# Curso de Git

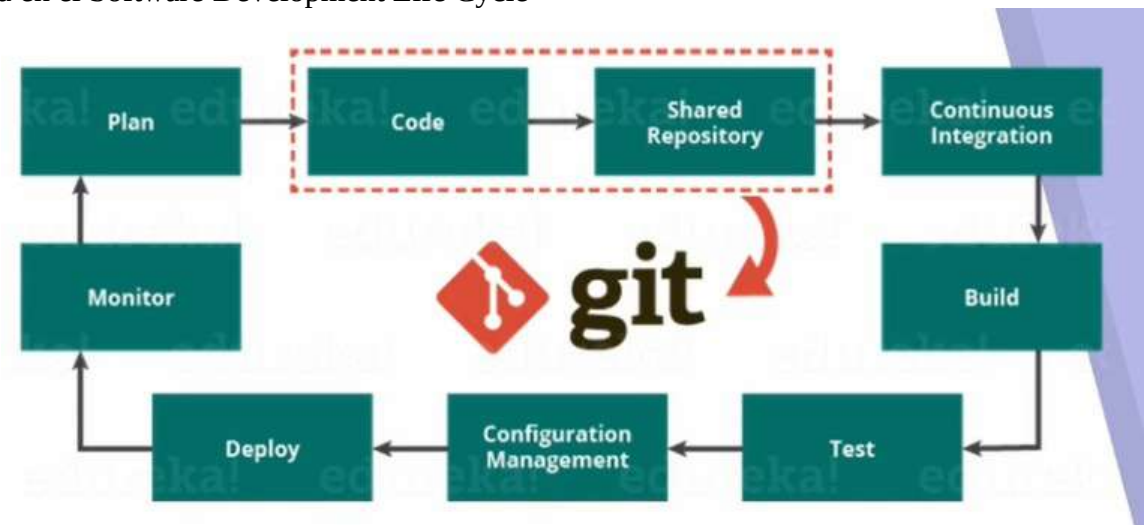
Manuel Vergara

## Índex de continguts

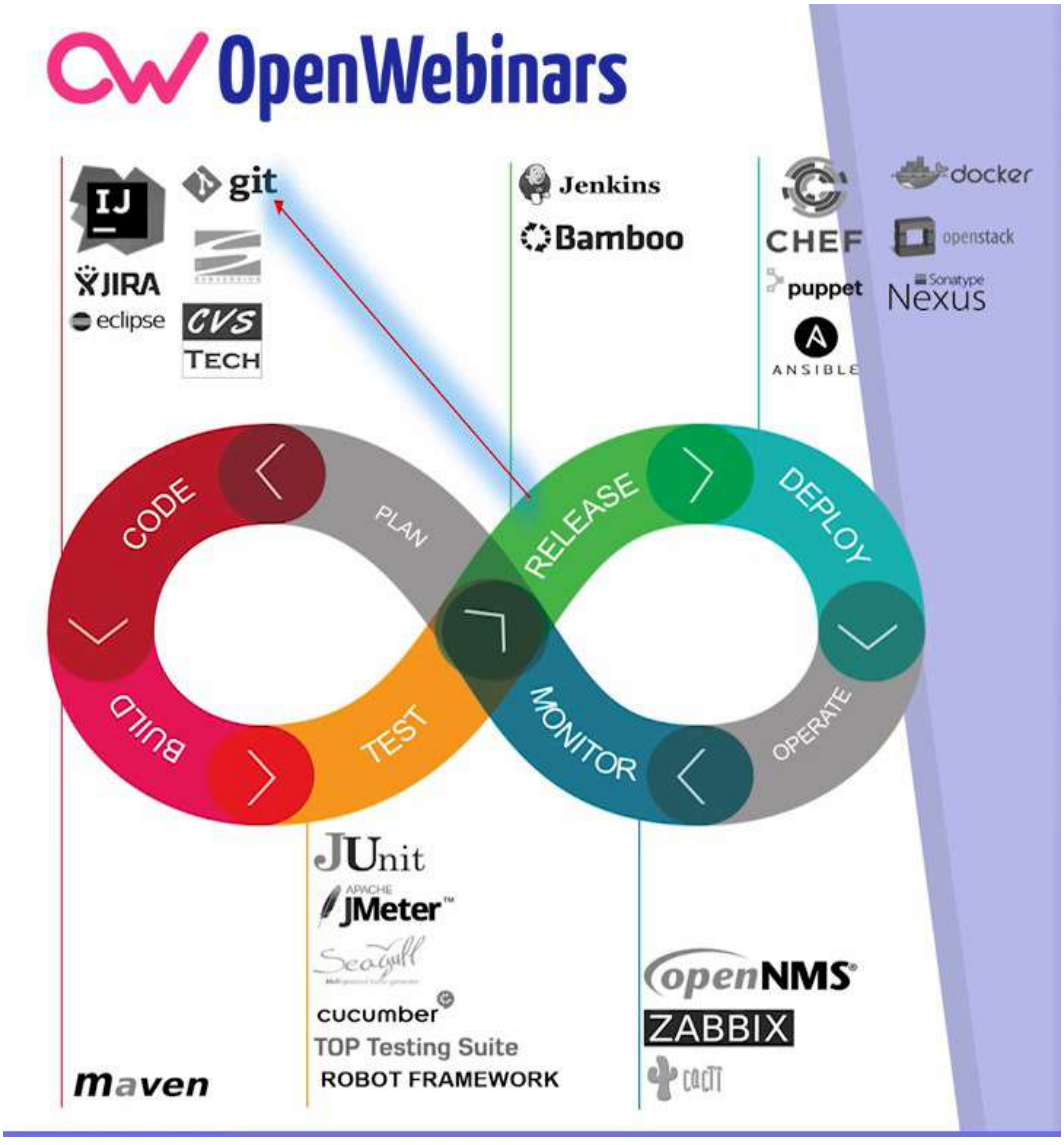
Introducción....3
Características....4
Sistemas de control de versiones centralizado (Subversion)....5
Sistemas de control de versiones centralizado (Git)....5
Detalle de estados de un fichero en Git....5
Descarga de GIT....6
Descarga de TkDIFF....6
Configuración....7
Organización del código fuente y otros....7
Crear un repositorio....7
Paso 1 - Git Status....7
Paso 2 - Git Diff....8
Paso 3 - Git Add....9
Paso 4 – Diferencias en estado preparado (Staged)....9
Paso 5 - Git Log....9
Paso 6 - Git Show....10
Trabajo remoto....10
Paso 1 - Git Remote....11
Paso 2 - Git Push....11
Paso 3 - Git Pull....12
Paso 4 - Git Log....12
Paso 1 - Git Checkout....13
Paso 2 - Git Reset....14
Paso 3 - Git Reset Hard....14
Paso 4 - Git Revert....14
Paso 5 - Git Revert....15
Paso 1 - Git Merge....16
Flujos de trabajo....16
Paso 1 - Git Branch....16
Paso 2 – Listar las ramas....17
Paso 3 – Hacer fusión (merge) a master....17
Paso 4 - Push Branches....17
Paso 5 – Limpiar ramas....18
Para encontrar errores....18
Conflicto de archivos locales con los remotos....18
Reescribir la historio de un proyecto....19
Etiquetas....19
Paso 4 - Listar Tus Etiquetas....19
Paso 5 - Crear Etiquetas....20
Paso 6 - Etiquetas Anotadas....20
Paso 7 - Etiquetas Ligeras....21
Moverse adelanta y atrás entre estados de un repositorio....21

# Introducción

Ayuda en el Software Development Life Cycle




Ciclo de desarrollo de software



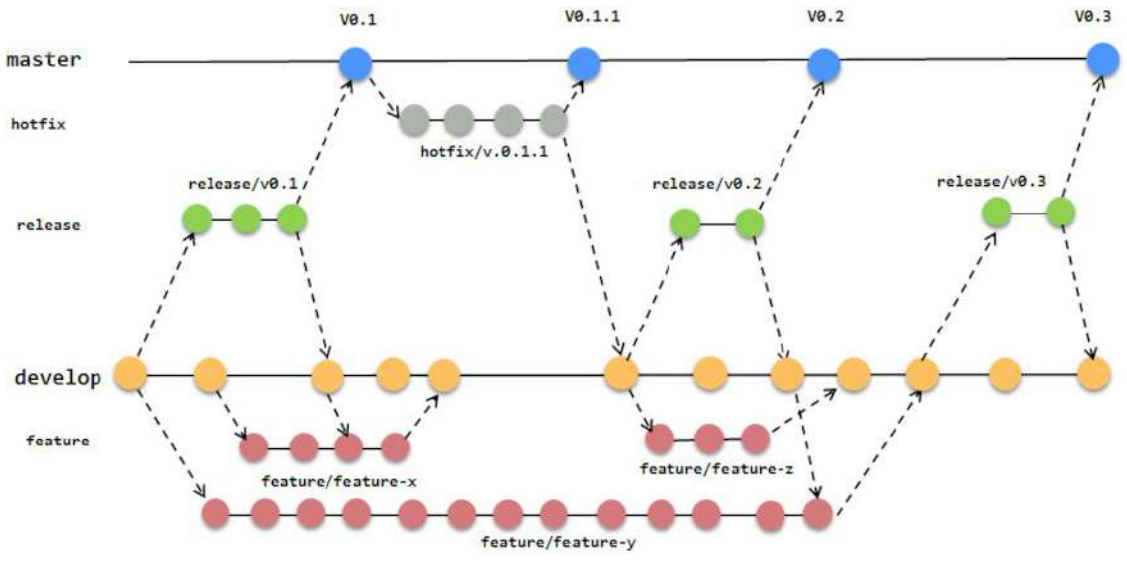
Git vs SVN

| | SVN | Git |
|---|---|---|
| Control de versiones | Centralizada | Distribuida |
| Repositorio | Un repositorio central donde se generan copias de trabajo | Copias locales del repositorio en las que se trabaja directamente |
| Autorización de acceso | Dependiendo de la ruta de acceso | Para la totalidad del directorio |
| Seguimiento de cambios | Basado en archivos | Basado en contenido |
| Historial de cambios | Solo en el repositorio completo, las copias de trabajo incluyen únicamente la versión más reciente | Tanto el repositorio como las copias de trabajo individuales incluyen el historial completo |
| Conectividad de red | Con cada acceso | Solo necesario para la sincronización |

Para observar las diferencias de una versión de código y otra: tkdiff (Creo que git en linux ya tiene un diff incluido)

Git es:

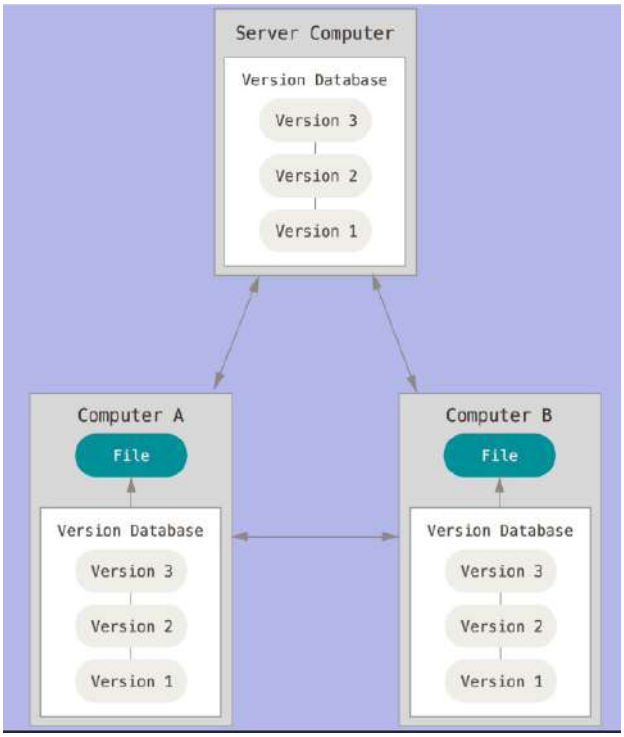
- Un sistema distribuido de control de versiones
- Muy potente
- No depende de un repositorio central
- Es software libre
- Disponemos de un historial de revisiones completo.
- Trabajar con ramas (branches) diferentes de código y fusiones (merges) de ramas de código es un proceso ágil




# Características

- Los VCS centralizados suponen un punto único de fallo. (subversion)
- Los sistemas de Control de Versiones Distribuidos (DVCS) salvan este problema.
- En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes replican completamente el repositorio

Replicas de versiones de proyectos en Sistemas de control de versiones Distribuidos




## Sistemas de control de versiones centralizado (Subversion)


*Figure 4. Almacenamiento de datos como cambios en una versión de la base de cada archivo.*

SVN maneja la información como un conjunto de archivos y las modificaciones hechas a cada uno de ellos a través del tiempo (deltas)

## Sistemas de control de versiones centralizado (Git)

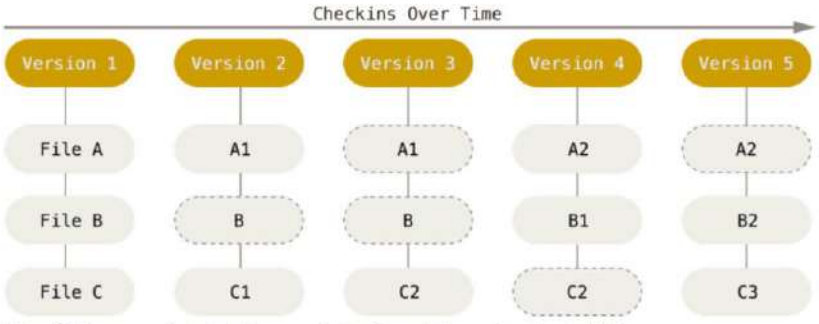


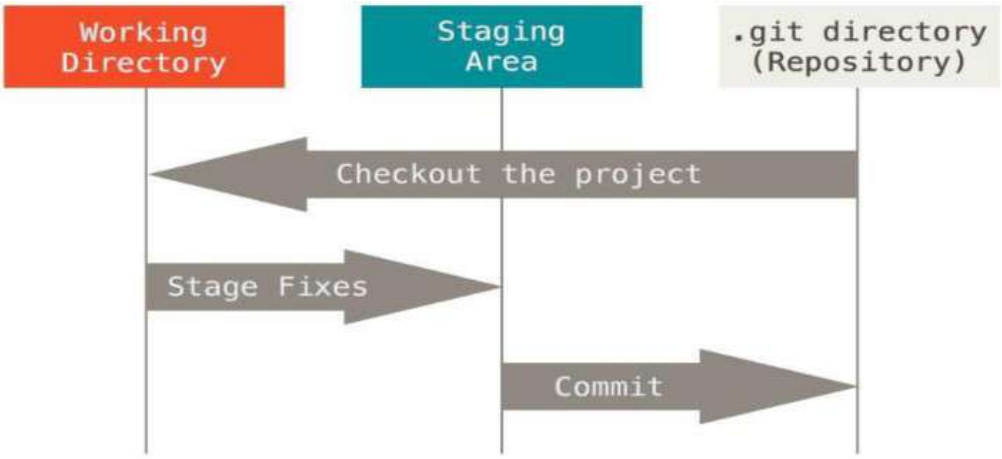

*Figure 5. Almacenamiento de datos como instantáneas del proyecto a través del tiempo.*

Git maneja sus datos como una secuencia de copias instantáneas.

## Detalle de estados de un fichero en Git

- **Confirmado** significa que los datos del archivo están almacenados de manera segura en tu base de datos local (commited).
- **Modificado** significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos (untracked)
- **Preparado** significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación (staged)

Esto nos lleva a las tres secciones principales de un proyecto de Git: El directorio de Git (Git Directory, .git), el directorio de trabajo (working directory), y el area de preparacion (staging area).




Git es software libre con una enorme comunidad. Es potente, distribuido y ágil

Existe un git para windows

tkdiff vs vimdiff

vimdiff está incluido en distro de GIT para linux

vim vs nano

Cambiar editor:

git config –global core.editor nano

¿Dónde se guardan las contraseñas?
Windows es credential.helper
Linux htpasswd ¿?

Opciones gráficas:
Atlassioan SourceTree
GitKraken - Buena opción multiplataforma y open source

Se utilizan TAGS para nombrar los códigos numéricos que identifican las versiones completas de código de un proyecto.

## Descarga de GIT

https://github.com/git-for-windows/git/releases/download/v2.18.0.windows.1/Git-2.18.0-32-bit.exe

## Descarga de TkDIFF

http://www.posoft.de/html/tkdiffMain.html

Por defecto, tenemos vimdiff en una distro de GIT para las diferencias de ficheros, pero si este binario lo metemos en:

c:\archivosdeprograma\git\usr\bin\tkdiff

## Configuración

git config —global user.name “TU NOMBRE”
git config —global user.email tunombre@tudominio.com
git config —global diff.tool vimdiff
git config —global diff.tool tkdiff
git config —global credential.helper manager
**En un windows si trabajamos en proyectos compartidos (LF a CRLF)**
git config —global core.autocrlf true
**En un linux, para que haga lo contrario (LF a CRLF):**
git config —global core.autocrlf input
**Dejarlo como está, sin alterar nada (para proyectos en los que todos tendremos el mismo SO)**
*git config —global core.autocrlf false*

# Organización del código fuente y otros

GIT permite disponer de un mirror completo del repositorio de código en local. Podemos seguir trabajando aunque no estemos conectados.

## Crear un repositorio

Comandos

- git init # Re/iniciando git
- git status # Ver estado de la
- git add # Añadir ficheros
- git commit # Crear una instantánea, un punto de grabación
- git ignore
- git diff
- git log # Revisar los registros. Con –oneline parecen resumidos
- git show

Crear un .gitignore para crear una especie de máscara de exclusión. Incluimos ficheros que ignora el comando git add. En cada línea será un fichero o una expresión regular para indicar varios ficheros. Ejmp:

*.tmp

Podemos indicar los ficheros al comando git add. Por ejemplo, añadir todo: git add .
Esta opción no añade los ficheros ocultos de linux (comienzan con un punto), hay que nombrarlos.

## Paso 1 - Git Status

Tal y como se ha discutido en la lección previa, git status nos permite visualizar los cambios en el *working directory* y en la *staging area* comparando con el repositorio.

Dado que el repositorio actual en el que lanzamos git status muestra que un cambio hecho en nuestro *working directory* y tenemos en él un fichero previamente aprobado (por ejemplo hello-world.js), si lo modificamos, y a la vez creamos otro fichero, commited.js, que ni siquiera ha sido movido aún a *staging area*, ¿qué nos mostrará?

Modificamos fichero “hello-world.js” agregándole un comentario.

Creamos un fichero commited.js

Lanzamos

```
git status
```

## Paso 2 - Git Diff

El comando git diff permite comparar cambios en el *working directory* contra una versión previamente aprobada. Por defecto, el comando compara el *working directory* y el commit que llamamos *HEAD* (el más reciente).

Si se desea comparar el *working directory* contra una versión más antigua, entonces podemos proporcionar su *hash_id* como parámetro en la llamada al comando; así: git diff

Comparar contra *commits* mostrará los cambios en todos los ficheros que se hayan modificado. Si deseamos comparar los cambios en un único fichero, podemos pasar su nombre como parámetro:

```
 git diff committed.js
```

Incluso ambas funcionalidades, así:

```
 git diff <commit> committed.js.
```

Por defecto, la salida está en formato diff.

By default the output is in the combined diff format. The command git difftool will load an external tool of your choice to view the differences.

En nuestro caso, elegiremos tkdiff, descargándola y colocándola donde se ha indicado arriba.

Si queremos vimdiff:

```
git config --global diff.tool vimdiff
```

Si queremos tkdiff:

```
git config --global diff.tool vimdiff
```

Hagamos una prueba modificando un fichero respecto de la versión anterior que esté aprobada.

Ejecutemos:

```
git diff
git difftool
```

Y veamos las posibilidades que tiene la herramienta.

## Paso 3 - Git Add

Anteriormente, vimos que podíamos aprobar un cambio mediante *commit* siempre y cuando previamente hubiésemos agregado el archivo al área de preparación (stage) mediante el comando git add .

Si renombramos (*mv*) o borramos (*rm*) ficheros, se necesita especificar esos ficheros con un git add para que sean registrados como un cambio. Esto a veces resulta confuso.

De forma alternativa, se puede usar git mv y el comando git rm de git para llevar a cabo la acción sobre el fichero, y además incluir el cambio en la *staging area*.

Practicamos agregando cambios con el comando git add. Practicamos haciendo renombrado y renombrado sin “git mv” y sin “git rm”. ¿Qué sucede?

Lo solucionamos con git add.

Ahora repetimos el proceso con git mv y el comando git rm. ¿Qué sucede?

## Paso 4 – Diferencias en estado preparado (Staged)

Una vez que los cambios están en la *staging area* no se mostrarán en la salida de git diff

Por defecto, git diff solo comparará el *working diretory* y no el *staging area* .

Para comparar cambios en el *staging area* contra el *commit* previo, debemos proporcionar el parámetro git diff —staged . Esto posibilita asegurarnos de que hemos preparado (*staged*) todos nuestros cambios.

- Lanzamos git rm README.md y efectuamos un commit.
- Ahora lanzar “git diff” y “git diff —staged” y ver si hay diferencia.
- Creamos un archivo README.md con nano y agregamos contenido.
- Posteriormente, hemos de lanzar “git diff” y “git diff —staged” y ver si hay diferencia.
- Ahora lanzamos git add de ese fichero recién creado.

Si lanzamos ahora ambos comandos, ¿hay alguna diferencia?

## Paso 5 - Git Log

El comando git log permite visualizar el histórico del repositorio y el *commit log* .

El formato de la salida de log es muy flexible. Por ejemplo para sacar cada *commit* en una única línea, el comando es:

```
 git log --pretty=format:"%h %an %ar - %s"
```

También funciona así:

```
git log --pretty="%h %an %ar - %s"
```

Podemos encontrar más detalles y opciones usando:

```
git log --help
```

Probar la salida del comando git log agregando las opciones descritas.

### Paso 6 - Git Show

Mientras que git log te dice el autor del *commit* y el mensaje, para visualizar los cambios efectuados en el *commit* se necesita usar el comando git show .

Al igual que otros comandos, por defecto, “git show” muestra los cambios del *commit* HEAD. Podemos usar el comando git show

Usar git log para obtener varios números de commit. Lanzar git show con dos de ellos.

## Trabajo remoto

escenario2

Descargar de github

Vinculamos con el repositorio
    git remote add origin https://github.com/iicc1/WarnBot_Telegram-Bot.git
Se usa la etiqueta origin porque es la más recomendada, el contenido de github es master y el contenido que tendremos en local será origin

Ahora mostramos información del repositorio vinculado
    git fetch origin
Esto no significa que descargamos, tan solo vemos información del repositorio vinculado, en remoto.

Podemos ver las ramas que nos podemos descargar conjunto
    git branch -v -a
Por defecto, será la rama master

Ahora descargamos indicando a donde queremos descargar y desde donde
    git pull origin master
Ahora ya lo tendremos en local

Cuando descargamos git clone estaremos haciendo a la vez
    git init
    git fetch
    git pull
Ejemplo:
    git clone https://github.com/iicc1/WarnBot_Telegram-Bot.git
Esto hará que se baje también la carpeta del repositorio con la estructura de direct/archiv dentro.
Para descargar directamente en la ubicación actual se añade un punto
    git clone https://github.com/iicc1/WarnBot_Telegram-Bot.git .

En esta lección aprenderemos cómo podemos compartir cambios en nuestro repositorio con otras personas, y combinar sus cambios dentro de nuestro repositorio.

Con Git como DVCS, tenemos nuestro repositorio local conteniendo todos los logs, ficheros y cambios realizados desde que se inicializó el repositorio. Para asegurar que todos están trabajando en la copia más reciente, es necesario compartir los cambios. Cuando compartimos estos cambios con otros repositorios, solo se sincronizarán las diferencias, por lo que es un procedimiento extremadamente rápido.

## Paso 1 - Git Remote

Los repositorios remotos permiten compartir cambios desde o hacia nuestro repositorio. Las ubicaciones remotas son generalmente servidores locales, una máquina de un equipo de trabajo o bien un almacén de repositorios en la nube como GitLab o Github.

Los repositorios remotos se añaden usando el comando git remote con un nombre amigable y una ubicación remota; normalmente una conexión HTTPS o SSH (para esto último no hace falta software específico como GitLab).

Ejemplo: https://github.com/sharkdp/bat

El nombre amigable permite referenciar la localización en otros comandos. Nuestro repositorio local puede referenciar múltiples repositorios remotos, dependiendo de nuestras necesidades.

Trabajaremos en otro directorio. Crearemos la carpeta scn3 y esa será nuestro *working directory*.

Agregaremos el repositorio remoto https://github.com/sharkdp/bat usando el comando git remote con el nombre amigable origin.

Formato de llamada:

```
git remote add origin /s/remote-project/1
git fetch origin
```

Si usamos *git clone* que será tratado en una lección futura, cuando se clona el repositorio, se agrega automáticamente como descriptor amigable que lo referencia, el nombre *origin*.

## Paso 2 - Git Push

Cuando estamos listos para compartir nuestros *commits*, es necesario hacer un *push* de ellos a un repositorio remoto usando *git push*. Un flujo de trabajo habitual de GIT sería llevar a cabo múltiples *commits* pequeños conforme vamos finalizando tareas, y hacerles un *push* a un repositorio remoto en hitos relevantes, como cuando finalizamos un bloque de trabajo, de manera que aseguramos la sincronización del código dentro de un equipo.

El comando *git push* se acompaña de dos parámetros. El primero es el nombre amigable del repositorio remoto (normalmente *origin*). El segundo es el nombre de la rama (normalmente, rama *master*). Por defecto, todos los repositorios tienen una rama *master* donde se trabaja con el código.

Haremos *push* de los *commits* de la rama *master* a la ubicación remota (*origin*).

Para este ejercicio, vamos crearnos un usuario en gitlab. Crearemos un nuevo proyecto y nos vincularemos a él. Hacemos un pull. Posteriormente, crearemos un archivo (por ejemplo

README2.md), y lo aprobaremos *commit* en nuestro master. Por último, lo subiremos al repositorio de Gitlab

## Paso 3 - Git Pull

El comando *git push* permite subir los cambios a un repositorio remoto, mientras que *git pull* funciona de forma inversa. El comando *git pull* permite sincronizar cambios de un repositorio remoto en nuestra versión local.

Los cambios desde el repositorio remoto son automáticamente fusionados (*merge*) en la rama en la que estamos trabajando en el momento de lanzar el comando.

- Crear un nuevo repositorio llamado scn3-pull.
- Descargar (*pull*) los cambios de una ubicación remota https://github.com/sharkdp/bat a nuestra rama *master.*
- Borrar los archivos siguientes appveyor.yml Cargo.lock Cargo.toml LICENSE-APACHE LICENSE-MIT README.md
- Agregar un archivo LEEME.txt
- Aprobar los cambios.
- Lanzar un “ls” para ver los archivos de la rama *master.*
- Hacer un *checkout* para cambiarnos a *origin/master*.
- Lanzar un “ls” para ver los archivos de la rama *origin/master.*
- Conmutarnos de nuevo a la rama master y volver a lanzar un comando “ls”

En el siguiente paso exploraremos qué cambios se han efectuado.

IMPORTANTE: Podemos crear y cambiar de rama con un mismo comando.

```
git checkout -b nombre-rama
```

Esto nos capacita para descargar una versión de repositorio, y luego versionar

## Paso 4 - Git Log

Tal y como se ha descrito en la lección previa, podemos usar *git log* para ver el histórico del repositorio. El comando *git show* permitirá ver los cambios realizados en cada *commit*.

En esta prueba, vamos a usar el repositorio anterior, que tenemos vinculado al proyecto BAT y vamos a filtrar los commits que lleven la cadena Fix o fix. Al más reciente de los cuales, le lanzaremos un *git show <hashid>* para visualizar los cambios.

Usar el modificador –pretty para llevar a cabo la consulta de commits en el repositorio remoto de BAT

Una de las ventajas principales cuando usamos un sistema de control de versiones es la capacidad de deshacer cambios y volver a una versión previa. Git proporciona enfoques poderosos y control sobre la gestión de los repositorios y su histórico. Exploraremos esas posibilidades en esta lección.

Notas previas:

hace falta crear un repositorio git con algunos commits previos

## Paso 1 - Git Checkout

Cuando trabajamos con Git, una lección habitual es deshacer cambios en nuestro *working directory*. El comando *git checkout* reemplazará todo en el *working directory* a la última versión aprobada (*commited*).

Si deseamos reemplazar todos los ficheros hacia el *working directory*, usamos el carácter punto (“.”) para referenciar al directorio actual. En otro caso, se proporciona una lista de directorios/ficheros separados por espacios.

Como sucede que *git checkout* acepta tanto nombres de rama, como ficheros (o path de ellos), se puede usar el indicador “—” para forzar a Git a interpretar que cualquier cosa que venga después de “—”, es un nombre de fichero (o todos los ficheros); en otro caso, se interpretará como el nombre de una rama.

Se ilustra en este ejemplo la diferencia:

Supongamos que tengo un archivo llamado *master* pero también una rama llamada *master* .

El comando

```
git checkout master
```

Haría un checkout de la rama, pero el comando:

```
git checkout -- master
```

chequearía el fichero *master.*

IMPORTANTE!! El comando

```
git checkout HEAD -- .
```

forzaría a reemplazar TODOS LOS ARCHIVOS (de ahí el “—”) lo que teníamos en el último commit a nuestro *working directory* (que es el “.”).

Si tenemos algo en staging y queremos eliminarlo de staging, entonces usamos:

```
 git reset HEAD fichero
```

Git checkout, no toca el área de preparación o stagging rea. Para eso tenemos un git reset.

Usar *git checkout* para limpiar o deshacer los cambios del *working directory.*

Pasos:

- Creamos un directorio de trabajo, entramos en él y lo inicializamos.
- Creamos el README.md, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero1, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero2, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero3, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Hacemos un borrado de “fichero*” y nos aseguramos de que el borrado esté en* staging*.

- Lanzamos “ls -l”
- Luego reemplazamos todo el contenido del directorio con lo que existiese en el HEAD de la rama master.

## Paso 2 - Git Reset

Si estamos en la mitad de un *commit* y tenemos agregados ficheros a la *staging area* pero cambiamos de idea, entonces necesitaremos usar el comando *git reset* . El comando *git reset* eliminará a los ficheros de la *staging area* y los dejará en el *working directory* en estado *untracked* (no registrados).

Si deseamos sacar todos los ficheros de la *staging area*, debemos usar el carácter “.” para indicarlo. En otro caso, se admite una lista de ficheros separados por espacios.

Esto es muy útil cuando intentamos mantener nuestros *commits* pequeños y concentrados de tal forma que podamos sacarlos de *staging* si hemos añadido demasiados de una tacada.

Creamos un escenario de nuestra elección. Hacemos un par de commits.

Por último creamos un fichero más. Lo agregamos a staging.

Lanzamos *git reset* para dejarlo en estado *untracked*.

## Paso 3 - Git Reset Hard

El comando *git reset —hard* combinará 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 de la *staging area* y del *working directory* de tal forma que volvemos a los mismos contenidos que estaban presentes en el último commit. Es un comando al que hay que tenerle respeto, porque podemos perder información si no estamos muy seguros de lo que estamos haciendo.

Usar HEAD limpiará el estado hacia lo que había en el último *commit*; si usamos git reset —hard *commit*. Recordemos que HEAD es un alias para el has del último *commit* de la rama.

Pasos:

- Creamos un directorio de trabajo, entramos en él y lo inicializamos.
- Creamos el README.md, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero1, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero2, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero3, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos fichero4 y fichero5, los incluimos en staging pero NO APROBAMOS.
- Lanzamos “ls -l” y “git status”.
- Hacemos un reset hard hacia HEAD-3 y lanzamos “ls -l” y “git status”.
- Debemos estar en un estado sin staging y con la misma “foto” que en HEAD-3.

## Paso 4 - Git Revert

Si ya has aprobado fichero pero te has dado cuenta de que has cometido un error, entonces el comando *git revert* te permite deshacer los cambios que se hayan producido en ese *commit* concreto. El comando creará un nuevo *commit* que tiene la afección invertida del *commit* al que hemos revertido (de modo que si volvemos a él, nos quedamos como estábamos).

Si aún no hemos subido (*pushed*) los cambios, entonces el comando git reset HEAD~1 tiene la misma afección y borrará el último *commit* que ha sido creado por el *revert*.

Usar *git revert* para revertir los cambios del último *commit.*

- Para ello:
- Creamos un directorio de trabajo, entramos en él y lo inicializamos.
- Creamos el README.md, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero1, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero2, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero3, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Deshacemos el initial commit y comprobamos si ha desaparecido algún fichero.

La motivación subyacente de crear nuevos commits es que reescribir la historia en GIT es algo desaconsejado. Si has subido (pushed) tus commits, entonces deberías crear nuevos commits para deshacer los cambios provocados, ya que otros usuarios podrían a ver hecho commits mientras tanto.

## Paso 5 - Git Revert

Para hacer *revert* de múltiples *commits* de una tacada usamos el carácter ~ 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*.

Usamos el comando *git revert HEAD…HEAD~2* para revertir los *commits* hechos entre HEAD y HEAD~2.

Para ello:

- Creamos un directorio de trabajo, entramos en él y lo inicializamos.
- Creamos el README.md, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero1, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero2, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero3, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Ahora, usamos el comando *git revert* tal y como se ha propuesto.

Si en un revert, no tenemos ganas de pasar uno a uno por los cambios del rango seleccionado, se puede hacer así:

```
git revert HEAD...HEAD~2 --no-edit
```

En esta lección aprenderemos cómo podemos compartir los cambios en nuestro repositorio con otras personas, y combinarlos en el nuestro.

Simularemos un trabajo con otros desarrolladores y provocaremos un conflicto. Descubriremos qué opciones tenemos para resolverlo.

**Paso 1 - Git Merge**

El comando *git fetch* descarga los cambios en una rama separada que puede ser comprobada (*checked out*) y fusionada (*merge*). Durante un *merge* git intentará automáticamente combinar los *commits*.

Cuando no existen conflictos, el *merge* será tratado rápidamente y no tendremos que hacer nada. Si sucede un conflicto, entonces recibiremos un error y el repositorio entrará en estado *merge*.

Creamos un directorio que inicializamos.

Creamos un archivo README.md y le agregamos contenido. Este fichero provocará una colisión.

Lo agregamos a staging y hacemos *commit.*

Nos vincularemos al proyecto BAT accesible en https://github.com/sharkdp/bat

Trataremos de lanzar un *merge* y resolver el conflicto.

El comando *git pull* es una combinación de *fetch* y de *merge*.

En lugar de un fetch+merge, se puede usar:

```
git pull origin master –allow-unrelated-histories
```

# Flujos de trabajo

En esta lección aprenderemos cómo podemos crear ramas en nuestro repositorio. Una rama (*branch* ) permite trabajar, de forma efectiva, en un *working directory* completamente nuevo. El resultado es que un único repositorio Git puede tener múltiples versiones diferentes del código base, entre las cuales podemos intercambiarnos sin movernos de directorio.

La rama por defecto en Git se llama *master* . Las ramas adicionales permiten llevar a cabo las mismas operaciones y comandos que podrías efectuar en la rama *master* , como cambios de tipo *committing* , *merging* y *pushing*.

Como ramas adicionales, funcionan de la misma manera que *master* y son ideales para prototipado y experimentos o pruebas que permiten ser fusionadas a la rama *master* si se decide así.

Cuando conmutamos o nos cambiamos de rama, Git cambia los contenidos del *working directory*. Esto es muy llamativo y sorprendente las primeras veces que lo usamos. Usando esta funcionalidad, no necesitamos cambiar configuraciones o parámetros para reflejar que estamos en ramas o ubicaciones diferentes.

## Paso 1 - Git Branch

Las ramas se crean basándonos en otra rama, generalmente *master*. El comando:

```
git branch <new_branch> <starting_branch>
```

toma una rama existente y crea una rama separada para trabajar en ella. Justo en el punto de lanzar el comando, las dos ramas son idénticas. Para cambiar de una rama a otra, usamos el comando:

```
git branch <new_branch>
```

Creamos un nuevo repositorio y lo inicializamos.

Vamos a gitlab, creamos un nuevo proyecto de tipo privado, y lo inicializamos con un README.md automático.

Agregamos el origin de nuestro proyecto gitlab y lo descargamos a nuestro master.

Creamos una nueva rama réplica y la llamamos “new_branch”.

Nos cambiamos a ella.

El comando *git checkout -b* creará y hará *checkout* de la nueva rama creada; es decir, es lo mismo que un *git branch + git checkout*

## Paso 2 – Listar las ramas

Para listar todas las ramas usamos el comando *git branch*.

El argumento adicional *-a* incluirá también las ramas remotas, y el parámetro *-v* incluirá el mensaje *commit* de HEAD de la rama. Recomendable usar ambos siempre.

Usando la carpeta de proyecto del paso anterior, listar todas las ramas con su último mensaje de *commit* lanzando *git branch*

## Paso 3 – Hacer fusión (merge) a master

Supongamos que se ha producido un *commit* a una nueva rama. Para fusionar (*merge*) dentro de la rama *master*, deberíamos primero hacer *chekout* a la rama objetivo (posicionarnos sobre ella), en este caso *master*, y estando en ella, lanzar un *git merge* para fusionar los cambios de la nueva rama sobre la rama *master.*

Ejemplo de llamada:

```
git merge rama_origen rama_destino_del_merge
```

Usando la carpeta de proyecto de nuestro ejercicio anterior, hacemos checkout a new_branch.

Modifico el archivo README.md, lo llevo a staging y lo apruebo.

Ahora, me cambio a master y fusiono.

Compruebo los cambios. Los apruebo.

## Paso 4 - Push Branches

Tal y como hemos visto en pasos anteriores, si queremos subir el contenido de una rama a una ubicación remota, tenemos que usar el comando:

```
git push <remote_name> <branch_name>
```

Ejemplo:

```
$ git push origin master
warning: redirecting to https://gitlab.com/juancarlos.rubio/ej06-branches.git/
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 340 bytes | 340.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
```

```
To https://gitlab.com/juancarlos.rubio/ej06-branches
   14a0d49..31980bc  master -> master
```

Trabajar en una carpeta de proyecto y subir cambios a gitlab. Comprobar que se suben correctamente.

### Paso 5 – Limpiar ramas

Limpiar ramas es importante para borrar ruido y confusión en un proyecto. Para borrar una rama se necesita usar el argumento *-d*. Por ejemplo:

```
git branch -d <branch_name>
```

Ahora que hemos fusionado la rama en *master* en el paso anterior, ya no nos hace falta. Borrémosla para mantener el repositorio limpio y comprensible.

## Para encontrar errores

El comando git diff muestra diferencias entre Working directory y el repositorio. Si tenemos un cambio añadido al area staging, para ver las diferencias tenemos que añadir la opción –cached

Con el comando git log –oneline vemos los commit efectuados. Si además añadimos -p nos mostrará el estado en cada commit, los cambios, y añadiendo -n podemos indicar el núm. de commits hacia atrás. Ejemplo:

git log -p -n2

También podemos ver los cambios concretos por fechas con –since y --until

Para encontrar el autor de alguno de los cambios podemos utilizar git blame, indicando el archivo concreto del que queremos saber los cambios:

git blame src/output.rs

Se puede hacer filtros para concretar, por ejemplo, las líneas 6 y 8 del fichero1

git blame -L 6,8 src/output.rs

## Conflicto de archivos locales con los remotos

Una de las ventajas de pequeños commits es que podemos ser quisquillosos y detallistas acerca que cuáles de esos commits queremos unir / merge, especialmente en proyectos de larga duración (Open Source). Cuando esto sucede, queremos ser capaces de picotear commits individuales y simplemente hacer merge de ellos en la rama principal. Veremos cómo hacerlo.

Ejemplo de consulta de cambios en el remoto:

git log --decorate -n 15 --oneline origin/master

Vamos a provocar una colisión, antes de bajar los documentos, creamos un README que no tiene nada que ver. Añadimos 3 líneas con 3 commit diferentes

Ahora, con el comando cherry-pick nos descargamos una versión concreta:

git cherry-pick origin/master 5d5bf61

Nos dice que hay un error por conflicto con el archivo README. Para más detalle y continuar se le dice:

git cherry-pick –continue

(Para no continuar es el comando: git cherry-pick --abort)

Continuando nos arroja el resultado de que debemos indicar como queremos fusionar los archivos.

Para quedarnos con el README remoto indicamos:

it checkout --theirs .

## Reescribir la historia de un proyecto

Para condesar la historia de los commits en uno porque ya no es relevante todos los detalles, utilizaremos el comando:

git rebase --interactive –root

--interactive hace que aparezca una pantalla donde elegir los commit a condensar

–root desde donde inicia

squash es un condensado con el commit anterior, habría que cambiar cada línia que queramos cambiar

## Etiquetas

Como muchos VCS, Git tiene la posibilidad de etiquetar puntos específicos del historial como importantes. Esta funcionalidad se usa típicamente para marcar versiones de lanzamiento (v1.0, por ejemplo). En esta sección, aprenderás como listar las etiquetas disponibles, como crear nuevas etiquetas cuales son los distintos tipos de etiquetas. En esta lección aprenderemos cómo trabajar con ellas.

### Paso 4 - Listar Tus Etiquetas

Listar las etiquetas disponibles en Git es sencillo. Simplemente escribe git tag:

```
$ git tag
v0.1
v1.3
```

Este comando lista las etiquetas en orden alfabético; el orden en el que aparecen no tiene mayor importancia.

También puedes buscar etiquetas con un patrón particular. El repositorio del código fuente de Git, por ejemplo, contiene más de 500 etiquetas. Si solo te interesa ver la serie 1.8.5, puedes ejecutar:

```
$ git tag -l 'v1.8.5*'
```

```
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

## Paso 5 - Crear Etiquetas

Git utiliza dos tipos principales de etiquetas: ligeras y anotadas. Una etiqueta ligera es muy parecido a una rama que no cambia - simplemente es un puntero a un *commit* especifico.

Sin embargo, las etiquetas anotadas se guardan en la base de datos de Git como objetos enteros.

Tienen un *checksum*; contienen el nombre del etiquetador, correo electrónico y fecha; tienen un mensaje asociado; y pueden ser firmadas y verificadas con *GNU Privacy Guard* (GPG). Normalmente se recomienda que crees etiquetas anotadas, de manera que tengas toda esta información; pero si quieres una etiqueta temporal o por alguna razón no estas interesado en esa información, entonces puedes usar las etiquetas ligeras.

## Paso 6 - Etiquetas Anotadas

Crear una etiqueta anotada en Git es sencillo. La forma más fácil de hacer es especificar la opción -a cuando ejecutas el comando tag:

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

La opción -m especifica el mensaje de la etiqueta, el cual es guardado junto con ella. Si no especificas el mensaje de una etiqueta anotada, Git abrirá el editor de texto para que lo escribas.

Puedes ver la información de la etiqueta junto con el *commit* que está etiquetado al usar el comando git show:

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date: Sat May 3 20:19:12 2014 -0700
my version 1.4
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700
changed the version number
```

El comando muestra la información del etiquetador, la fecha en la que el *commit* fue etiquetado y el mensaje de la etiquetar, antes de mostrar la información del *commit*.

### Paso 7 - Etiquetas Ligeras

La otra forma de etiquetar un *commit* es mediante una etiqueta ligera. Una etiqueta ligera no es más que el *checksum* de un *commit* guardado en un archivo - no incluye más información. Para crear una etiqueta ligera, no pases las opciones -a, -s ni -m:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

Esta vez, si ejecutas git show sobre la etiqueta, no veras la información adicional. El comando solo mostrará el commit:

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700
changed the version number
```

## Moverse adelanta y atrás entre estados de un repositorio

Con el comando:

```
git checkout <hash-id>
```

Entramos en modo desconectado, pero tenemos una foto idéntica a la que teníamos en ese instante del tiempo.

En **subversion** hubiera sido con el comando:

```
svn up -r800
```

(suponiendo la versión 800)

Para volver al HEAD, escribimos:

```
git checkout master
```

o bien

```
git checkout -
```

(un único guion al final del comando).

Movernos al commit inicial y al HEAD y comprobar cómo los ficheros aparecen y desaparecen

**Nota:** git stash guarda el estado que tengamos en ese momento sin aprobar o en staged.

Se recupera con:

```
git stash apply
```

Git stash permite hacer una foto para almacenar de manera temporal lo que tengamos en el area staging