# Git

# Git es un sistema de gestión de versiones de código fuente.

En un proyecto, desde un principio hasta ahora, sería deseable movernos por las diferentes etapas del código.

Y también revertir cambios para retornar a un punto deseado.

Git es una herramienta que hace esto posible.

Git es el sistema de control de versiones más utilizado en el mundo actualmente.

Una cantidad asombrosa de proyectos de software dependen de Git para el control de versiones, incluidos proyectos comerciales y de código abierto, especialmente utilizando el servicio de alojamiento de repositorios de git, GitHub, que ahora es propiedad de Microsoft. De ahí la importancia de aprender Git.

## Repositorios

Para Git, cada proyecto es un repositorio. Puede constar de todos los archivos de un programa, de una carpeta con archivos de código, pero también de información de otra índole como una relación de archivos de apuntes o de recetas de cocina.

Cuando desplegamos el repositorio a la nube en Github, en un trabajo en equipo, los repositorios son distribuidos, es decir, cada miembro del equipo dispone en su máquina de los archivos del repositorio.

La idea base es que cada miembro actualice el repositorio y que a su vez esos cambios actualicen los repositorios distribuidos.

# Comandos de Git

Para ejemplificar los comandos en un caso práctico, vamos a ir planteando necesidades en un caso práctico para ver cómo aplicar cada comando para cada caso específico.

**Nota: Todos los comandos se ejecutarán en una terminal y para ejecutarlos, primero habrá que escribir el comando y luego pulsar intro.**

En primer lugar, debemos asegurarnos de que debemos tener git instalado. Para ello, podemos ejecutar el comando: **git --version**

En segundo lugar, debemos tener git configurado para que este sepa quién es el autor de los cambios que se harán en los repositorios. Configuramos el nombre del usuario ejecutando: **git config --global user.name "Fulano".** Después, ejecutamos: **git config --global user.email** [**fulano@gmail.com**](mailto:fulano@gmail.com)

Tanto el nombre de usuario como el email configurado deberían coincidir con las credenciales de nuestra cuenta de Github. El motivo de esto es que, cuando publiquemos nuestros cambios en un repositorio en la nube cuando trabajemos en un equipo, deberemos identificarnos como autores de estos cambios. Así, todos los miembros del equipo sabrán quién ha hecho determinados cambios en el repositorio.

Estos cambios de configuración son una vez para siempre, a no ser que decidamos cambiarlos algún día porque queramos identificarnos de otra manera. Para cambiar esta configuración, simplemente ejecutaríamos los mismos comandos con otro nombre de usuario y otro email. Si queremos revisar estas credenciales, ejecutaríamos el comando: **git config --global -l**

Una vez que ya tenemos git configurado, vamos a iniciar un repositorio local. Más adelante lo desplegaremos a nuestra cuenta de Github.

**Importante: En este ejemplo, vamos a almacenar archivos de texto en una carpeta, centrándonos en los comandos de git más que en el contenido de la carpeta. Esta carpeta será el repositorio. Pero lo normal es que la carpeta tenga un objetivo mucho más determinado en cuanto a uso, por ejemplo, una carpeta donde almacenar todos los archivos de un programa que desarrollemos.**

Creamos una carpeta con el nombre *Textos.* Entramos a una terminal ubicándonos en esta carpeta. **Es muy importante que ejecutemos todos los comandos que impliquen cambios en el repositorio en esta carpeta porque esta será el repositorio.**

Comenzamos el repositorio ejecutando el comando: **git init**

Este comando crea una carpeta *.git* oculta en la carpeta *Textos.* Esta carpeta es la que git utiliza para apuntar los cambios que hacemos en el repositorio. Es **MUY IMPORTANTE** no eliminar esta carpeta ni modificar su contenido. Hacer esto implicaría destruir toda la gestión previa del repositorio.

Vamos a crear dos archivos de texto en la carpeta con los nombres *texto1.txt* y *texto2.txt*

Suponemos que queremos comenzar a gestionar cambios en el repositorio y que estos dos archivos formen parte del mismo. Primero, debemos confirmar que ambos archivos van a formar parte del repositorio.

Podemos ejecutar cualquiera de estos comandos:

Para agregar los archivos uno a uno: **git add texto1.txt** y **git add texto2.txt**

Para agregar los archivos con la misma extensión: **git add \*.txt**

Para agregar todos los archivos al repositorio: **git add .**

Esta acción manda los archivos al *stage* de git. El *stage* es el lugar donde se guardan los cambios antes de actualizarlos definitivamente en el repositorio cuando hagamos un *commit.* Un *commit* confirma un lote de cambios en el repositorio. Estos cambios pueden ser archivos nuevos, modificaciones de archivos existentes, eliminaciones de archivos… El *stage* va memorizando todos los cambios que hacemos previos a cada confirmación.

Para ir viendo todos los cambios que vamos haciendo y el estado del *stage,* ejecutamos: **git status***.* Este comando lo podemos ejecutar cada vez que queramosconsultar el estado del *stage.*

Si queremos quitar un archivo del *stage*, podemos ejecutar el comando **git reset.** Suponemos que ya no queremos que *texto1.txt* forme parte del repositorio. Ejecutamos el comando: **git reset texto1.txt**

Ejecutamos **git status**

Git nos informa de que *texto2.txt* forma parte del *stage* pero *texto1.txt* ya no. Suponemos ahora que volvemos a querer que *texto1.txt* forme parte de repositorio. Ejecutamos el comando: **git add texto1.txt**

Ahora suponemos que queremos confirmar nuestros primeros cambios en elrepositorio. Ejecutamos el comando: **git commit -m “Primer commit”**. Si ahora ejecutamos **git status** veremos que ya no hay nada en el stage porque toda esta información temporal se ha confirmado en el repositorio.

Vamos a eliminar *texto2.txt,* modificar *texto1.txt* y después vamos a crear un archivo nuevo con el nombre *texto3.txt.* Suponemos que nos arrepentimos de estas tres acciones y queremos que todo quede tal y como estaba cuando hemos hecho el último commit. Ejecutamos el comando: **git checkout**

Este comando nos muestra los cambios que podemos deshacer. Para deshacer estos cambios ejecutamos el comando: **git checkout -- .**

Podemos observar que *texto2.txt* se restaura y que en *texto1.txt* se han anulado las modificaciones. Sin embargo, *texto3.txt* no se ha eliminado. ¿Por qué? Porque a este archivo git no le ha dado seguimiento todavía. Vamos a demostrarlo. Ejecutamos el comando: **git add .**

Esto pasa *texto3.txt* al stage3 y git a partir de ahora le dará seguimiento. Ahora confirmamos los cambios ejecutando el comando: **git commit -m “Incluido texto3”**

Ahora, borramos el archivo *texto3.txt* y ejecutamos el comando: **git status**

Git nos dice que *texto3.txt* se ha eliminado. Ahora ejecutamos el comando: **git checkout -- .**

Deberemos ver ahora que *texto3.txt* se ha recuperado porque el estado del repositorio vuelve a ser el que era después del segundo commit.

Modificamos el contenido del archivo *texto2.txt.* Ahora, para ver las diferencias entre la versión actual y la versión anterior, ejecutamos el comando: **git diff texto2.txt**

Para ver todas las diferencias de todos los archivos, ejecutamos el comando: **git diff**

Suponemos que queremos actualizar estos cambios en el repositorio. Ejecutamos el comando: **git add .**

Recordamos que así pasamos los cambios al *stage.* Si ahora ejecutamos **git diff --staged** podemos ver las diferencias entre los archivos, tal como estaban en la confirmación anterior y los cambios trasladados al *stage.*

Confirmamos los cambios ejecutando el comando: **git commit -m “Modificado texto2”**

Para ver el histórico de commits que hemos ejecutado en el repositorio, ejecutamos el comando: **git log**

## Actividad 1

Crear una carpeta con el nombre *Mas textos.*

Iniciar un repositorio en la carpeta. **git init**

Crear dos archivos de texto en la carpeta.

Agregar ambos archivos al *stage.* **git add .**

Chequear el estado del *stage.* **git status**

Confirmar los primeros cambios en el repositorio con el mensaje *Primeros cambios.* **git commit -m "Primeros cambios"**

Eliminar un archivo de texto y recuperarlo mediante un comando de git. **git checkout -- .**

Modificar un archivo y ver las modificaciones mediante un comando de git. **git diff**

Crear un archivo de texto más y agregarlo (darle seguimiento) al *stage.* **git add .**

Confirmar estos cambios en el repositorio con el mensaje *Primeras modificaciones.* **git commit -m "Primeras modificaciones"**

# Comandos de git (continuación)

Seguimos con otro caso práctico. Eliminamos la carpeta textos y la creamos de nuevo. Desde una terminal y ubicándonos en la carpeta, ejecutamos el comando: **git init**

Creamos un archivo *texto1.txt* y ejecutamos el comando: **git add .**

Luego confirmamos cambios ejecutando el comando: **git commit -m “Primer commit”**

Creamos un archivo *texto2.txt* y ejecutamos el comando: **git add .**

Luego confirmamos cambios ejecutando el comando: **git commit -m “Segundo commit”**

Creamos un archivo *texto3.txt* y ejecutamos el comando: **git add .**

Luego confirmamos cambios ejecutando el comando: **git commit -m “Tercer commit”**

Suponemos que deseamos anular el último commit. Tendríamos tres alternativas:

Primera alternativa: El llamado *reset suave (soft).* Si ejecutamos un *reset soft*, anula los cambios en el repositorio, estos permanecen en la carpeta y también en el *stage.* En nuestro ejemplo, un *reset soft* quitaría la confirmación *Tercer commit,* pero los cambios de *texto3.txt* quedarían en el archivo y en el *stage.* Lo comprobamos. Ejecutamos el comando de *reset suave:* **git reset --soft HEAD~1**

git reset --soft ejecuta el reset suave y HEAD^ lo ejecuta en el último commit. (Nota: HEAD^^ desharía los dos últimos commits y así sucesivamente)

Volvemos a confirmar los cambios ejecutando el comando: **git commit -m “Tercer commit”**

Segunda alternativa: El llamado *reset medio (mixed).* Si ejecutamos un *reset mixed,* anulamos los cambios en el repositorio, estos permanecen la carpeta pero se van del stage. En nuestro ejemplo, un *reset mixed* quitaría la confirmación *Tercer commit* y los cambios de *texto3.txt* quedarían en el archivo pero en el *stage.* Lo comprobamos. Ejecutamos el comando de *reset medio:* **git reset --mixed HEAD~**

Ejecutamos el comando: **git status**

La información que se nos muestra nos corrobora que el archivo *texto3.txt* no está en el *stage.*

Volvemos a confirmar los cambios ejecutando el comando: **git add .**

Después ejecutamos el comando: **git commit -m “Tercer commit”**

Tercera alternativa: El llamado *reset duro (hard).* Si ejecutamos un *reset hard,* anulamos los cambios en el repositorio y se deshacen los cambios en el *stage* e incluso en la carpeta . En nuestro ejemplo, un *reset hard* quitaría la confirmación *Tercer commit* destruiría el archivo *texto3.txt.* Lo comprobamos. Ejecutamos el comando de *reset duro:* **git reset --hard HEAD~**

Otra manera de deshacer commits es utilizando su identificador. Ejecutamos el comando: **git log --oneline**

Observamos que cada commit tiene un identificador que consta de números y letras. Podríamos haber ejecutado cualquiera de los comandos anteriores apuntando al identificador del commit a deshacer. Por ejemplo, si hubiéramos ejecutado el comando: **git reset --soft 2900cf2** hubiéramos deshecho ese commit.

Hay que tener mucho cuidado al deshacer commits, sobre todo los de grandes implicaciones. Vamos a ejecutar dos vedes el comando: **git reset --hard HEAD~**

A menudo será mejor hacer las modificaciones pertinentes y hacer nuevos commits que deshacer commits ya hechos. No obstante, git nos da soluciones para todo. Vamos a suponer que deseamos devolver a nuestro repositorio a la situación del tercer commit. Vamos a ejecutar el comando: **git reflog**

Este comando devuelve un histórico de todos los movimentos, mientras que git log nos muestra los movimientos del repositorio de los commits que no hemos deshecho.

Nos fijamos en el identificador del “Tercer commit” y ejecutamos el comando **git reset --hard identificador.**

## Actividad 2

Crear una carpeta con el nombre *Otros textos.*

Iniciar un repositorio en la carpeta. **git init**

Crear un archivo de texto en la carpeta.

Agregar el archivo al *stage.* **git add .**

Confirmar los cambios con el nombre “Primer texto”*.* **git commit -m “Primer texto”**

Crear otro archivo de texto en la carpeta.

Agregar el archivo al *stage.* **git add .**

Confirmar los cambios con el nombre “Segundo texto”*.* **git commit -m “Segundo texto”**

Deshacer el último cambio con un *reset soft*. **git reset --soft HEAD~1**

Volver a confirmar los cambios con el nombre “Segundo texto”*.* **git commit -m “Segundo texto”**

Deshacer el último cambio con un *reset hard*. **git reset --hard HEAD~1**

En el repositorio ya solo deberíamos de ver en primer texto.

# Comandos de Git (Continuación)

Las ramas son una división del estado del código. Esto permite crear nuevos caminos a favor de la evolución de las aplicaciones. Por ejemplo, imagina que quieres añadir una nueva función o tal vez arreglar un error. Puedes generar una nueva rama para comenzar con estos cambios. Esta rama tendrá el mismo contenido que el repositorio donde nace, pero a partir de su creación tiene su propia vida y todos los commits que hagamos en esa rama afectarán a la rama en cuestión, pero no a la rama principal. Cuando una rama ya ha cumplido su propósito, convergerá con el repositorio principal y lo actualizará.

En git las ramas se llaman *Branch*. La rama principal se llama *master* y al resto de ramas les tendremos que dar nombres nosotros.

Vamos a poner un ejemplo. Vamos a crear una carpeta con el nombre *libros.* Dentro creamos un archivo de texto con el nombre *autores.txt.* Ahora vamos a crear un repositorio en esta carpeta y vamos a confirmar los primeros cambios. Ejecutamos los comandos:

**git init**

**git add .**

**git commit -m “Primer commit”**

Ahora vamos a suponer que, por un lado, queremos ir creando archivos de cuentos y por otro lado archivos de novelas. Como van a llevar un desarrollo independiente, creamos las ramas *cuentos* y *novelas.*

Ejecutamos el comando: **git branch cuentos**

Ejecutamos el comando: **git branch novelas**

Ejecutamos el comando: **git status**

Para listar las ramas creadas, ejecutamos el comando: **git branch**

Ahora suponemos que queremos ir escribiendo cuentos. Nos cambiamos a la rama *cuentos* ejecutando el comando: **git checkout cuentos**

Como ya estamos en la rama de cuentos, creamos los archivos *cuento1.txt* y *cuento2.txt.* Confirmamos los cambios:

Ejecutamos el comando: **git add .**

Ejecutamos el comando: **git commit -m “Primeros cuentos”**

Ahora, en la rama cuentos tenemos los dos archivos de cuentos, pero en la rama principal *(master)* y en la rama *novelas* no están. Lo comprobamos:

Nos cambiamos a la rama master. Ejecutamos el comando: **git checkout master**

El contenido de la carpeta master cambia y ya no aparecen los cuentos porque estos están en la rama *cuentos*.

Suponemos que pasa el tiempo y ya hemos terminado los cuentos. Ahora queremos terminar con la rama *cuentos* y que estos pasen a la rama *master,* que es donde estarán todos los libros. Primero nos vamos a la rama *master*. Será desde aquí donde traeremos los cuentos. Ejecutamos el comando: **git checkout master**

Ahora ejecutamos el comando: **git merge cuentos**

Este comando hace converger el contenido de las dos ramas, trayendo los cuentos a la rama principal.

La rama *cuentos* podría seguir evolucionando y podríamos seguir incorporando cuentos a la rama principal. En nuestro caso, como la rama *cuentos* ya ha cumplido su función, decidimos eliminarla ejecutando el comando: **git branch -d cuentos**

Ahora vamos a suponer que nuestros libros ya han evolucionado lo suficiente como para marcar un hito. Ese hito sería el número de versión del repositorio. Git Ese número lo decidimos nosotros y en git se llama *tag*.

Suponemos que queremos ponerle como tag *v1.0.0.* Ejecutamos el comando: **git tag v1.0.0**

Para ver el versionado del repositorio, ejecutamos el comando: **git log**

Para ver todos los versionados, ejecutamos el comando: **git tag**

El último commit, que en realidad ha sido el *merge* de *cuentos*, registra el tag que hemos proporcionado.

Ahora vamos a seguir con las novelas. Para cambiarnos a esa rama, ejecutamos el comando: **git checkout novelas**

Creamos el archivo *novela1.txt*. Ahora suponemos que otra persona ha cambiado la rama principal creando un nuevo archivo *editoriales.txt.* Lo simulamos nosotros. Ejecutamos el comando: **git checkout master**

Creamos el archivo *editoriales.txt*. Ahora confirmamos los cambios ejecutando el comando: **git add .**

Terminamos la confirmación con el comando: **git commit -m “Agregadas editoriales”**

Ahora suponemos que ese archivo nos hace falta para seguir con la evolución de la rama *novelas.* ¿Cómo podemos incluir editoriales (que está en la rama master) en la rama *novelas*? Primero nos vamos a la rama *novelas* ejecutando el comando: **git checkout novelas**

Y una vez en la rama *novelas,* ejecutamos el siguiente comando: **git rebase master**

El comando *rebase* permite insertar contenido de la rama *master* a otra. Por último, planteamos otro caso. Suponemos que alguien cambia las editoriales de la rama *master*. Ahora la rama *master* tiene diferentes editoriales que la rama *novelas*. ¿Cómo sincronizamos los dos archivos? Para simular el caso, nos vamos a la rama *master* ejecutando el comando: **git checkout master**

Modificamos el archivo *editoriales.txt.* Para confirmar los cambios en el repositorio, ejecutamos el comando: **git commit -am "modificadas editoriales"**

*-am* permite de una vez agregar los cambios al stage y confirmarlos con un mensaje. Ahora listamos todos los commits ejecutando el comando: **git log --oneline**

Nos fijamos en el identificador del último commit, confirmación que nos ha modificado el archivo de editoriales, y actualizamos este archivo en la rama *novelas.* Primero nos vamos a la rama novelas ejecutando el comando: **git checkout novelas**

Luego ejecutamos el comando: **git cherry-pick identificador**

El comando *cherry-pick* permite actualizar información de una rama a otra.

## Actividad 3

Crear una carpeta con el nombre *Apuntes.*

Iniciar un repositorio en la carpeta. **git init**

Crear un archivo de texto en la carpeta con el nombre *AutorApuntes.txt*.

Agregar el archivo al *stage.* **git add .**

Confirmar los cambios con el nombre “Agregado autor”*.* **git commit -m “Agregado autor”**

Crear una rama con el nombre *Tema1.* **git branch Tema1**

Cambiarse a la rama *Tema1* y agregar dos archivos de texto dentro. **git checkout Tema1**

Agregar estos archivos al *stage* de la rama y confirmar los cambios con el nombre “Agregados apuntes tema 1”*.* **git add .**

**git commit -m “Agregados apuntes tema1”**

Fusionar los cambios de la rama *Tema1* con la rama principal *(master).* **git checkout master**

**git merge Tema1**

Eliminar la rama *Tema1.* **git branch -d Tema1**

# Github

A continuación, vamos a ver cómo alojar un repositorio en la nube en nuestra cuenta de Github. Suponemos que queremos crear un repositorio de textos. Lo primero que debemos hacer es crear el repositorio en nuestra cuenta de Github. Vamos a la web de Github, nos identificamos y vamos a la opción New y cumplimentamos los datos que se nos piden. En el nombre del repositorio escribimos *Textos,* el repositorio lo ponemos como público *(Public)* y activamos la opción *Add a README file*. Esta opción creará en el repositorio un archivo que será la presentación del repositorio. Esto permite que cuando alguien entre a la URL de nuestro repositorio verá un archivo con la información que pongamos. Pulsamos el botón *Create repository* para terminar el proceso de creación. El siguiente paso es sincronizar este repositorio en la nube con una carpeta de nuestro ordenado. Creamos en nuestro equipo una carpeta *Repositorios,* entramos a una terminal y nos ubicamos en esta carpeta.

Ejecutamos el comando: **git clone https://github.com/usuario/repositorio**

Con este comando ya clonamos el repositorio en nuestro equipo y a partir de ahora, podremos sincronizar nuestro repositorio en la nube con el de nuestro ordenador. Vamos a probar esto. Creamos un archivo *texto1.txt* y lo agregamos al *stage* ejecutando el comando: **git add .**

Ahora efectuamos un commit ejecutando el comando**: git commit -m "agregado texto1"**

Subimos al repositorio los cambios ejecutando el comando: **git push**

Para actualizar en nuestro repositorio local el repositorio en la nube ejecutaríamos el comando: **git pull**

El trabajo con ramas sería igual al visto antes. Trabajaríamos con las ramas que fueran y para actualizarlas en el repositorio en la nube ejecutaríamos el comando: **git push origin nombrerama.** La primera vez que queremos actualizar la rama desde el repositorio local a GitHub debemos ejecutar **git push --set-upstream origin nombrerama** para establecer el vínculo entre la rama local y la rama remota.

Para actualizar en el repositorio remoto los tags que vamos haciendo en local, debemos ejecutar **git push origin nombreetiqueta.**

Para eliminar una rama de nuestro repositorio en la nube ejecutaríamos el comando: **git push origin --delete cuentos.**

Y para actualizar en nuestro repositorio local una rama desde el repositorio remoto ejecutaríamos el comando: **git pull origin nombrerama.**

## Actividad 4

Crear un repositorio en Github

Clonarlo en una carpeta de nuestro repositorio local. **git clone https://github.com/usuario/repositorio**

Hacer cambios en nuestro repositorio local creando un archivo, pasándolo al stage y haciendo un commit.

**git add .**

**git commit -m "agregado texto1"**

Subir los cambios al repositorio remoto. **git push**

# Git Flow

Git Flow es una herramienta que ayuda a estandarizar un Work Flow cuando se trabaja en equipo. El objetivo es mantener un estándar de flujo de trabajo que se ha establecido en el equipo como el que se debe seguir. Git Flow logra esto ofreciendo ciertos atajos al desarrollador que hacen que la gestión del proyecto sea más cómoda y ajustada ese flujo.

## Ramas de Git Flow

1 rama *Master* donde se alojará el código estable del producto, por ejemplo, versiones en producción consolidadas. Se aconseja utilizar tags en sus commits.

1 rama *Develop* donde se alojará el código en desarrollo. Aquí se encontrará la última versión del código, con las últimas características, pero que no ha sido testado y por ello aún no está listo de ser mezclado con la rama master. Desde esta rama haremos el *merge* con las ramas *feature.*

N ramas *Features* donde se estará trabajando sobre características concretas para tenerlas aisladas de otras ramas y poder cambiar de una a otra sin problema. Una rama *feature* siempre será una bifurcación de *develop* yse crearán a partir de la última *develop.* Cuando se completa una *feature,* esta se fusiona con *develop* y se elimina de forma automática.

Ramas *Release.* Cuando la rama *Develop* tenga las suficientes *Features* como para considerar un nuevo lanzamiento, se creará una rama *Release* con los cambios. Cuando creamos una *Release*, se entiende que se empieza un nuevo ciclo de lanzamiento. No se deberían integrar nuevas *Features* a partir de ese punto, a no ser que se detecte algún error o se quiera añadir documentación. Cada *Release* se fusionará con *Master* y se etiqueta el número de versión. A la vez, también se fusionará con *Develop* y después se eliminará de forma automática.

Ramas *HotFix.* Las ramas *HotFix* están pensadas para reparar errores. Son muy similares a las ramas *Feature*, pero se derivan de *Main*. *HotFix* es el único tipo de rama de función que se bifurca directamente de *Main*. Cuando se termine un *HotFix*, este se debe fusionar con *Main* y *Develop*. *Main* debería etiquetar un número nuevo de versión.

Ramas *Bugfix*: Si se detecta un bug que no ha llegado a producción, realizaremos un *bugfix*. Se puede elegir una rama, pero lo habitual es hacerlo en *Develop*.

Ramas *Support*: Es un tipo especial de rama que se puede crear a partir de *Master*. Se usa si se tienen clientes que no desean actualizar a las nuevas versiones y que necesitan soporte en versiones anteriores. Sería como una versión de *Master* con sus propias ramas.

Método resumido

Se crea una rama **Develop** desde la rama **Master**.

Se crean ramas **Feature** a partir de la rama Develop.

Cuando una Feature esté lista, se fusiona con Develop.

Se crea una rama **Release** desde la rama Develop.

Cuando una Release se completa, se fusiona con Develop y con Master.

Si se detectan errores en Master, se crean ramas **HotFix** desde Master.

Una vez se haya subsanado el error, una Hotfix se fusiona con Develop y Master.

## Ejemplo de uso

Vamos a hacer un ejemplo. Comenzamos un repositorio local en una carpeta como lo hacemos siempre: Ejecutamos el comando: **git init.**

Después, lo convertimos a un repositorio que se va a construir con Git Flow. Ejecutamos el comando: **git flow init**

Pulsamos intro para aceptar todas las opciones por defecto y comprobamos el resultado ejecutando el comando **git branch**

Debemos ver cómo se crean las ramas *develop* y *master* y que se nos posiciona automáticamente en *develop*. Ahora suponemos que queremos crear una rama *feature* para redactar textos en el repositorio. Ejecutamos el comando: **git flow feature start textos**

Esta acción nos crea la rama y nos sitúa en ella.

Ahora creamos dos archivos de texto en la rama y para hacer el commit utilizamos los comandos típicos: **git add .**  y **git commit -m “Agregados textos”**

Ahora suponemos que ya hemos terminado el trabajo con la rama *textos.* Para fusionarla con *develop* ejecutamos el comando: **git flow feature finish textos**

Esta acción pasa los textos a la rama *master* y elimina la rama *textos*. Ahora suponemos que queremos marcar un hito en el versionado del repositorio. Ejecutamos el comando: **git flow release start 1.0.0**

Agregamos un archivo para documentar la versión llamado *info.txt.* Para hacer el commit utilizamos los comandos típicos: **git add .**  y **git commit -m “Agregado archivo info”**

Ahora suponemos que ya queremos marcar la versión 1.0.0. Ejecutamos el comando: **git flow release finish -m "1.0.0" 1.0.0**

Esta acción elimina la rama *release 1.0.0* y pasa los cambios a *Master* y *Develop.* Ahora suponemos que hemos detectado un fallo de ortografía en un texto. Para subsanarlo, creamos una rama de tipo *hotfix* ejecutando el comando: **git flow hotfix start errortexto**

Esta acción nos crea la rama *errortexto* y nos posiciona en ella. Cambiamos algo de un archivo de texto y ejecutamos el commit con el comando: **git commit -am "Solucionado error texto"**

Ahora, una vez subsanado el error, decidimos terminar con la rama *errortexto.* Ejecutamos el comando: **git flow hotfix finish -T "1.0.1" -m "1.0.1" errortexto**

Este comando pasa los cambios a las ramas *master* y *develop*. También elimina la rama *errortexto.* Podemos comprobarlo ejecutando el comando: **git branch**

También podemos consultar el histórico del repositorio ejecutando el comando: **git log --oneline**