones que se han hecho en la carpeta de trabajo, y que no se encuentran validadas. Si los queremos validar a una versión tenemos que usar la orden **git add <file>**, es decir, añadirlos al área de preparación.

Para guardar o validar (commit) los ficheros en el repositorio Git, primero hay que añadirlos al ***área de preparación***. Esta zona intermedia va a permitir que sólo se validen los ficheros que están en el área de preparación.

## Añadir ficheros al área de preparación

Lo siguiente a hacer es añadir los ficheros al área de preparación. Utilizamos la orden **git add <nombrefichero>.** Podemos utilizar **git add <directorio>** para añadir un directorio, o también podemos poner **git add .** y añade todo el contenido del directorio actual.

**$git add fichero1.txt**

alumno@wvirtual MINGW64 /c/repositorio1 (master)

**$git status**

No commits yet

Changes to be commited:

(use “git rm --cached <file> …” to unstage)

New file: fichero1.txt

Untracked files:

(use “git add <file> …” to include in what will be commited)

fichero2.txt

fichero3.txt

alumno@wvirtual MINGW64 /c/repositorio1 (master)

Se muestra en color verde un fichero que puede ser validado (fichero1.txt) y que está en el área de preparación. Los otros dos ficheros (fichero2.txt y fichero3.txt) se muestran en color rojo y están en el área de trabajo.

## Aprobar los cambios

Probamos ahora a validar los cambios, utilizamos la orden **git commit –m “Mensaje de validación”**. La orden lleva un flag para añadir un mensaje a la hora de validar los cambios. Si valido los cambios me dirá lo siguiente:

**$git commit –m “Valido el fichero 1”**

\*\*\* Please tell me who you are.

Run

git config --global user.email “you@example.com”

git config --global user.name “Your Name”

to set you account’s default identity.

Omit --global to set the identity only in this repository.

Fatal: unable to auto-detect email address (got ‘alumno@w10virtual.(none)’)

alumno@wvirtual MINGW64 /c/repositorio1 (master)

No deja validar, pues cada vez que se hace ***commit*** se necesita saber qué usuario es el que ha realizado la validación. Se necesita configurar los datos del usuario (nombre y correo electrónico), sólo la primera vez.

**$ git config --global user.email “alicia@example.com”**

**$ git config --global user.name “Usualicia”**

**$ git commit -m “Valido el fichero 1.”**

[master (root-commit) 932e8ae] Valido el fichero 1.

1 file changed, 1 insertion(+)

create mode 100644 fichero1.txt

Al hacer commit nos aparecerá un identificativo o ***apuntador*** de la versión creada y con la cabecera [*master (root-commit) id*], en este caso el master va a ser HEAD, es decir, la versión mas reciente. Si volvemos a visualizar el estado del área de trabajo, seguiremos viendo que los ficheros 2 y 3 están en estado modificado. Los añadimos al área de preparación y los validamos. Se genera otro id.

**$ git status**

**$ git add .**

**$ git status**

**$ git commit -m “Valido los ficheros 2 y 3.”**

[master 17494fb] Valido los ficheros 2 y 3.

2 files changed, 2 insertions(+)

create mode 100644 fichero2.txt

create mode 100644 fichero3.txt

## Fichero .gitignore para poder excluir ficheros especificos

El fichero .gitignore es una característica esencial de Git que permite especificar intencionalmente qué archivos y directorios deben ser ignorados por Git. Esto es útil para evitar incluir archivos no deseados en el control de versiones, como archivos temporales, de configuración local, o binarios que no deberían formar parte del repositorio.

La sintaxis dentro de este fichero permite crear una estructura de control de los ficheros a controlar:

* **Comentarios**: Las líneas que comienzan con # son comentarios y son ignoradas por Git.

# Este es un comentario

* **Ignorar Archivos o Directorios**: Para ignorar un archivo o directorio específico, simplemente escribe su nombre o ruta relativa.

/build # Ignora la carpeta 'build' en la raíz del proyecto

\*.log # Ignora todos los archivos con extensión .log

* **Patrones de Coincidencia**: Utiliza comodines para ignorar archivos que coincidan con un patrón específico.

\*.tmp # Ignora todos los archivos .tmp

* **Negaciones**: Utiliza el símbolo de exclamación ! para especificar archivos o directorios que no deben ser ignorados, incluso si coinciden con una regla anterior.

\*.txt # Ignora todos los archivos .txt

!important.txt # No ignores important.txt

* **Ignorar Archivos en Subdirectorios**: Si deseas ignorar archivos o directorios en cualquier lugar del proyecto, no utilices la barra inclinada inicial.

logs/ # Ignora la carpeta 'logs' en cualquier parte del proyecto

* **Ignorar Archivos y Directorios Específicos**: Para ignorar archivos específicos dentro de un directorio particular.

/docs/\*.pdf # Ignora todos los archivos .pdf dentro del directorio 'docs'

Si queremos crear un fichero .gitignore lo primero que hay que hace es crear un fichero dentro del repositorio en la raíz del proyecto. Para ello utilizamos el comando “**touch .gitignore”.** Una vez creado se debe editar para añadir los elementos que queramos ignorar en base a la sintaxis explicada previamente. Una vez editado lo añadiremos al área de preparación con el comando **git add .gitignore** y seguido lo validaremos en el repositorio con un **git commit -m “mensaje”**

$touch .gitignore

$git add .gitignore

$git commit -m "Añadido archivo .gitignore para excluir archivos innecesarios"

Un ejemplo práctico de un fichero .gitignore podría ser el siguiente ejemplo. Supongamos que tienes un proyecto en el que quieres ignorar todos los archivos de log, el directorio node\_modules y los archivos de configuración local. Tu archivo .gitignore podría verse así:

# Ignorar directorio de dependencias

node\_modules/

# Ignorar archivos de registros

\*.log

# Ignorar archivos de configuración local

config/local.json

## Ver versiones creadas

Una vez validados los cambios, para ver las versiones del repositorio utilizo el comando **git log**. Se muestran los ***commit*** que se han hecho, el usuario que lo ha hecho, la fecha y los mensajes de validación, y los apuntadores de cada versión. La versión ***HEAD -> Master***, es la última versión, la versión actual y la versión más reciente.

**$ git log**

Commit 17494fb5bdc08775a489fc7fe04dfbecc41ec6ac (HEAD -> Master)

Author: Usualicia <alicia@example.com>

Date: Sat May 9 19:19:51 2020 +0200

Valido los ficheros 2 y 3.

Commit 932e8ae54b2e76fa2aa54ffe3395640baf490ea8

Author: Usualicia <alicia@example.com>

Date: Sat May 9 18:59:01 2020 +0200

Valido los ficheros 1.

Para ver las versiones, en forma más abreviada utilizamos la orden **git log -–oneline**.

**$ git log --oneline**

17494fb (HEAD -> Master) Validos los ficheros 2 y 3.

932e8ae Valido el fichero 1.

alumno@w10virtual MINGW64 /c/repositorio1 (master)

El formato de la salida del log lo podemos poner de forma que las salidas sean más ordenadas, por ejemplo, para sacar cada commit en una única línea, el comando es:

**$ git log --pretty=format:”%h %an %ar - %s”**

**$ git log --pretty=”%h %an %ar - %s”**

Hay otro comando que permite ver los cambios realizados en todos los commits, o en un commit concreto, en este caso se añade el id o apuntador:

**$ git show**

**$ git show idCommit**

## Añadiendo y deshaciendo cambios

Vamos a hacer cambios en un fichero del repositorio para que pase al área de trabajo, y lo vamos a comparar con el mismo fichero que está en el área de commit.

Utilizamos el comando **git diff**. Permite comparar cambios que hay en el área de trabajo contra una versión previamente aprobada. Por defecto compara el área de trabajo y el área de commit que llamamos HEAD, la versión más reciente.

Añadimos un cambio al fichero *fichero1.txt*. Lo editamos con el editor ***nano fichero1.txt***, y añadimos la línea “*Línea nueva en fichero1*”. A continuación ejecutamos el comando **git diff**.

**$ nano fichero1.txt**

alumno@w10virtual MINGW64 /c/repositorio1 (master)

**$ git diff**

diff --git a/fichero1.txt b/fichero1.txt

index a1be692..49b0467 100644

--- a/fichero1.txt

+++ a/fichero1.txt

@@ -1 +1, 2 @@

**Fichero prueba1**

**$ git log --pretty=format:”$h %an %ar - %s”**

+Línea nueva en fichero 1

alumno@w10virtual MINGW64 /c/repositorio1 (master)

Indica en qué ficheros ha detectado cambios, y los cambios detectados. A un fichero lo identifica con *a/fichero1.txt* será el que está en el área de ***commit***, y al otro, con el nombre *b/fichero1.txt*, que está en el área de ***trabajo***. Y también indica con los signos + y - los cambios.

También podemos consultar la diferencia entre varios commits indicando los apuntadores. Por ejemplo, calculo la diferencia enter el commit ***HEAD*** y dos momentos anteriores.

**$ git diff HEAD~2 HEAD**

Por defecto, **git diff** solo comparará lo que hay en área de ***trabajo*** con lo que hay en el área de commit. Si se desea comparar lo que está en el área de preparación con lo que hay en el área de trabajo, hay que utilizar el comando **git diff --staged**.

Cuando realicemos cambios para modificar de nombre un fichero (**mv**), o borrarlo (**rm**), es necesario hacerlo con git (**git mv** o **git rm**), porque si lo hacemos utilizando los comandos de renombrar o borrar del sistema operativo, normalmente Linux, Git no se enterará, y no recogerá esos cambios ni en el ***área de trabajo*** ni en ***el área de commit***.

Cuando se realizan cambios de este tipo, de borrado o nombrado, **los cambios se registrarán en el área de trabajo**. Si hacemos **ls –la** no veremos los ficheros borrados o directorios borrados. Pero en el ***área de trabajo*** esos cambios sí se mostrarán.

Para demostrar como actúa Git ante los cambios de nombres de ficheros vamos a cambiar el nombre a los ficheros ***fichero1.txt*** y ***fichero2.txt***. Pero antes validaremos los cambios realizados al *fichero1.txt*.

**$ git add fichero1.txt**

**$ git commit -m "Valido cambios en fichero 1"**

[master ba9e605] Valido cambios en fichero 1

1 file changed, 1 insertion(+)

**$ git log --oneline**

ba9e605 (HEAD -> master) Valido cambios en fichero 1

17494fb Valido los ficheros 2 y 3

932e8ae Valido el fichero 1.

Cambiaremos el nombre del *fichero1.txt* por el nombre *fichero11.txt* utilizando el comando **mv.** Al utilizar este comando, Git interpretará que hemos borrado el *fichero1.txt* y hemos creado un nuevo fichero llamado *fichero11.txt*.

**$ mv fichero1.txt fichero11.txt**

**$ ls -la**

total 19

drwxr-xr-x alumno 197121 0 may. 9 18:07 ./

drwxr-xr-x alumno 197121 0 may. 9 18:07 ../

drwxr-xr-x alumno 197121 0 may. 9 18:07 .git/

-rwxr-xr-x alumno 197121 0 may. 9 18:09 fichero1.txt

-rwxr-xr-x alumno 197121 0 may. 9 18:07 fichero2.txt

-rwxr-xr-x alumno 197121 0 may. 9 18:07 fichero3.txt

**$ git status**

On branch master

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

deleted: fichero1.txt

Untracked files:

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

fichero11.txt

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

Como se puede observar los cambios no se añaden al área de preparación. Por eso hay que añadir el nuevo fichero, *fichero11.txt*, al área de preparación con **git add** y eliminar del repositorio el *fichero1.txt* con **git rm**.

**$ git add fichero11.txt**

**$ git status**

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: fichero11.txt

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

deleted: fichero1.txt

**$ git rm fichero1.txt**

rm 'fichero1.txt'

**$ git status**

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

renamed: fichero1.txt -> fichero11.txt

**$ git commit -m "Valido el cambio de nombre de fichero 1 a 11"**

[master fec5ee2] Valido el cambio de nombre de fichero 1 a 11

1 file changed, 0 insertions(+), 0 deletions(-)

rename fichero1.txt => fichero11.txt (100%)

**$ git log --oneline**

fec5ee2 (HEAD -> master) Valido el cambio de nombre de fichero 1 a 11

ba9e605 Valido cambios en fichero 1

17494fb Valido los ficheros 2 y 3

932e8ae Valido el fichero 1.

Como se puede observar, después de añadir los dos cambios al área de preparación, Git informa que se ha producido un cambio de nombre. Ahora vamos a cambiar el nombre del *fichero2.txt* utilizando el comando **git mv**.

**$ git mv fichero2.txt fichero22.txt**

**$ ls -la**

total 15

drwxr-xr-x alumno 197121 0 may. 9 18:07 ./

drwxr-xr-x alumno 197121 0 may. 9 18:07 ../

drwxr-xr-x alumno 197121 0 may. 9 18:07 .git/

-rwxr-xr-x alumno 197121 0 may. 9 18:09 fichero11.txt

-rwxr-xr-x alumno 197121 0 may. 9 18:11 fichero22.txt

-rwxr-xr-x alumno 197121 0 may. 9 18:07 fichero3.txt

**$ git status**

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

renamed: fichero2.txt -> fichero22.txt

**$ git commit -m "Valido el cambio de nombre de fichero 2 a 22"**

[master 489235b] Valido el cambio de nombre de fichero 2 a 22

1 file changed, 0 insertions(+), 0 deletions(-)

rename fichero2.txt => fichero22.txt (100%)

**$ git log --oneline**

489235b (HEAD -> master) Valido el cambio de nombre de fichero 2 a 22

fec5ee2 Valido el cambio de nombre de fichero 1 a 11

ba9e605 Valido cambios en fichero 1

17494fb Valido los ficheros 2 y 3

932e8ae Valido el fichero 1.

Como se puede observar al utilizar el comando ***git mv*** se cambia el nombre del fichero en el área de trabajo y se añade este cambio directamente al área de preparación sin tener que utilizar **git add** y **git rm**.

## Deshaciendo cambios

Es importante tener en cuenta que los cambios que se hacen en el área de trabajo, si se borran del área de trabajo no se pueden recuperar, se recuperarán si antes han pasado por las áreas de preparación o de commit.

El comando **git checkout** reemplazará todo lo que tengamos en el área de trabajo con la última versión validada, es decir, nos va a traer la versión HEAD del área de commit, la última versión.

**git checkout .** , todo el contenido del área de trabajo, el directorio actual de trabajo (.), será reemplazado por lo que haya atualmente en el área de commit. Si hubiese algún cambio en el área de preparación, esos cambios van a permanecer.

**git checkout --<nombre\_fichero>** Para hacer checkout de un fichero concreto.

**git checkout <nombre\_rama>** Para hacer checkout de una rama.

Por ejemplo, borro un fichero y lo recupero (se supone que se validó y está en el HEAD):

**$ rm fichero1.txt**

**$ git checkout .**

**git checkout**, no toca el área de preparación. Para eso tenemos el comando ***git reset***.

Si tenemos algo en el área de preparación y queremos eliminarlo de preparación, y devolverlo al área de working entonces usamos **git reset HEAD <fichero>**

Para pasar de preparación a trabajo utilizamos **git reset**.

Por ejemplo, borro un fichero con Git, este va directamente a preparación. Si lo quiero sacar de ahí escribo **git reset HEAD fichero11.txt**:

**$ git rm fichero11.txt**

**$ git status**

**$ git reset HEAD fichero11.txt**

**$ git status**

Con **git restore <nombredefichero>** devolvemos el fichero al estado anterior a los cambios. El fichero de salida estaba en el área de trabajo, y va a salir del área de trabajo sin cambios:

**$ echo “Creo otra linea” >> fichero3.txt**

**$ cat fichero3.txt**

**$ git status**

**$ git restore fichero3.txt**

**$ cat fichero3.txt**

**$ git status**

El comando **git reset --hard** combina tanto el efecto de **un git checkout** como el de un **git reset** en un único comando. El resultado es que se eliminarán los ficheros del área de preparación y del área de trabajo de tal forma que volvemos a los mismos contenidos que estaban presentes en el último commit.

Probamos lo siguiente: el fichero3 se borra, y directamente pasa al área de preparación, el fichero11 se modifica, y se añade al área de preparación con *git add*, y el fichero22 se modifica y está en el área de trabajo:

**$ git rm fichero3.txt**

**$ echo “Creo línea” >> fichero11.txt**

**$ git add fichero11.txt**

**$ echo “Creo línea” >> fichero22.txt**

**$ git add fichero22.txt**

**$ git status**

Limpio lo que hay en trabajo y preparación con ***reset hard***, borro todos los cambios que hice anteriormente y me traigo lo que hay en el HEAD (**git reset --hard HEAD**). Visualizo el fichero3.txt y veo que tiene los datos anteriores al reset y que no está borrado:

**$ git log --oneline**

**$ git reset --hard HEAD**

**$ cat fichero3.txt**

Si queremos volver a una instantánea, a un momento en el que hicimos un commit, utilizamos el comando **git reset --hard <apuntadorDeValidacion>**. El apuntador de cada commit se muestra cuando vemos el log con **git log –-online.**

**$ git log --oneline**

abb9bf6 (HEAD -> Master) Valid

965ec1d Valido los ficheros 3

489235b (HEAD -> master) Valido el cambio de nombre de fichero 2 a 22

fec5ee2 Valido el cambio de nombre de fichero 1 a 11

ba9e605 Valido cambios en fichero 1

17494fb Validos los ficheros 2 y 3

932e8ae Valido el fichero 1.

Si quiero volver al momento en que creé los tres ficheros, y añadí una línea a cada uno de ellos, sería el momento del log ba9e605, habré perdido todo lo realizado en los momentos posteriores.

**$ git reset --hard ba9e605**

**$ ls -la**

**$git log --oneline**

**$ git status**

Se pueden deshacer los últimos cambios realizados en el último commit. Para ello utilizamos la orden **git revert HEAD**.

Modificamos un fichero, y lo validamos. Si queremos deshacer esos cambios se creará un nuevo commit con la operación deshecha, además visualizará una pantalla con los cambios que va a revertir.

**$ echo “Creo otra linea” >> fichero1.txt**

**$ echo “Creo otra linea” >> fichero1.txt**

**$ git add fichero1.txt**

**$ git commit -m “Valido cambio para revert en 1”**

**$ git log --oneline**

f05d505 (HEAD -> Master) Valido cambio para revert en 1

ba9e605 Valido cambios en fichero 1

17494fb Valido los ficheros 2 y 3

932e8ae Valido fichero 1

**$ git revert HEAD**

**$ git log --oneline**

5ca7217 (HEAD -> Master) Revert “Valido cambio para revert en 1”

42a9397 Valido cambio para revert en 1

f05d505 Valido cambio para revert en 1

ba9e605 Valido cambios en fichero 1

17494fb Valido los ficheros 2 y 3

932e8ae Valido fichero 1

Ahora queremos revertir todos los cambios a uno de los momentos iniciales, al momento en el que se han validado los fichero1 y fichero2, 17494fb

**$ git revert 17494fb**

Removing fichero3.txt

CONFLICT (modify/delete): fichero2.txt deleted in parent of 17494fb… Valido los ficheros 2 y 3 and modified HEAD. Version HEAD of fichero2.txt left is tree.

Error: could not revert 17494fb… Valido los ficheros 2 y 3

Hint: after resolving the conflicts, mark the corrected paths

Hint: with ‘git add <paths>’ or ‘git rm <paths>’

Hint: and commit the result with ‘git commit’

Al hacer **git revert** pueden producirse conflictos, puede ser que se pida revertir a una situación, en la que Git no resuelve y tenemos que decidir qué hacer.

Por ejemplo, que encuentre archivos que exiten en HEAD y no existe en la instantanea a la que se desea acceder. En el ejemplo dice que hay conflicto en el *fichero2.txt*, que fue eliminado del commit 17494fb al que queremos revertir, pero aparece modificado en el HEAD.

Si mostramos el estado del área de trabajo nos va a sugerir posibles acciones, y nos va a indicar los cambios que va a realizar. Aparecerán en verde los cambios aceptados, y en rojo los que entran en conflicto, aceptado es el borrado del fichero3, pero no el borrado del fichero2, y una de las sugerencias es que este fichero se borre:

**$ git status**

On branch master

You are currently reverting commit 17494fb.

(fix conflicts and run “git revert –continue”)

(use “git revert –skip” to skip this patch)

(use “git revert –abort” to cancel the revert operation)

Changes to be committed:

(use “git restore –staged <file> … “ to unstage)

deleted: fichero3.txt

Unmerged paths:

(use “git restore --staged <fiule> …” to unstage)

(use “git add/rm <file>… “ as appropiate to mark resolution)

deleted bt them: fichero2.txt

Para resolver el conflicto podemos saltarnos el paso con **git revert –skip**, o podemos abortar la orden **git revert --abort**, o también podemos solucionar el conflicto.

Nos fijamos en los cambios que no se pueden hacer, aparecerán en rojo y dentro de *Unmerged paths*, y lo que sugiere es que borremos el fichero: deleted by them: fichero2.txt. Así pues, se borra y se visualiza de nuevo el estatus del área de trabajo. Ya no aparecerán cambios en “rojo”, ya indica los cambios a validar. Continuamos con la operación revert, y se mostrará en una ventana que nos dirá los cambios que se van a realizar, podemos añadir un comentario a la operación y finalmente aparece todo validado.

**$ git rm fichero2.txt**

**$ git status**

On branch master

You are currently reverting commit 17494fb.

(fix conflicts and run “git revert –continue”)

(use “git revert –skip” to skip this patch)

(use “git revert –abort” to cancel the revert operation)

Changes to be committed:

(use “git restore –staged <file> … “ to unstage)

deleted: fichero2.txt

deleted: fichero3.txt

**$ git revert –continue**

**$ git status**

**$ ls -la**

**$ git log --oneline**

También se pueden revertir los cambios desde un momento a otro momento, es decir de múltiples commits, se utilizará el ~ para significar “menos”. Por ejemplo, HEAD~2, son dos commits desde HEAD. Esto puede combinarse con los caracteres “…” que indican el rango entre dos commits.

Por ejemplo, ponemos: **git revert HEAD...HEAD~6** para revertir los commits hechos entre HEAD y HEAD~6. Irá a 6 commits hacia atrás. Para cada validación mostrará la ventana de cambios, y podemos añadir mensaje.

Si en un revert no queremos pasar uno por uno los cambios del rango seleccionado, se pueden quitar añadiendo *-–no-edit*.

**$ git revert HEAD...HEAD~3 --no-edit**

## Gestión de ramas

Ya hemos visto que por defecto *Git* llama ***master*** a la ***rama principal*** del proyecto o del desarrollo. Cuando hablamos de ramificaciones, significa que cogemos la rama principal de desarrollo (***master***) y a partir de ahí continuamos trabajando añadiendo nuevas funciones, sin seguir la rama master.

*Git* maneja las ramificaciones de forma muy rápida, promueve un ciclo de desarrollo donde las ramas se crean y se unen entre sí, incluso varias veces en el mismo día. Git va a permitir tener muchas versiones diferentes del código base o desarrollo principal, y ***dentro de nuestra área de trabajo podremos intercambiar ramas sin cambiar de área de trabajo.*** Las ramas son ideales para hacer pruebas y experimentos, y si se decide, ser fusionadas con la rama master.

Cuando se crea una rama simplemente se crea ***un nuevo apuntador*** que podremos moverlo libremente. Si creamos una rama con nombre “pruebas” esto creará un nuevo apuntador apuntando a la misma confirmación donde estés actualmente. El comando para crear ramas es **git branch <nombre-rama> <rama de partida>,** donde la rama de partida por defecto es master.

En el siguiente ejemplo se crea la rama *pruebas*, y hasta que nos cambiemos de rama, todas tienen el mismo apuntador master, pruebas y origin/master apuntan en ese momento a HEAD.

**$ git branch pruebas**

**$ git branch –v –a**

**\*** master fc11dc5 Creo repo3 paso3

pruebas fc11dc5 Creo repo3 paso3

**$ git log –-oneline**

fc11dc5 (HEAD -> master, pruebas) Creo repo3 paso3

73daf15 Nueva linea en repo1 y 2 paso2

Caa0b10 Creo repo1 y repo2 paso1

La rama en la que estamos posicionados es la rama que llevará el apuntador HEAD. Cuando cambiamos de rama, Git cambia los contenidos del directorio del área de trabajo, y cambia el apuntador HEAD a la rama acutal. El comando para cambiar de rama es **git checkout <nombrerama>**.

Cada vez que hagamos el cambio de rama a la rama master local (**git checkout master**) Git nos va a informar de si la rama local está por delante de la rama remota y sugerirá hacer **git push** para publicar las confirmaciones locales y sincronizar.

Nos movemos a la rama creada, creamos varios ficheros, los añadimos al área de preparación y los validamos. Si visualizamos tendremos los ficheros iniciales más los nuevos en la rama ***pruebas***.

**$ git checkout pruebas**

**$ echo “nuevo1” > nuevo1**

**$ echo “nuevo2” > nuevo2**

**$ echo “nuevo3” > nuevo3**

**$ git add nue\***

**$ git commit –m “Ficheros nuevos en la rama”**

**$ ls -la**

**$ git branch –a -v**

Si cambiamos a la rama ***master***, veremos que tenemos los ficheros iniciales. Estamos dentro del mismo ***área de trabajo*** siempre, pero cambiamos de rama de forma muy sencilla.

**$ git checkout master**

**$ git branch –a –v**

**$ ls -la**

## Guardar los cambios del área de trabajo (Git Stash)

El comando **git stash** es una herramienta útil en Git para guardar temporalmente los cambios en el área de trabajo que aún no han sido listados en el área de preparación (staged) ni confirmados (committed). Esto permite que puedas cambiar de rama sin tener que comprometer los cambios incompletos, y luego recuperar estos cambios más tarde.

Supongamos que hemos realizado cambios en distintos ficheros y nos piden trabajar en otra tarea en otra rama. Si cambiara a otra rama sin haber subido al área de preparación, perdería el trabajo realizado hasta el momento. Pero imaginemos que el código no está finalizado. El comando **git stash** guarda los cambios actuales en el área de trabajo y el índice (área de preparación) en una pila temporal y limpia el área de trabajo. Esto te permite trabajar en algo más urgentemente sin perder tus cambios actuales. Para volver a trabajar con el contenido guardado se debe usar el comando **git stash apply** aplica los cambios más recientes guardados en el stash a tu área de trabajo actual. **git stash pop** aplica los cambios del stash más reciente y luego elimina ese stash de la pila.

**$echo "Cambio" >> example.txt**

**$git stash push -m "ultimo"**

**Saved working directory and index state On master: ultimo**

**$git stash list**

**stash@{0}: On master: ultimo**

**stash@{1}: On master: Third change**

**$ git stash pop**

**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: example.txt**

**Untracked files:**

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

**examenple.txt**

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

**Dropped refs/stash@{0} (3e1e528a8c0b0138b81d1f600fbc95d2697bba0c)**

Puedes también especificar un stash particular si hay múltiples stashes. Para poder ver la lista de distintos stash debemos usar el comando **git stash list**, que muestra todos los stashes guardados en la pila, permitiéndote ver y seleccionar cuál deseas aplicar o eliminar mediante su identificativo. **git stash drop stash@{index}** elimina el stash específico. **git stash apply stash@{index}** aplica los cambios del stash especificado sin eliminarlo de la lista. A continuación, se muestra un ejemplo de cómo trabajar con un listado de stashes.

**$echo "Change 1" >> example.txt**

**git stash push -m "First change"git stash list**

**$echo "Change 2" >> example.txt**

**git stash push -m "Second change"**

**$ git stash list**

**stash@{0}: On master: Third change**

**stash@{1}: On master: Second change**

**$ git stash apply stash@{1}**

**$ git stash drop stash@{1}**

**Dropped stash@{1} (5c7a4df73298b53b499870573910121f88372a25)**

**$ git stash list**

**stash@{0}: On master: Third change**

## Fusionar ramas

En esta prueba ralizaremos cambios en la rama ***pruebas***, en uno de los ficheros iniciales (*repo1.txt*), validamos los cambios y luego los cambios los fusionaremos con la rama ***master***.

**$ git checkout pruebas**

**$ echo “Cambio1 desde la rama pruebas” >> repo1.txt**

**$ echo “Otro cambio desde pruebas” >> repo1.txt**

**$ git add repo1.txt**

**$ git commit –m “Cambios en la rama pruebas de repo1.txt”**

Si observamos ahora el log vemos que ***pruebas*** y ***master*** tienen distintos apuntadores. HEAD apunta a la rama ***pruebas***.

**$ git log –-oneline**

e5a3d40 (HEAD -> pruebas) Cambios en la rama pruebas de repo1.txt

a5599d6 Ficheros nuevos en la rama

fc11dc5 (master) Creo repo3 paso3

73daf15 Nueva linea en repo1 y 2 paso2

caa0b10 Creo repo1 y repo2 paso1

Para fusionar (merge) dentro de la rama ***master***, primero nos movemos a la rama objetivo, en este caso master, y lanzamos **git merge <rama origen> <rama destino del merge>** para fusionar la nueva rama sobre la rama ***master***.

**$ git checkout master**

**$ git merge pruebas master**

Ahora ***master*** y ***pruebas*** tienen el mismo apuntador HEAD, pero el repositorio remoto está dos commits por detrás.

**$ git log –-oneline**

e5a3d40 (HEAD -> master, pruebas) Cambios en la rama pruebas de repo1.txt

a5599d6 Ficheros nuevos en la rama

fc11dc5 (origin/master, origin/HEAD) Creo repo3 paso3

73daf15 Nueva linea en repo1 y 2 paso2

caa0b10 Creo repo1 y repo2 paso1

## Comando cherry -pick

El comando **git cherry-pick** es una herramienta poderosa en Git que permite aplicar los cambios introducidos por commits específicos de una rama a otra, sin necesidad de fusionar toda la rama. Esto puede ser extremadamente útil cuando deseas trasladar un conjunto particular de cambios o correcciones sin afectar otros cambios en la rama de origen.

Suponiendo que tenemos dos ramas, por ejemplo: “main” y “feature”. En la rama feature se han realizado varios commits y ahora se desea aplicar uno de esos commits en la rama main. Deberemos asegurar estar en la rama sobre la que queremos aplicar, si no fuera asi aplicaríamos el comando **git checkout nombre-rama**, para cambiar a la rama. Una vez sobre la rama de interés, se utilizará el comando **git cherry-pick <commit-hash> (**donde <commit-hash> es el identificador SHA-1 del commit que deseas aplicar). Una vez aplicado, si queremos confirmar podríamos revisar el log de commits para confirmar que se ha añadido mediante el comando **git log --oneline**.

**$ git log –-oneline**

**abc1234 (HEAD -> feature) Added Feature 3**

**def5678 Added Feature 2**

**ghi9101 Added Feature 1**

**jkl2345 (main) Initial commit**

**$ git checkout main**

**$ git cherry-pick def5678**

**$ git log --oneline**

**mno3456 (HEAD -> main) Added Feature 2**

**jkl2345 Initial commit**

Puedes aplicar múltiples commits de una sola vez proporcionando un rango de commits con el comando **git cherry-pick <start-commit>^..<end-commit>.** Esto aplicará todos los commits desde <start-commit> hasta <end-commit> inclusive.

Si el commit que estás aplicando entra en conflicto con el código de la rama actual, Git pausará el proceso de cherry-pick y te permitirá resolver los conflictos manualmente. Después de resolver los conflictos, debes añadir los archivos modificados **git add <resolved-files>** y continuar el proceso **git cherry-pick –continue**

Si decides abortar el proceso de cherry-pick debido a conflictos irreconciliables, deberás utilizar el comando **git cherry-pick --abort**