# Unidad 3 - Control de versiones con Git

El elemento más importante de un proyecto de software es, evidentemente, su código. Necesitamos evolucionarlo, desarrollarlo, de forma cuidadosa, cuidando los cambios que se introducen en el mismo. Esto es especialmente importante cuando hay dos, tres o más personas programando sobre el mismo código fuente.

## ¿Qué es el control de versiones?

El control de versiones (*version control*, *source control*) es la práctica de registrar y gestionar cambios en el código. Un sistema de control de versiones (*source control management*, SCM) provee de un historial vivo del desarrollo del código y ayuda a resolver conflictos cuando se trata de fusionar (mergear, merge) contribuciones de diferentes fuentes (generalmente personas que quieren añadir cambios para el desarrollo de la aplicación). Para la mayoría de equipos de desarrollo, el códifgo fuente es un repositorio de conocimiento de valor incalculable que ayuda a resolver problemas que los desarrolladores han coleccionado y refinado con gran cuidado. El control de versiones protege el códio fuente de catástrofes y degradación casual de los errores humanos y consecuencias involuntarias.

Sin control de versiones, puedes tener la tentación de crear múltiples copias de tu código en tu equipo. Esta práctica es muy peligrosa: es terriblemente fácil cmabiar o eliminar un fichero en la copia equivocada del código, potenciando la pérdida de trabajo. Los sistemas de control de versiones ayudan a resolver este problema gestionando todas las versiones de tu código a la vez, pero presentando solo una de estas al tiempo.

## ¿Por qué es importante?

Para resolver esta pregunta, necesitamos conocer cuales son los valores del desarrollo de software:

- Reusabilidad, porque ¿por qué hacer dos veces la misma cosa? La reutilización del código es una práctica común y hace más fácil el desarrollo del código.
- Seguimiento, puesto que toda la actividad realizada debe ser registrada y gestionada para realizar informes cuando sean necesarios. A menudo son cuestiones legales las que obligan a esto, además de poder identificar problemas y quien los ha introducido, y así corregirlos de forma óptima.
- Gestión, ¿pueden los líderes de equipo definir y reforzar flujos de trabajo, revisar reglas, crear puertas de calidad y reforzar la calidad del código a través del ciclo de vida de la aplicación?
- Eficiencia, ¿se están usando los recursos adecuados para el trabajo y así minimizar los tiempos y el esfuerzo invertidos?
- Colaboración, que es lo que permite a los equipos mejorar, pues todos aprenden de todos y se puede desarrollar sobre las fortalezas de todo el equipo.
- Aprendizaje, pues las organizaciones se beneficia de la inversión que realizan en el crecimiento y aprendizaje de sus empleados. No es solo importante para las personas que se

incorporan de nuevas en un equipo, puesto que es una aprendizaje que nunca acaba para todas las personas en un equipo. Contribuyen así no solo al proyecto en el que están embarcados, si no también a toda la industria en su conjunto.

La versión de control facilita muchos si no todos de estos valores.

El control de versiones trata sobre el registro de cada cambio que se realiza sobre el software: registrar y gestionar el quién, cuando y qué. Es el primer paso para aseguriar la calidad desde el código fuente. Toda esta creación de valor que empieza desde este punto no es solo valiosa para el equipo de desarrollo, si no para el cliente final.

También permite "retroceder en el tiempo" y revertir a versiones anteriores si es necesario, ver quién ha realizado esos cambios, cuando y cómo han sido creados. Y sin duda permite la experimentación sin riesgo, además de permitir la colaboración.

**Sin el control de versiones, la colaboración sobre el código fuente sería una experiencia dolorosa e incómoda.**

Será a su vez una práctica diaria, una herramienta con la que tratarás día a día y que abrirá todas estas puertas que mencionamos.

Trabajar con control de versiones, ya sea para proyectos profesionales o personales, permite:

- Crear flujos de trabajo
- Trabajar con versiones
- Colaboración
- Mantener un historial de los cambios
- Automatización de tareas

En muchas ocasiones, como podemos ver, se trata de buenas prácticas para mejorar la comunicación con el equipo.

## Conceptos clave

Para introducirnos en el control de versiones, necesitaremos asentar algunos conceptos clave que mencionaremos a menudo, y que se convertirán en vocabulario habitual en el equipo que implementa cultura DevOps.

### Repositorios

Un repositorio es el lugar donde se almacena el código fuente de una aplicación. Es un directorio que está bajo el control de un gestor de control de versiones. Todos los cambios que son realizados en ese directorio (nuevos directorios, ficheros que se cambian de localización, cambios en el contenido de ficheros, ficheros eliminados, etc.) son detectados para que podamos operar con ellos.

Un repositorio puede estar en una carpeta o directorio de nuestro equipo, o estar en un servidor remoto. Si un repositorio está en una carpeta, todo lo que esté en esa carpeta será controlado por el control de versiones.

**Tipos de repositorios**

- Centralizado

Existe una única copia en la que cada developer colabora. Los cambios se registran en esa única copia y son visibles por todos los miembros del equipo.

- Distribuido

Es la tendencia actual. Cada developer en el equipo se crea una copia (clona) el repositorio original y trabaja sobre ella, pulleando o pusheando los cambios que procedan. Puede consumir mucho espacio, pero entre lo barato de los espacios en disco y los sistemas de compresión modernos, apenas se nota la diferencia.

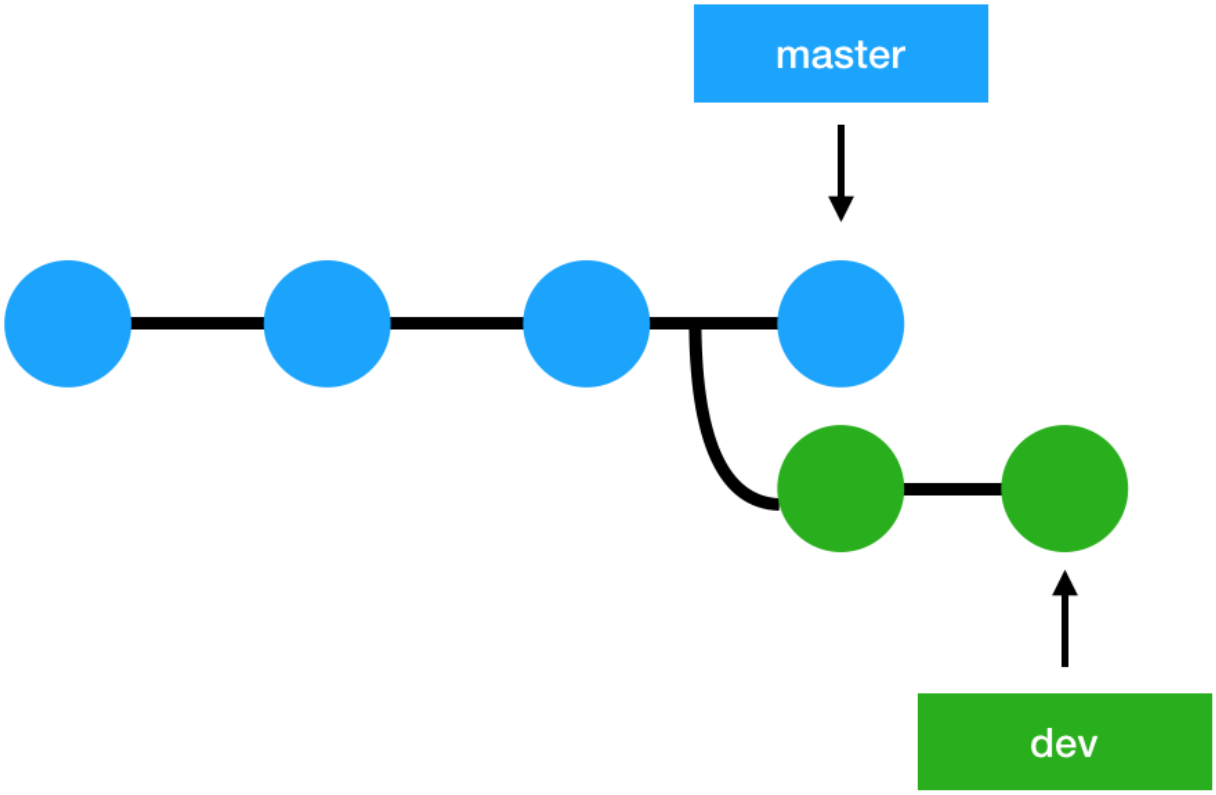
## *Commit*

Los *commits* son la unidad bajo la que se almacenan cambios en un repositorio. Consta de varias partes, a saber:

- ID, identificador unívoco de un *commit*.
- *Message*, donde en resumen se describen los cambios realizados.
- Descripción, opcional, donde se detallan los cambios y se añade otra información útil.
- ID de su *commit* predecesor, de forma que se localizan de forma secuencial, creando la línea temporal.

## *Branch*

Las ramas son las versiones que se gestionan en un repositorio. Tienen nombres para su fácil identificación. Están compuestas de *commits*, y son en esencia líneas temporales que recogen cambios a lo largo del tiempo. Es por esto que decimos que podemos "viajar en el tiempo" a través de nuestro control de versiones, pues podemos viajar entre *commits* para revertirlos, crear nuevas ramas a través de un _commit específico...




Creamos ramas a partir de otras, partiendo del punt

Existen ciertas convenciones con los nombres de las ramas, pero todo está sujeto a las decisiones del equipo de desarrollo. Por lo general, se siguen las siguientes:

- `master`, la rama principal, de producción. Es la menos susceptible a cambios y debe estar siempre en buen estado.
- `develop`, la rama de desarrollo. Esta es más susceptible a cambios, es donde deben ir a parar las contribuciones del equipo de desarrollo. A cada cambio que se vaya a implementar, se abre una nueva rama a partir de desarrollo, se realizan los cambios, que serán implementados en la rama `develop`. Si otros cambios han de realizarse, se sigue esta misma pauta, manteniendo a `master` con buena salud y sin cambios "experimentales".

### *Clone*

Un repositorio puede estar almacenado en la nube o en un servidor. Para empezar a trabajar con él, debemos tener una copia idéntica para empezar a modificarla. Con *clones* podremos tener copias locales idénticas al repositorio para poder operar con él e ir añadiendo cambios.

El clonado se hace a través de comandos, es un proceso de descarga del repositorio.

### *Staging*

Los cambios son detectados en nuestro repositorio local, a medida que modificamos el código fuente estos ficheros se listan para que hagamos *commits* de ellos.

¿Pero qué ocurre si no todos estos ficheros irán en el mismo *commit*? ¿Cómo excluyo o filtro los que quiero utilizar para un *commit*? Aquí se introduce el *staging*, un método con el que especificamos qué ficheros se van a incluir en el *commit*. Todos los ficheros, al modificarse, están *unstaged*. Modificamos su estado a *staged* para tenerlo en cuenta en el próximo cambio.

### *Pull request*

Cuando tenemos los cambios en ramas específicas, llegará un momento en que necesitaremos incluirlos en la rama *develop* o en cualquier otra que necesitemos. Podemos fusionarlos de forma directa si tenemos permisos, pero la forma más correcta es hacerlo a través de *pull requests*.

Un *pull request* es una solicitud para incorporar los cambios de una rama a otra. En este proceso, se inicia una conversación en el equipo. Todo el código que se quiere introducir se puede visualizar, para comentarlo, proponer cambios o revisar si cumple unos requisitos de calidad mínimos.

Es una excelente forma de aprender a través de la revisión de código y compartir con todo el equipo los cambios que se pueden introducir, evitando introducir errores... y haciendo partícipe del código que se produce.

### *Merges*

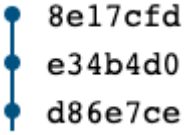
Hemos conocido los *pull requests* y lo sanos que resultan para nuestra calidad en el código, ¿pero qué ocurre cuando estos cambios son validados y aptos para introducirse en la rama objetivo?

Se produce una fusión de los cambios, los cambios de la rama fuente se fusionan con la rama objetivo en un *merge*. Los *merges* se hacen desde la rama objetivo, que cogen los cambios propuestos y los incorporan. En ocasiones, cuando hay conflictos entre cambios, toca resolverlos a mano: ¿qué cambio debe quedarse y cual ser descartado? Es ahí donde la comunicación en el equipo resulta vital.

## El flujo de sistema de control de versiones

Imaginemos que tienes un repositorio, tu código fuente en un directorio controlado por un sistema de control de versiones.

Los cambios que produces en el código funcionan como una línea temporal, donde cada *commit* es un momento concreto que puedes identificar. *Commit* a *commit* vas creando tu línea de tiempo, que puedes representar como una línea que se detiene en diferentes puntos, tus *commits*. Esta es tu rama principal, a la que solemos llamar master.



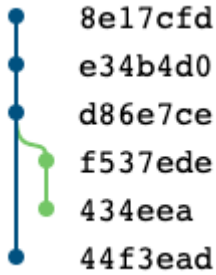

Tu código funciona bien, es estable. Pero quieres añadir nuevos cambios sin afectar a los que ya existen, a tu rama `master`, dibujada en azul. Así que vamos a trabajar con una versión alternativa

para no afectar a estos cambios.


Creamos una rama nueva, llamada `cambios`, que vemos en verde. En esta rama vamos a experimentar, programar aquello que necesitemos. Cuando hemos acabado, creamos un nuevo *commit*.


En el gráfico, vemos cómo estos cambios son interpretados por el sistema de control de versiones.

Podemos seguir desarrollando sobre la rama `cambios`, de modo que seguiría su línea.




Pero lo que también es posible es volver a la rama `master`, dejando los cambios de la rama `cambios` tal y como están, para seguir desarrollando en la rama `master`.


Esto es lo más importante: **los cambios que hemos hecho en una rama no afectan a la otra**.

En el siguiente vídeo veremos cómo podemos cambiar de ramas, y comprobaremos también que, **los cambios que hemos hecho mientras estábamos en una rama, no serán visibles cuando cambiemos a otra diferente**.

**Las ramas son las diferentes versiones con las que trabajamos**, y no se interceptan unas con otras.

## ¿Qué es Git?

Creado en 2005 por Linus Torvalds (creador del kernel de GNU/Linux, el sistema operativo *open-source*). Es el software que instalamos en nuestros equipos y a través del cual realizamos el control de versiones.

Con Git en nuestro equipo podemos:

- Crear repositorios
- Gestionar *branches*
- Crear *commits*

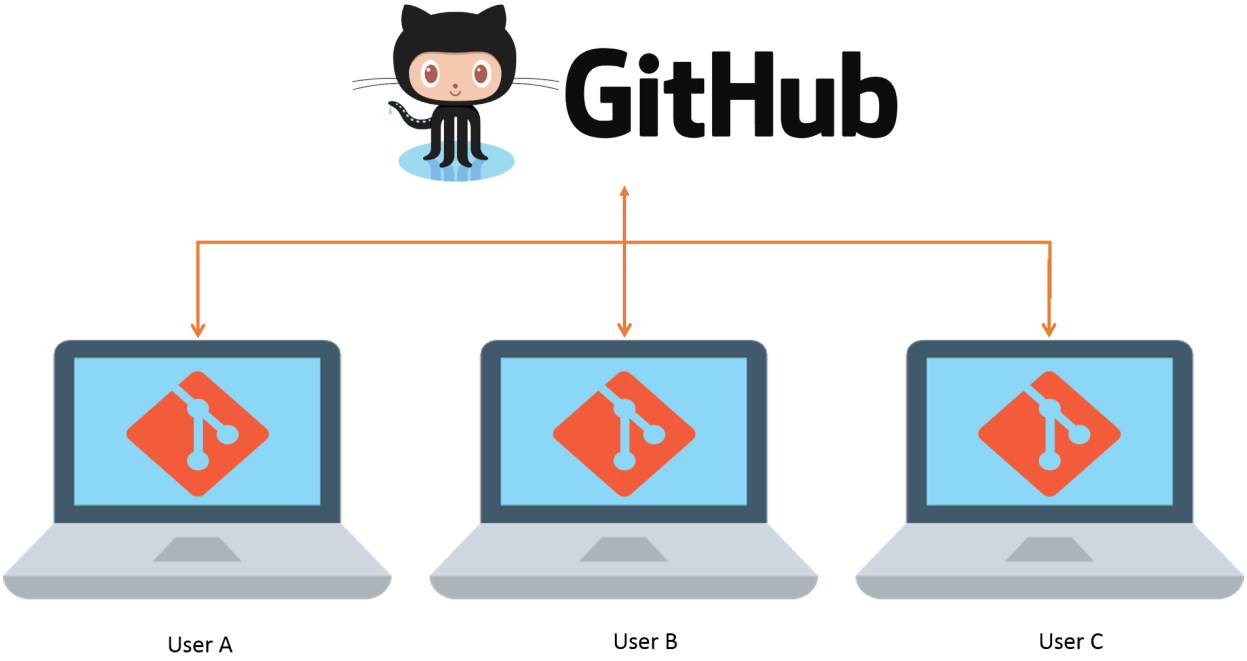
Es la herramienta con la que, a través de la terminal o consola, ejecutaremos los comandos para realizar estas acciones y más.

### Git y GitHub

Ya sea por el nombre o porque nuestro primer contacto con el control de versiones en la red nos dirige a GitHub, es un error común confundir GitHub con Git. Nada más lejos de la realidad, pues son muy diferentes y aunque comparten parte de su definición, tienen objetivos muy diferentes.

Git es el sistema por el cual almacenamos, controlamos y gestionamos los cambios sobre un repositorio localizado en nuestro equipo de forma local o en un servidor dentro de nuestra empresa. Puede estar en cualquier parte, aislado de forma total de la red o no.

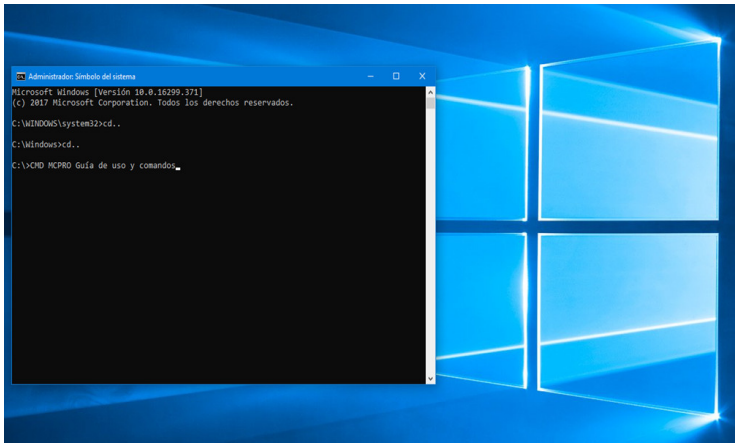
GitHub, en cambio, es una plataforma web para almacenar nuestros repositorios en la nube. Ahora es propiedad de Microsoft, y en los últimos años ha ido adoptando mecánicas que lo acercan también al concepto de red social. Muchas personas que desarrollan *software* se comunican a través de la plataforma, comparten intereses y también conocimiento, creando una comunidad cada vez más grande y que evoluciona a pasos agigantados.

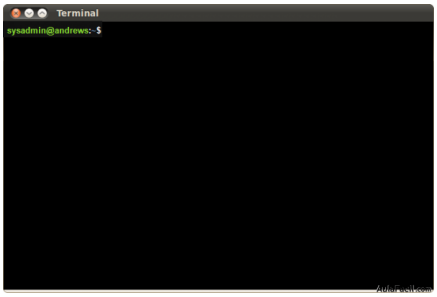



## Navegación en consola

¿Cómo utilizamos Git? A través de una consola o terminal, que existe en nuestros sistemas operativos como una herramienta más. Puede inspirar cierto respeto, pero al final es una utilidad más en nuestros equipos en la que podemos apoyarnos para realizar ciertas tareas.

---




Es aquí donde vamos a trabajar de forma inicial con nuestros repositorios, pero debemos aprender a cómo utilizar esta herramienta.

Lo más importante es saber que la terminal te localiza en una carpeta o directorio concreto. La primera línea te indica la ruta o carpeta en la que te encuentras. Si no te queda claro, puedes ejecutar:

- En Windows: `chdir`
- En Linux: `pwd`

Tras escribir el comando, pulsa ENTER.

Y te dirá la ruta de la carpeta en la que te encuentras.

¿Qué hay dentro de esa carpeta? Puedes listar todas las carpetas y archivos que se encuentren en la carpeta donde te encuentras con los siguientes comando:

- En Windows: `dir`
- En Linux: `ls`

Y verás una lista de todos esos ficheros y carpetas que haya en la carpeta. Después de ver las carpetas, si queremos movernos a una en concreto, podemos ejecutar:

- En Windows: `cd nombreCarpeta`
- En Linux: `cd nombreCarpeta`

Y si volvemos a ejecutar el comando que nos indica donde estamos, veremos cómo ha cambiado nuestra localización. Si ejecutamos el comando que lista carpetas y ficheros, también mostrará el contenido de nuestra localización actual.

Si queremos retroceder a la carpeta anterior, podemos ejecutar:

- En Windows: `cd ..`
- En Linux: `cd ..`

Veremos que, a medida que vamos cambiando de localización, la línea en la cual escribimos los comandos va cambiando, mostrándonos en cada momento dónde nos encontramos.

Estos comandos no son parte de control de versiones, si no de gestión de sistemas operativos. Sin embargo, son útiles de conocer por si necesitamos navegar entre diferentes carpetas que contengan diferentes repositorios.

## Buenas prácticas

- Cambios pequeños
- No commits de ficheros personales o información sensible
- Actualiza con frecuencia, siempre antes de pushear para evitar conflictos
- Asegúrate de que compila y pasa los tests
- Buenos mensajes de commits
- Linkea a work ítems
- Sigue las convenciones del equipo.

## Flujos de trabajo recomendados

Estas son algunas recomendaciones para el uso de Git:

- Flujo de ramas con *features*. Crea una rama para cada cambio a introducir, bajo el nombe de `feature/cambio-nuevo`. Se crea un espacio aislado de trabajo, que no afecta a la rama principal, que debería estar limpia de errores y siempre con calidad de producción. También

favorece a flujos ágiles. Puedes implementar políticas en las que los cambios asociados a una tarea específica deben ir a una sola rama.
- Desarrollo distribuido. Copia local completa del repositorio para cada developer. Se eliminan los bloqueos si algo va mal en el repositorio, ya que con las copias locales todo puede funcionar bien en cada copia individual. Si se corrompe tu repositorio, puedes crear una copia fresca con los últimos cambios.
- *Pull requests*. Solicitud de incorporación de los cambios a una rama. Permite generación de discusiones, cambios que son mejorables o discutibles. Se pueden proponer cambios sin temor a destruir el proyecto (especialmente útiles para juniors)
- Comunidad. Es tan popular y está tan integrado en el desarrollo de software que es improbable que nuevos integrantes desconozcan cómo funciona.
- Ciclo de *releases* rápido. Cambios tan pequeños y frecuentes permiten despliegues más veloces, en lugar de despliegues monolíticos.
- Integración continua (CI) y compilaciones automáticas.

## Anti-patrones en Git

No existe una única forma válida de hacer las cosas en Git, sin embargo existen muchas formas de hacer las cosas mal y caer en las malas prácticas.

- Crear muchas ramas simultáneas y mergear (o intentarlo) al final. Solución: mergea poco a poco, actualízate con la principal.
- Falta de comunicación, de confianza en el equipo: crear una rama independiente por falta de confianza. Solución: comunícate, enseña mejores prácticas, utiliza *pull requests*.
- Cambios que deberían ir en conjunto separados en múltiples ramas. Solución: finaliza todo lo que puedas en conjunto.

## Trabajando con Git

Git es una herramienta que es accesible desde terminal y también desde diferentes clientes que simplifican la interacción a través desde interfaces más gráficas. Pero al final, lo que opera por debajo son los comandos de Git, que serán los que conocemos ahora.

Como en el desarrollo de software, primero necesitaremos ver las entrañas del mismo para aprender en profundidad y enfrentarnos a situaciones más problemáticas cuando estas lleguen.

A través de comandos, conocemos el estado de nuestros repositorios y operamos con ellos. Poco a poco iremos conociendo estos comandos. Abre tu terminal, ya sea la integrada en el sistema operativo, o bien la que nos ofrece Visual Studio Code, que ya debe estar instalado en tu equipo para la realización de este curso.

Para conocer la versión de Git instalada en nuestro equipo, en caso de necesitar conocerla, ejecutaremos:

```
git --version
```

**Inicializando un repositorio**

Un repositorio no se inicia de forma automática, solo cuando clonamos un repositorio ya tendremos un directorio bajo el control de Git.

Si queremos iniciar un nuevo proyecto y ponerlo bajo el control de Git, abriremos el terminal, nos situaremos en la carpeta donde queremos abrir un repositorio y ejecutamos el siguiente comando:

```
git init
```

Esa carpeta y todas las que crees dentro de ella, estarán bajo el control de Git y serán parte del repositorio. **Fuera de esta carpeta, el repositorio no existe.** Si ejecutas los comandos que vamos a explicar fuera de una carpeta que no esté bajo el control de Git, verás cómo aparecen fallos pues no existe un repositorio sobre el que operar.

Si vas a ejecutar comandos de Git, asegúrate de que estás en una carpeta que esté en un repositorio, que esté bajo el control de Git.

**Control de cambios**

Puesto que cada cambio que se realice sobre el repositorio es detectado y registrado, necesitaremos conocer qué ha cambiado en nuestro repositorio. Esto lo comprobamos a través del siguiente comando:

```
git status
```

Cuando se ejecuta, veremos qué ficheros han sido modificados en una lista. Solo cuando un directorio está bajo el control de Git este comando tiene efecto.

¿Cuales son los cambios que se han introducido en la lista de estos ficheros? Para ello tenemos otro comando:

```
git diff
```

Se mostrarán por consola, paginados, todos los cambios que se han realizado en el repositorio y que no han sido añadidos en un *commit*. Si son muchos tendrás que ir avanzando con las teclas de dirección hacia arriba o hacia abajo para ver qué líneas han cambiado, agrupados por fichero.

Esta muestra de cambios se puede parametrizar, para comodidad del mostrado de los cambios. El siguiente comando muestra los cambios realizados en la misma línea, mostrando por color el contenido que se ha eliminado y que se ha añadido:

```
git diff --color-words
```

**Añadiendo ficheros**

Si hemos realizado varios cambios en nuestro repositorio y todo está preparado y probado para realizar un *commit*, necesitaremos especificar qué ficheros van a ser includos en este.

¿Con qué ficheros vamos a realizar el siguiente *commit*? Cambiaremos su estado de *unstaged* a *staged*. Para añadir todos los cambios, escribimos el siguiente comando:

```
git add *
```

En cambio, si queremos añadir solo unos ficheros o cambios específicos, seremos más selectivos, especificando qué ficheros entran o qué directorios.

```
git add onefile.cs
```

O quizás, para añadir todos los ficheros cambiados de un directorio concreto:

```
git add dir/*
```

**Haciendo *commits***

¿Todo preparado? ¿Los cambios que has almacenado van a ser parte de un *commit*? Un commit está compuesto de varias partes, como ya conocemos. Una de ellas, será un mensaje explicativo de los cambios que contiene.

Evita los mensajes vagos o poco aclaratorios, puesto que deben ser documentación de lo que has realizado, para guiar a las personas que trabajan contigo o simplemente a tu versión del futuro que necesita recordar qué cambios realizaste.

Para realizar un *commit* ejecuta el siguiente comando:

```
git commit
```

Se abrirá entonces el editor de texto plano por defecto en su sistema operativo o aquel que determinaste en la instalación de Git. Ahí especificarás el mensaje del *commit*. Puedes especificar, en la primera línea, el resumen de los cambios introducidos.

En las siguientes líneas hay espacio para información más detallada y de utilidad para completar el resumen del *commit*, o quizás para mencionar a personas implicadas... todo aquello que consideres necesario o aquello que se haya marcado como convención en el equipo de desarrollo.

Aunque... ¿es posible realizar *commit* y añadir un mensaje en el mismo comando? La respuesta es sí, y se realiza con el siguiente comando y parámetros:

```
git commit -m "mensaje del commit"
```

## Los *commits* del pasado

Una vez hemos hecho algunos *commits*, o quizás otras personas en el equipo, necesitaremos ver la lista hasta ese momento.

Es tan sencillo como ejecutar:

```
git log
```

Podemos especificar un comando en el que podemos ver incluso la estructura de ramas en la consola, ejecutando:

```
git log --graph --oneline --all
```

Las ramas diferentes, los *mergeos* realizados y mucha más información útil, se puede visualizar a través de la consola.

### ***Stash*** **o borradores**

En ocasiones, necesitas cambiar de rama, almacenar trabajo en curso sin hacer *commit* para atender otra tarea más urgente. O simplemente hay cambios que quieres guardar y que atenderás más adelante.

Existe el concepto de *stash*, que funciona como un borrador. Los cambios dejan de estar disponibles, es como un falso *commit* que más tarde puedes recuperar. Para crear un *stash*, marca los ficheros modificados que te interesen como *staged* y ejecuta en tu terminal:

```
git stash
```

Ahora puedes cambiar de rama o atender otras tareas sin haber perdido los cambios con los que estabas trabajando anteriormente.

Cuando necesites recuperarlos, basta con ejecutar en tu terminal:

```
git stash apply
```

Ten en cuenta que se recupera entonces el último *stash* almacenado, pues funciona como una pila o *stack*. Puedes ver todos los *stash* almacenados con el siguiente comando.

```
git stash list
```

¿Cómo puedes recuperar un *stash* específico? Si observas en la lista, todos los *stash* vienen numerados. Basta con especificar cuál quieres recuerar de la siguiente forma, especificando el número del stash en cuestión:

```
git stash apply 3
```

### **Descarte de los cambios (*danger zone*)**

Si has hecho cambios que no vas a incluir en un *commit* o que han quedado desfasados por cualquier motivo, puedes deshacer los cambios de forma directa. **Esta acción no se puede deshacer**, así que obra con cuidado, **puedes perder trabajo útil de forma que no puedas recuperarlo**.

```
git reset --hard
```

Una alternativa puede ser almacenarlos en un *stash* de forma temporal, asegurando así que no vas a perder ese trabajo. Descarta de esta forma solo si te has asegurado de que es trabajo que no vas a

necesitar.

**Operando con ramas**

Las ramas, como ya sabemos, son las diferentes versiones que tenemos a nuestro alcance en un repositorio. Se gestionan todas a la vez, pero solo podemos acceder a una al mismo tiempo.

¿Cómo puedo saber en qué rama me encuentro en un determinado momento? Ejecutando el siguiente comando.

```
git branch
```

Si observamos, vemos todas las ramas que están en nuestro repositorio local y en la que nos encontramos destacada sobre el resto.

Para cambiar de una rama a otra:

```
git checkout otra-rama
```

Si volvemos a ejecutar `git branch` comprobaremos que el cambio ha sido realizado. **Cuidado:** no podremos cambiar de rama si tenemos cambios pendientes de gestionar. Tenemos varias alternativas en este caso:

- No cambiar de rama y seguir trabajando en la que nos encontramos.
- Almacenar los cambios en un *stash*, tal y como hemos aprendido, y aplicarlos de nuevo una vez hayamos cambiado de rama, para seguir trabajando sobre ellos.
- **Descartar los cambios, opción no recomendable si no lo estudiamos con atención**. Puede ser que no sean útiles o sea trabajo a descartar. En cualquier caso, estudia bien y considera con cuidado antes de realizar esta acción. **Es trabajo que no podrás recuperar**.

Para añadir una nueva funcionalidad y no afectar directamente a la rama principal, crearemos nuestras propias ramas que más adelante fusionaremos (a través de *merge*) con la rama principal o la de desarrollo:

```
git branch nueva-rama
```

Entonces navegaremos a esta nueva rama que hemos creado para trabajar sobre ella, a través de `git checkout`. ¿Es posible crear y viajar a una nueva rama en el mismo comando? La respuesta es afirmativa, solo tienes que ejecutar el siguiente comando:

```
git checkout -b otra-nueva-rama
```

**Trabajando en remoto**

Los `remotes` son las diferentes fuentes de código con las cuales puedes operar. Imagina que tienes que añadir tus cambios dos repositorios idénticos. En este hipotético caso, cada repositorio sería un `remote`. Los `remote` tienen un nombre que los identifica, y puedes verlos de la siguiente forma:

```
git remote
```

De esta forma, vemos los `remotes` que tienes, pero solo el nombre. Si quieres ver el nombre y a la URL a la que apunta, ejecuta:

```
git remove -v
```

No es algo inamovible, por lo que puedes ir añadiendo, eliminando o modificando según tus necesidades. Para añadir un `remote`:

```
git remote add nombre-remote url-remote
```

Para eliminar un remote:

```
git remote remove nombre-remote
```

**Repositorios ya existentes**

Puede ser que te hayas incorporado a un equipo de desarrollo que está trabajando con un proyecto que tiene ya un cierto tiempo de vida. O quizás hayas visto un repositorio en GitHub u otra plataforma al que quieres contribuir. Para eso necesitarás clonar ese repositorio, es decir, crear una copia local con la que trabajar, realizar cambios y ofrecerlos para incorporarlos al repositorio original.

Para ello, ejecutaremos el siguiente comando:

```
git clone url-repositorio
```

Se creará entonces un directorio con el clonado del repositorio en cuestión, con todos los ficheros, directorios que incluya... así como todo el historial de *commits* que se haya realizado sobre él.

Si ejecutas `git remote`, podrás ver el origen, la fuente de la cual hemos clonado este repositorio. Y como ya conocemos, podemos eliminarla, añadir otra complementaria o cualquier otra operación que necesitemos.

**Actualizando ramas**

A medida que pasa el tiempo, se van introduciendo cambios en nuestro código. No queremos trabajar sobre código desactualizado, siempre necesitamos trabajar con las últimas versiones, es vital para evitar introducir bugs o enfrentarnos a *merges* complicados.

Existen dos formas de actualizar nuestras ramas con los *commits* que tenemos de forma remota. Recuerda que trabajamos con ramas, de la cual tenemos nuestra versión en local y la que actualizamos constantemente, en remoto, en nuestro repositorio remoto.

Puedes entonces actualizar las ramas sin descargarlas en tu repositorio local, o actualizarlas descargándolas. La primera forma simplemente te hace consciente de los *commits* que se han hecho e incorporado en otras ramas, sin que afecte a tu código. La segunda te las descarga en tu repositorio local, para que puedas operar con ellas, y si existen conflictos entre lo que descargas y los cambios con los que estás trabajando, toca resolver y decidir con cuales te quedas.

Para actualizar las ramas **sin descargar** con las que trabajas necesitarás ejecutar el siguiente código, hacemos `fetch`:

```
git fetch una-rama
```

Para hacer `fetch` de todas las ramas, ejecutamos:

```
git fetch --all
```

Si queremos especificar otro `remote`, añadiremos un parámetro:

```
git fetch remote rama
```

Para actualizar las ramas **descargando su contenido** con las que trabajas necesitarás ejecutar el siguiente código, hacemos `pull`:

```
git pull otra-rama
```

Para hacer `pull` de todas las ramas, ejecutamos:

```
git pull --all
```

Si queremos especificar otro `remote`, añadiremos un parámetro adicional:

```
git pull remote rama
```

**Subiendo cambios**

Una vez has realizado tus cambios, hecho *commits* de los mismos, necesitarás almacenarnos en el repositorio en remoto. Es una forma de poder trabajar desde cualquier máquina, pudiendo descargar esa rama desde cualquier parte tal y como hemos aprendido.

Es también una medida de seguridad, pues si ocurre una catástrofe y tu equipo no es accesible por algún motivo, no has perdido el trabajo realizado durante horas, quizás días.

¿Cómo los almacenas en remoto, donde sabes que estarán a salvo y siempre accesibles? Haciendo `push` de los cambios, "empujándolos" hacia una localización remota:

```
git push
```

Si necesitas especificar una rama en la que no te encuentras actualmente, puedes ejecutar:

```
git push una-rama
```

Y si también necesitas especificar otro `remote`, escribe en tu terminal:

```
git push remote la-rama-definitiva
```

**Incorporando cambios, *merge***

Ha llegado la hora de realizar una fusión, un *merge* de cambios, de forma manual. Lo primero que haremos es movernos hacia la rama que va a recibir los cambios, la rama objetivo. Una vez allí, debemos tener los cambios con los que vamos a fusionar, es decir: tener las ramas actualizadas.

¿Estamos listos? ¿Todo en orden? Entonces puedes comenzar el merge con el siguiente comando:

```
git merge rama-con-cambios
```

***Tags***

Las *tags* o etiquetas son referencias a un punto específico en el historial de Git. Se captura un punto concreto, un *commit* que se marca para una versión. Es como una rama que nunca va a cambiar, se congela en el tiempo pues marca una versión de la aplicación. La aplicación seguirá evolucionando, y llegarán nuevas versiones. Pero una vez se ha creado, no cambiará.

Para crear una *tag*, nos moveremos a la rama en la que queremos crear una nueva versión. Y desde nuestro terminal, ejecutaremos:

```
git tag -a version1 -m "La versión definitiva"
```

Como podemos ver, el parámetro `-a` le pone nombre a la versión, y el parámetro `-m` le da una descripción a la misma. El parámetro `-a` puede ser "version1" o quizás "v1.4", o incluso "v1.3". Todo depende de la convención elegida por el equipo de desarrollo.

Si queremos ver todas las *tags* en nuestro código, podemos ejecutar:

```
git tag
```

Puedes ver los detalles de una *tag* (*commit* sobre el que se ha realizado, la fecha y la autoría) con este comando:

```
git show nombreTag
```

## Y esto es solo el principio

Git es una gran herramienta, sobre todo flexible, para manejar de forma eficiente nuestro código fuente. Recuerda que la comunicación es vital para cuidar de forma efectiva nuestro código y mantener su buena salud.

Experimentar es también muy importante, la consola es otra herramienta más y está a nuestro servicio para poder realizar toda clase de procesos. Piérdele el miedo y agiliza tu manejo con la terminal.

En futuros módulos veremos y comprobaremos cómo de importante es el control de versiones en nuestros procesos de integración de cultura DevOps.

### Desde NeonCoding

El equipo de NeonCoding lleva trabajando con Git desde sus inicios, parte del equipo con menos experiencia, como por ejemplo Yeray.

| | |
|---|---|
|  | La consola o terminal me da un poco de miedo a veces, pero si trabajo con cuidado y atención todo va bien, cada vez tengo más soltura con Git. |
|  | Poco a poco, con la experiencia, irás ganando confianza. Además, ya ves que es muy útil para revisar los cambios que se realizan. |

Así pues, en NeonCoding, es una experiencia diaria, que se mejora y perfecciona en el día a día, con grandes beneficios para el trabajo que tienen entre manos y para el que vendrá en el futuro.