Sistema de control de versiones: GIT

Índice

Control de Versiones

Se llama Control de versiones a la **gestión** de los diversos **cambios** que se realizan sobre los elementos de algún **producto** o una **configuración** del mismo.

Una **versión**, **revisión** o **edición** de un producto, es el estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación.

Sistemas de control de versiones o VCS (**V**ersion **C**ontrol **System**)

Software de Control de Versiones

Son un grupo de aplicaciones originalmente ideadas para gestionar ágilmente los **cambios** en el código fuente de los programas y poder **revertirlos**.

El ámbito se ha ampliado a otras actividades que generan ficheros digitales:

- Documentos
- Ofertas
- Dibujos
- Esquemas
- Etc..

Evolución de los Sistemas de Control de Versiones

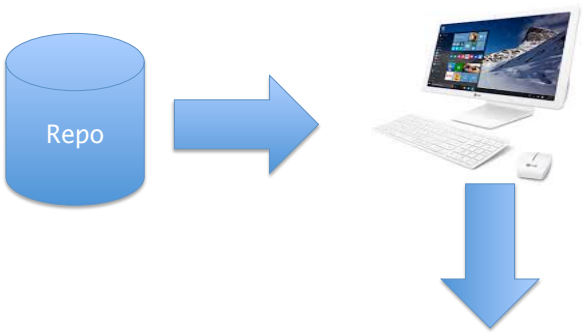
Misma Computadora	Cliente/ Servidor	Distribuidos
Los desarrolladores usaban la misma computadora. La gestión de versiones estaba orientada a ficheros individuales.	Los desarrolladores usan un repositorio central al que acceden mediante un cliente en su máquina.	Cada desarrollador trabaja directamente con su repositorio local, y los cambios se comparten entre repositorios en un paso posterior.
Revision Control System [RCS]. Source Code Control System [SCCS].	Programas de código abierto. Programas propietarios.	Programas de código abierto. Programas propietarios

Características de VCS

- Mecanismo de almacenamiento de los elementos que deba gestionar.
  - Archivos de texto/imágenes/documentación
- Posibilidad de realizar cambios sobre los elementos almacenados.
  - Modificaciones parciales
  - Añadir/Borrar/Renombrar/Mover elementos.
- Registro histórico de las acciones realizadas con cada elemento o conjunto de elementos pudiendo volver o extraer un estado anterior del producto.

Formas de colaborar

El primer paso en el proceso de colaboración usando un VCS es crearse una copia local obteniendo dicha información de un repositorio.



En su copia local el colaborador puede modificar los ficheros copiados, añadir, eliminar, renombrar, mover, etc..

Formas de colaborar

Después de que tengamos preparada una actualización local si deseamos poner la a disposición de otros colaboradores tenemos dos formas de hacerlo:

De forma **exclusiva**:

1. Comunico al Repositorio que elemento quiero modificar.
2. El sistema se encarga de impedir que otro colaborador lo modifique.
3. Una vez hecha la modificación, esta se comparte con el resto de colaboradores.
4. Al terminar de modificar, se libera.

Ejemplos:
- Sourface
- Subversión dispone de mecanismos para usarlo

Formas de colaborar

Después de que tengamos preparada una actualización local si deseamos poner la a disposición de otros colaboradores tenemos dos formas de hacerlo:

De forma **colaborativa**:

1. Cada usuario modifica su copia local.
2. Cuando el usuario desea compartir sus cambios avisa al sistema
3. El sistema automáticamente combinará las diversas modificaciones.
4. Pueden aparecer conflictos o inconsistencias que deberán ser resueltos en algunos casos manualmente

Ejemplos:
- Subversion
- Git

El control de versiones Team Foundation Server permite escoger cualquiera de las dos formas de colaboración

Arquitecturas de almacenamiento de los repositorios

Centralizados

Existe un repositorio centralizado de todo el código, del cual es responsable un único usuario (o conjunto de ellos). Se facilitan las tareas administrativas a cambio de reducir flexibilidad, pues todas las decisiones fuertes (como crear una nueva rama) necesitan la aprobación del responsable.

Ejemplos son CVS, Subversion o Team Foundation Server.

Distribuidas:

Cada usuario tiene su propio repositorio. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Es frecuente el uso de un repositorio que sirve de punto de sincronización de los distintos repositorios locales.

Ejemplos: Git y Mercurial.

Ventajas de los repositorios centralizados

En los sistemas centralizados hay más control a la hora de trabajar en equipo ya que tienen una versión centralizada de todo lo que se está haciendo en el proyecto.

En los sistemas centralizados las versiones vienen identificadas por un número de versión. Sin embargo en los sistemas de control de versiones distribuidos no hay números de versión, ya que cada repositorio tendría sus propios números de revisión dependiendo de los cambios. En lugar de eso cada versión tiene un identificador al que se le puede asociar una etiqueta (tag).

Ventajas de los repositorios distribuidos

- Necesita menos veces estar conectado a la red para hacer operaciones. Esto produce una mayor autonomía y una mayor rapidez.
- Aunque se caiga el repositorio remoto la gente puede seguir trabajando.
- Al hacer los distintos repositorio una réplica local de la información de los repositorios remotos a los que se conectan, la información está muy replicada y por tanto el sistema tiene menos problemas en recuperarse.
- Permite mantener repositorios centrales más limpios.
- El servidor remoto requiere menos recursos.

Ventajas de los repositorios distribuidos

- Al ser los sistemas distribuidos más recientes que los sistemas centralizados, y al tener más flexibilidad por tener un repositorio local y otro/s remotos, estos sistemas han sido diseñados para hacer fácil el uso de ramas (creación, evolución y fusión) y poder aprovechar al máximo su potencial.

Terminología

Repositorio	El repositorio es el lugar en el que se almacenan los datos actualizados e históricos de cambios, a menudo en un servidor. Puede ser un sistema de archivos en un disco duro, una base de datos, etc..
Revisión (*"version"*)	Una revisión es una versión determinada de la información que se gestiona.
Tag	Darle a alguna versión de cada uno de los ficheros del proyecto en desarrollo en un momento preciso un nombre común ("etiqueta" o "rótulo") para asegurarse de reencontrar ese estado de desarrollo posteriormente bajo ese nombre.

Terminología

Línea base (*"baseline"*)	Una revisión aprobada de un documento o fichero fuente, a partir del cual se pueden realizar cambios subsiguientes.
Abrir rama (*"branch"*) o ramificar	Un proyecto puede ser *branched* o bifurcado en un instante de tiempo de forma que, desde ese momento, se tienen dos copias (ramas) que evolucionan de forma independiente siguiendo su propia línea de desarrollo.

Terminología

Desplegar (*"Check-out"*, *"checkout"*, "co")	Un despliegue crea una copia local desde el repositorio. Se puede especificar una revisión concreta, y predeterminada, aunque se suele obtener la última versión.
"Publicar" o "Enviar" (*"commit"*, *"check-in"*, "ci", *"install"*, *"submit"*)	Un *commit* sucede cuando los cambios hechos en una copia local son escritos o integrados sobre el repositorio.

Terminología

Conflicto Un conflicto ocurre cuando el sistema no puede manejar adecuadamente cambios realizados por dos o más usuarios en un mismo archivo.

Resolver El acto de la intervención del usuario para atender un conflicto entre diferentes cambios al mismo archivo.

Terminología

Cambio (*"change"*, *"diff"*, *"delta"*) Un cambio representa una modificación específica a un archivo bajo un sistema de control de versiones.

Lista de cambios (*"changelist"*, *"change set"*, *"patch"*) En muchos sistemas de control de versiones con *commits* multi-cambio atómicos, una lista de cambios identifica el conjunto de cambios hechos en un único *commit*.

Terminología

Integración o fusión (*"merge"*)	Una integración o fusión une dos conjuntos de cambios sobre un fichero o un conjunto de ficheros en una revisión unificada de dicho fichero o ficheros.
Integración inversa	El proceso de fundir ramas de diferentes equipos en el *trunk* principal del sistema de versiones.
Actualización (*"sync"* ó *"update"*)	Una actualización integra los cambios que han sido hechos en el repositorio (por ejemplo por otras personas) en la copia de trabajo local.

Terminología

Copia de trabajo (*"workspace"*)	La copia de trabajo es la copia local de los ficheros de un repositorio, en un momento del tiempo o revisión específicos. Todo el trabajo realizado sobre los ficheros en un repositorio se realiza inicialmente sobre una copia de trabajo, de ahí su nombre. Conceptualmente, es un cajón de arena o *sandbox*.
Congelar	Significa solo permitir cambios (*commit's*) para solucionar un fallo de una entrega (*release*) y, por lo tanto, suspender cualquier otro cambio antes de liberar una entrega (release), con el fin de obtener una versión consistente del proyecto o producto.

Qué es Git

Es un software de control de versiones diseñado por **Linus Torvalds**, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.

Hay muchos proyectos software que utilizan Git entre ellos el grupo de programación del núcleo de Linux.

Git: Características

1. Fuerte apoyo al desarrollo no lineal, por ende rapidez en la gestión de ramas y mezclado de diferentes versiones. Una presunción fundamental en Git es que un cambio será fusionado mucho más frecuentemente de lo que se escribe originalmente, conforme se pasa entre varios programadores que lo revisan.

2. Gestión distribuida. Al igual que Darcs, BitKeeper, Mercurial, SVK, Bazaar y Monotone, Git le da a cada programador una copia local del historial del desarrollo entero, y los cambios se propagan entre los repositorios locales. Los cambios se importan como ramas adicionales y pueden ser fusionados de la misma manera que se hace con la rama local.

Git: Características

3. Los almacenes de información pueden publicarse por HTTP, FTP, rsync o mediante un protocolo nativo, ya sea a través de una conexión TCP/IP simple o a través de cifrado SSH. Git también puede emular servidores CVS, lo que habilita el uso de clientes CVS pre-existentes y módulos IDE para CVS pre-existentes en el acceso de repositorios Git.

4. Los repositorios Subversion y svk se pueden usar directamente con git-svn.

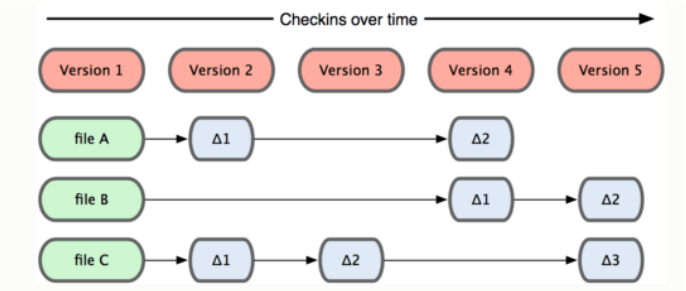
5. Gestión eficiente de proyectos grandes, dada la rapidez de gestión de diferencias entre archivos, entre otras mejoras de optimización de velocidad de ejecución.

Git: Características

Snapshot – no diferencias

La principal diferencia entre Git y cualquier otro VCS (Subversion y compañía incluidos) es cómo Git modela sus datos.

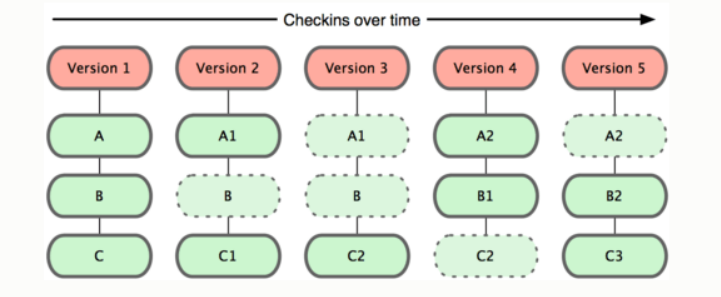
Los otros sistemas como listas de cambios:



Git: Características

Snapshot – no diferencias

Git almacena la información como instantáneas del proyecto a lo largo del tiempo básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Si los archivos no se han modificado, Git no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado.



Git: Características

Integridad.

Todo en Git es verificado mediante checksum. Un código hash SHA-1 (40 caracteres hexadecimales).

Esta funcionalidad está integrada en Git al más bajo nivel y es parte integral de su filosofía. No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git lo detecte.

Aspecto del código hash de git:

24b9da6552252987aa493b52f8696cd6d3b00373

Git: Características

Git generalmente sólo añade información

La mayoría de las acciones en Git solo añaden información BB.DD.

Es muy difícil que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información.

Como en cualquier VCS, puedes perder o estropear cambios que no has confirmado todavía; pero después de confirmar una Snapshot es muy difícil de perder.

Con podemos experimentar sin peligro de fastidiar gravemente las cosas.

Git: Características

Estados en Git:

1. confirmado (committed)
2. modificado (modified)
3. y preparado (staged).

Confirmado: los datos están almacenados de manera segura en la base de datos local (Repo).

Modificado: Se ha modificado el archivo pero todavía no está confirmado en la base de datos local (Repo).

Preparado: Se ha marcado un archivo modificado, en su versión actual, para que vaya en la próxima confirmación.

Git: Características

Secciones de un proyecto Git:

1. **El directorio de Git (Git directory) o repositorio**
2. El directorio de trabajo (working directory)
3. El área de preparación (staging area).

El directorio de Git: es donde Git almacena los metadatos y la base de datos de objetos del proyecto.

Es la parte más importante de Git y es lo que se copia cuando clonas un repositorio desde otro ordenador.

Git: Características

Secciones de un proyecto Git:

1. El directorio de Git (Git directory) o repositorio
2. **El directorio de trabajo (working directory)**
3. El área de preparación (staging area).

El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida del directorio de Git, y se colocan en disco local para que se puedan usar o modificar.

Git: Características

Secciones de un proyecto Git:

1. El directorio de Git (Git directory) o repositorio
2. El directorio de trabajo (working directory)
3. **El área de preparación (staging area).**

El área de preparación es un archivo, contenido en el directorio de Git, que almacena información acerca de lo que va a ir en la próxima confirmación.

A veces se le denomina índice, pero se está convirtiendo en un estándar referirse a ella como el área de preparación.

Staging area

Git requiere que los cambios sean marcados explícitamente para el próximo commit.

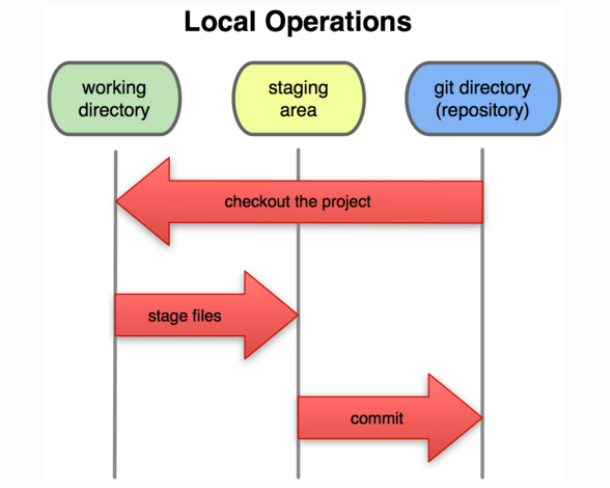
Proceso:

1.- Modificamos el fichero
2.- Añadir al staging area
3.- Añadir al repositorio

En el staging area tendremos una imagen completa de los cambios.

Los nuevos archivos siempre deben ser explícitamente añadidos al staging area.

Flujo de cambio de estado



Comandos: Creación de un repositorio Git

$ git init

```
$ git init
$ git init <directorio>
$ git init --bare <directorio>
```

$ git clone

$ git config

Comandos Git: git init

Inicializa el repositorio local.

Se ejecuta una sola vez.
Hemos de ejecutar éste comando antes de cualquier otro.

Desde un directorio existente:

Abrir un terminal
Posicionarnos en el directorio en el que deseamos crear el repositorio.
Ejecutar el comando `$ git init`

Crea un subdirectorio /<nombre del directorio>/.git
Crea el esqueleto de fichero necesarios en el repositorio

Comandos Git: git init

Otros usos del comando `git init`

`$ git init` <nombre del directorio>

1. Crea el directorio <nombre del directorio> vacío
2. Crea un subdirectorio /<nombre del directorio>/.git
3. Crea el esqueleto de fichero necesarios en el repositorio.

Comandos Git: git init

Otros usos del comando git init

```
$ git init --bare <directory>
```

- Inicializa un directorio vacío pero omite el directorio de trabajo.
- El directorio compartido deberá ser creado siempre con el flag –bare.
- Existe una convención: los directorios creados con el flag --bare deberán finalizar con la terminación **".git**",

Ejemplo:

```
$ git init –bare miPrimerProyecto.git
```

Comandos Git: git init

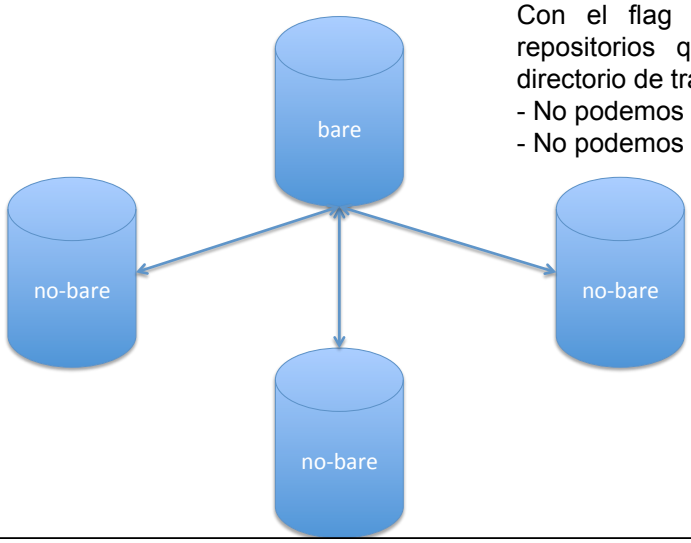
En la mayoría de los proyectos:

`$ git init` sólo necesita ser ejecutado una vez para crear un repositorio central

Los desarrolladores normalmente no usan **`git init`** para crear sus repositorios locales. En su lugar utilizan **`git clone`** para copiar un repositorio existente en su máquina local.

Comandos Git: git init

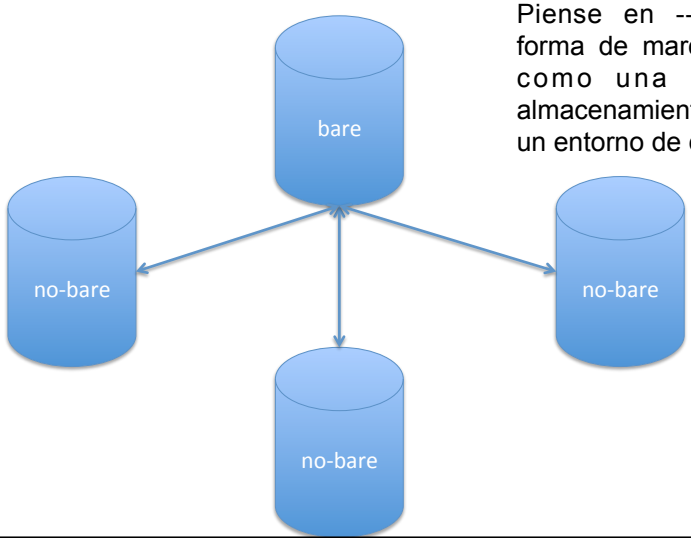
Repositorios bare [Compartidos]



Con el flag "--bare" creamos repositorios que no tienen el directorio de trabajo.
- No podemos editar ficheros.
- No podemos hacer commit.

Comandos Git: git init

Repositorios bare [Compartidos]



Piense en --bare como una forma de marcar un repositorio como una instalación de almacenamiento, en oposición a un entorno de desarrollo.

Comandos Git: git clone

El comando **git clone** genera un repositorio local a partir repositorio git existente.

Ventaja:

La clonación crea automáticamente una conexión remota llamada origen que señala de nuevo al repositorio original. Esto hace que sea muy fácil de interactuar con un repositorio central.

Uso:

$ git clone <repo>

<repo>puede estar localizado en un sistema de ficheros o sobre una máquina remota accesible vía SSH o vía HTTP

Comandos Git: git clone

<repo>puede estar localizado en un sistema de ficheros o sobre una máquina remota accesible vía SSH o vía HTTP

Ejemplo vía SSH:

```
$git clone git@HOSTNAME:USERNAME/REPONAME.git
```

Ejemplo vía HTTP:

```
$ git clone https://ejrbalma@bitbucket.org/
ejrbalma/prueba.git
```

Comandos Git: git clone

Otro uso del comando

`$ git clone` <repo> <nombreDirectorio>

Clona el repositorio localizado en <repo> en el directorio <nombreDirectorio> en la máquina local.

Comandos Git: git clone

Si un proyecto ya está en un repositorio central, el comando `git clone` es la forma más común que los usuarios obtengan una copia de desarrollo.

El comando `git clone` , al igual que `git init`, se ejecuta una sola vez.

De una única operación el desarrollador ha obtenido una copia de trabajo del repositorio.

Todas las operaciones de control de versiones y colaboraciones se gestionan a través de su repositorio local.

Comandos Git: git config

La configuración sólo se hace una vez aunque se puede cambiar cuantas veces se necesite.

`$git config` permite obtener y establecer variables de configuración, que controlan el aspecto y funcionamiento de Git.

Estas variables pueden almacenarse en tres ubicaciones distintas:

1. /etc/gitconfig: **`git config --system`** . Almacenamos los valores para todos los usuarios del sistema y todos sus repositorios.
2. ~/.gitconfig: Específico a tu usuario **`git config --global`**
3. .git/config: ***`git config`*** del repositorio actual.

Cada nivel sobrescribe los valores del nivel anterior : .git/config tienen preferencia sobre /etc/gitconfig.

Comandos Git: git config

Configuración de la identidad:

```
$ git config --global user.name "Enrique Ramon"
$ git config --global user.email ejrbalma@gmail.com
```

Configurando el editor:

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

Configurando tu herramienta de diferencias:

```
$ git config --global merge.tool vimdiff
```

Comandos Git: git config

Comprobando la configuración:

```
$ git config –l o [--list]
```

Puede que veas claves repetidas, porque Git lee la misma clave de distintos archivos (/etc/gitconfig y ~/.gitconfig, por ejemplo). En ese caso, Git usa el último valor para cada clave única que ve.

Comprobando la configuración de una clave específica:

```
$ git config user.name
```

Comandos Git: git config

Configuración editor. Ejemplos:

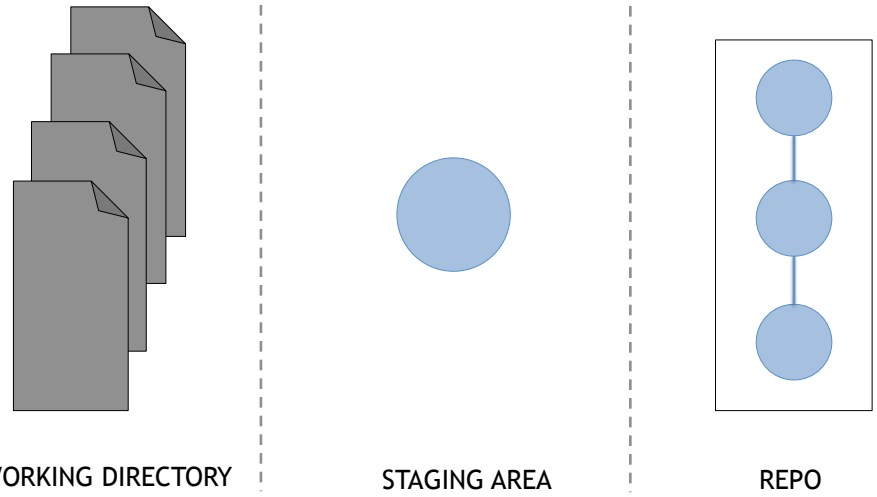
Windows 32 bits: Sublime :

```
git config --local core.editor "'c:/program files (x86)/sublime text 3/sublimetext.exe' –w”
```

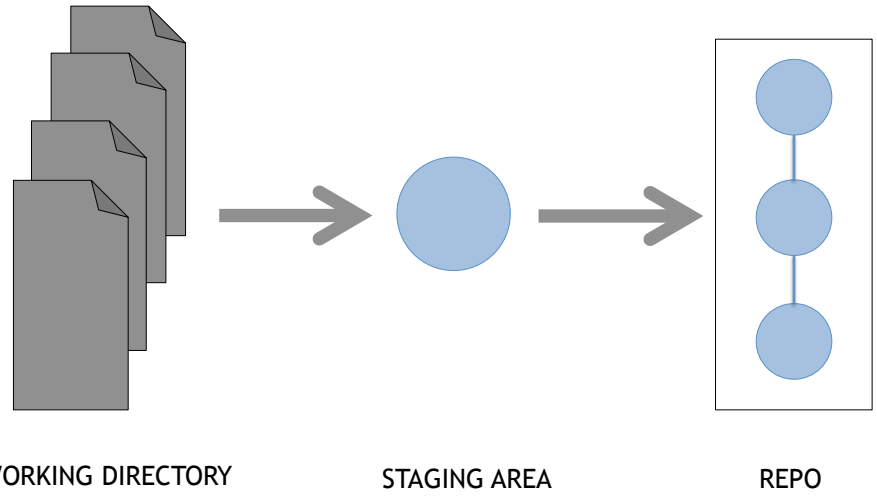
Windows 64 bits: Sublime

```
git config --local core.editor "'c:/program files/sublime text 3/sublimetext.exe' –w”

git config --global core.editor "vim"
```

Estructura de un repositorio Git



Estructura de un repositorio Git



Staging área – Área de preparación

El área de preparación es una de las características más singulares de Git.

Ayuda a pensar en ella como una cache entre el directorio de trabajo y la historia del proyecto.

En lugar de hacer commit de todos los cambios que se hayan realizado desde la última confirmación, la staging área, gestiona qué cambios se incluyen en la próximo instantánea del repositorio.

Como en cualquier sistema de control de versiones, es importante crear **commit atómicos** para que sea fácil localizar los problema y revertir los cambios con un impacto mínimo en el resto del proyecto.

El Repositorio. Archivos

En el directorio de trabajo los archivos:

- bajo seguimiento (tracked)
- sin seguimiento (untracked).

Los archivos bajo seguimiento: son aquellos que existían en la última instantánea;

- sin modificaciones
- modificados
- o preparados.

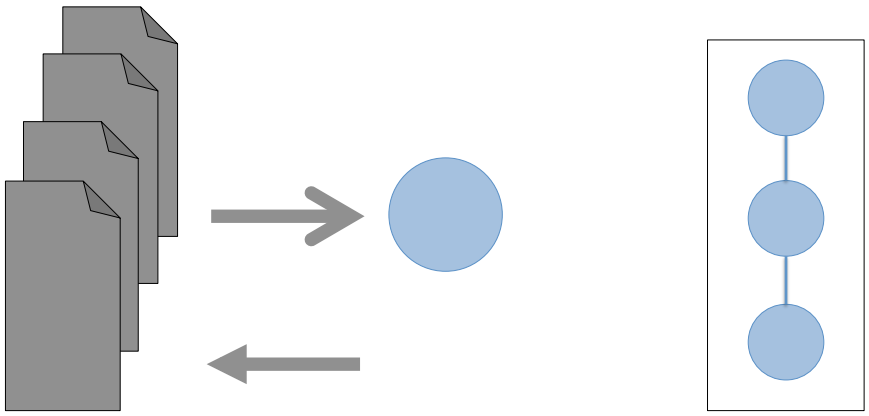
Los archivos sin seguimiento son todos los demás :

- Archivos del directorio que no están en el último snapshot
- Archivos que no están en staging área.

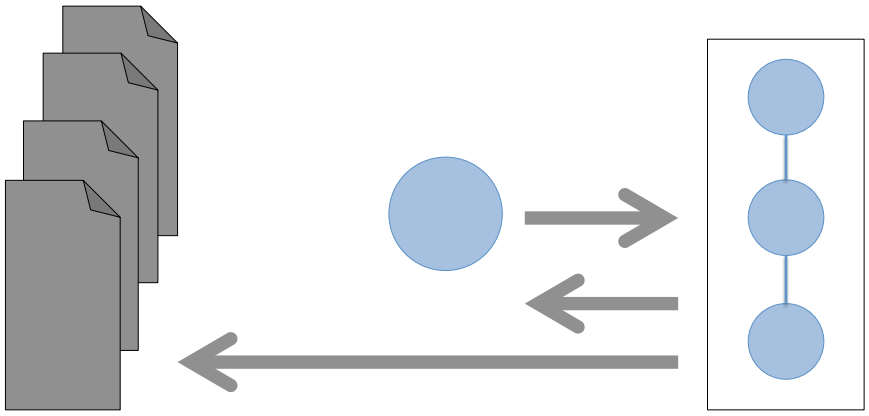
Cuando clonamos la primera vez:

- ✓ Todo los archivos están bajo seguimiento.
- ✓ Sin modificaciones

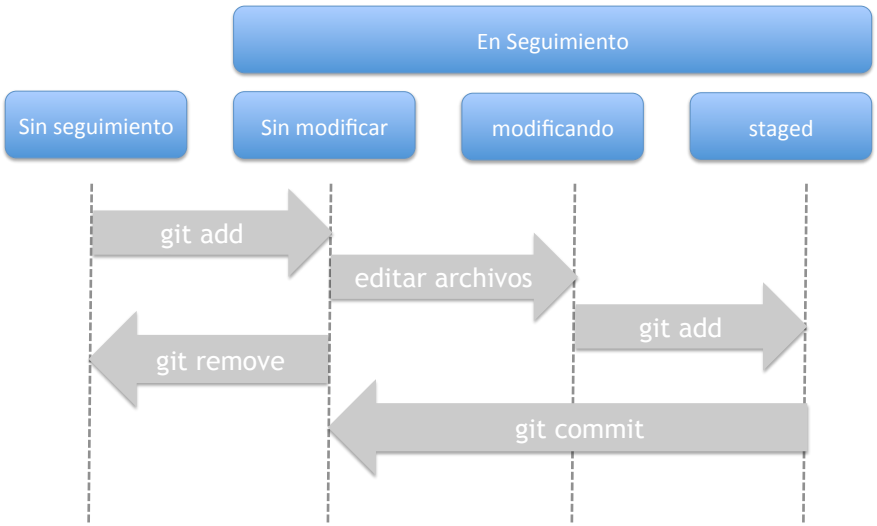
Staging area – Área de preparación



Staging área – Área de preparación



Flujo de cambios de estado de los archivos



El Repositorio: Comandos mínimo para trabajar con Git

Los comandos imprescindibles para implementar el workflow con git son:

- git add
- git commit
- git status

Son los comandos que dispone git para asegurar versiones del proyecto en el repositorio.

El desarrollo de un proyecto gira en torno al patrón de comandos:

edit/ stage /commit

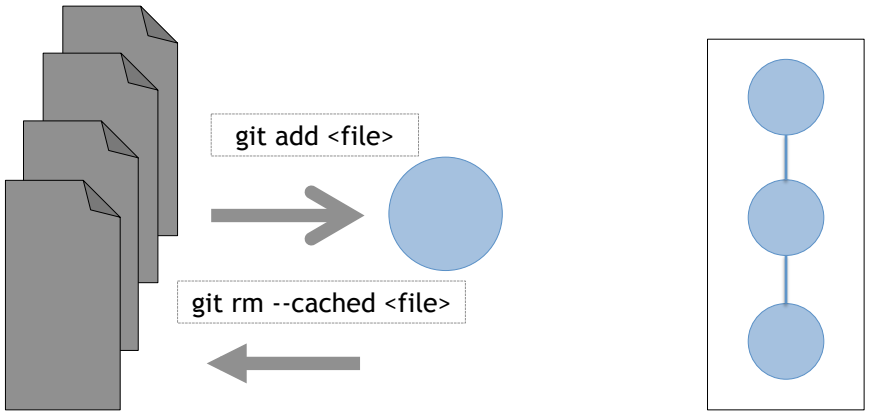
Comandos Git: git add

Este comando añade un archivo modificado en el directorio de trabajo al staging área.

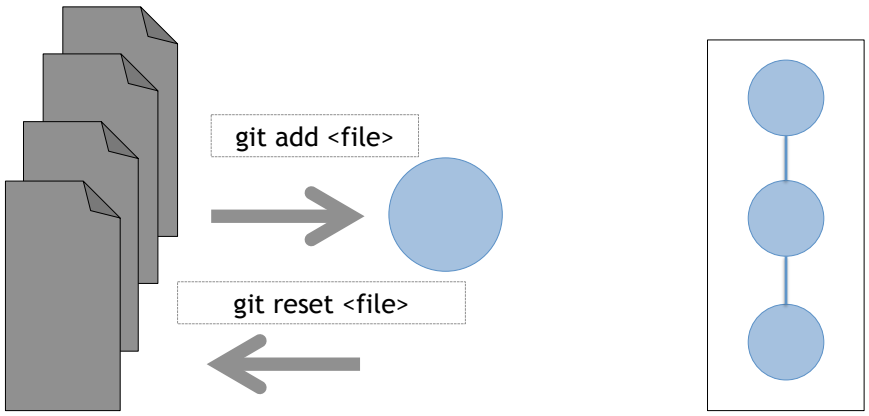
Le dice a Git que desea incluir cambios de un archivo en el próximo “git commit”.

Sin embargo, git add no afecta al repositorio.

Staging área – Área de preparación



Staging área – Área de preparación



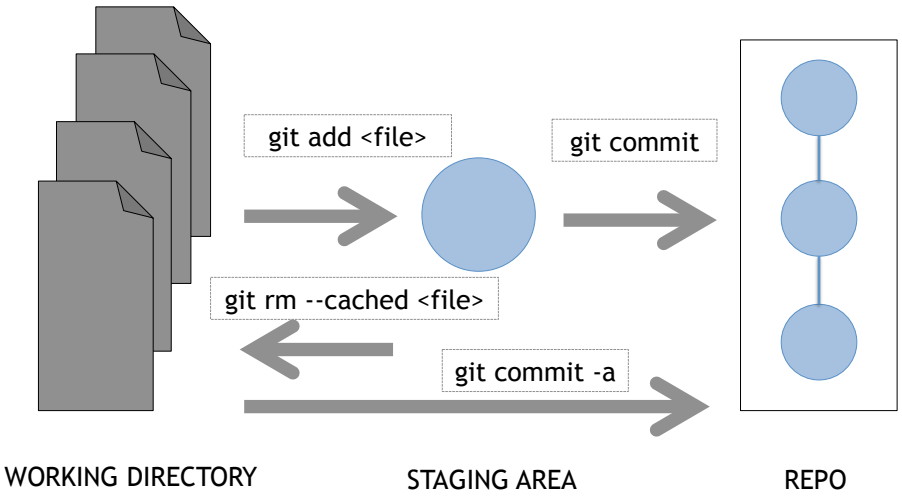
Comandos Git: git add

Usos del comando

```
git add <file>

git add <directory>

git add –p
```

Iniciar una sesión interactiva que permite seleccionar partes de un archivo que se incluirán en la próxima confirmación.
Se presentará con un trozo de cambios y le pregunta si se incluye o no. Utilizar “Y” para añadir el trozo, “n” para ignorar el trozo, “s” dividirlo en trozos más pequeños, “e” para editar manualmente el trozo, y “q” para salir.

Staging área – Área de preparación



Comandos Git: git commit

Este comando realiza un **commit** de un snapshot del staging área a la historia del proyecto (repositorio).

Los diferentes snapshot que han sido realizados son considerados versiones "seguras" de un proyecto en el repositorio Git que nunca va a cambiar a menos que explícitamente alguien lo haga.

```
$ git add
$ git commit
```

Son los comandos imprescindibles para realizar el workflow en un repositorio Git.

Comandos Git: git commit

Usos:

```
$ git commit
```

Abre un editor de texto. Hemos de incluir un mensaje de confirmación. Después de introducir un mensaje, guardar el archivo y cierre el editor.

```
$ git commit –m “Mensaje”
```

```
$ git commit –a –m “Mensaje”
```

Se salta el área de preparación (staging área). Incluye en el snapshot todos los archivos que estuvieran en seguimiento alguna vez.

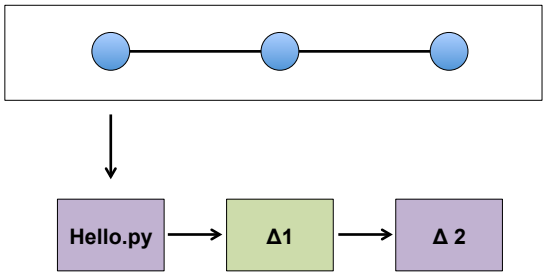
Índice- Test comandos: add/status/commit

- Hacer un git status después del commit.
  - Que no hay ficheros para realizar el commit.
  - Que el directorio de trabajo de git está limpio. Luego el commit lo que hace es eliminar del directorio de trabajo los ficheros que acaba de estar en el repositorio
- Repetimos el proceso:
  - Modificamos un fichero
  - Añadimos un fichero
  - Ejecutamos el git status.
    - Indica que un fichero, al que seguimos se ha marcado.
    - Indica que aparece un nuevo fichero que sin seguimiento
    - Ejecutamos git add <file> y lo incluimos en el staging area
    - Ejecutamos git status. Está pendiente uno de los ficheros.
- git commit –amend ver este caso
  - git commit -m 'initial commit'
  - $ git add forgotten_file
  - $ git commit --amend

Git: Snapshot versus Diferencias

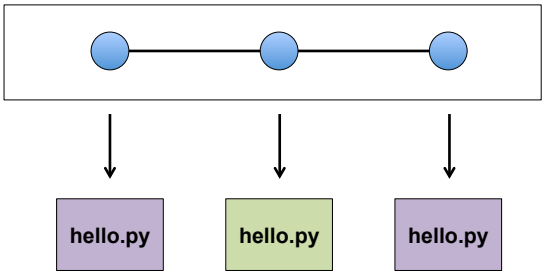
Otra diferencia entre Git y cualquier otro VCS (Subversion) es cómo Git modela sus datos.

Los otros sistemas gestionan sus datos como listas de cambios:



Git: Snapshot versus Diferencias

Git almacena la información como instantáneas del proyecto a lo largo del tiempo, básicamente, hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Si los archivos no se han modificado, Git no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado.



Inspeccionar un repositorio Git y Staging Area

Dos comando:

1. `$ git status`
2. `$ git log`

Comandos Git: git status

Objetivo:

Principal herramienta para determinar el estado en el que se encuentra cada uno de los archivos.

Básicamente muestra lo que se ha hecho con `git add` y `git commit`.

Los mensajes de estado también incluyen instrucciones pertinentes para archivos de staging / unstaging.

Comandos Git: git status

Uso:

```
$ git status
```

Listas los ficheros:

- Incluidos en staging area.
- Los que no se han incluido.
- Los ficheros que no son seguidos.

Comandos Git: git status

Si ejecutas este comando justo después de clonar un repositorio, se debería ver algo así:

```
$ git status
# On branch master
 nothing to commit, working directory clean
```

Commandos Git: git status

Otro ejemplo de ejecución, después de modificar alguno ficheros.

```
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#modified: file.txt
#
# Changes not staged for commit:
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in
working directory)
#
#modified: otrofiles.txt
#
# Untracked files:
# (use "git add <file>..." to include in what will be
committed)
#
#file.exe
```

Comandos Git: git status

Es una buena práctica comprobar el estado de su repositorio antes de confirmar los cambios para que accidentalmente no se haga un commit de algo de lo que no se quiere hacer.

Ejemplo:

```
# Edit hello.py
$ git status
# hello.py is listed under "Changes not staged for
commit"
$ git add hello.py
$ git status
# hello.py is listed under "Changes to be committed"
$ git commit
$ git status
# nothing to commit (working directory clean)
```

Comandos Git: git log

El git log visualiza la información de los commit que se han realizado.

Permite listar, filtrar y buscar cambios específicos en la historia del proyecto.

Este comando solo nos permite consulta la historia de los commit realizados.

Comandos Git: git log

Uso del comando:

- `git log`
- `git log - <limit> (limita el número de commits visualizados)`
- `git log –oneline (display una línea por commit)`
- `git log stat (da información ampliada, ficheros que se han incluido, qué se ha modificado…)`

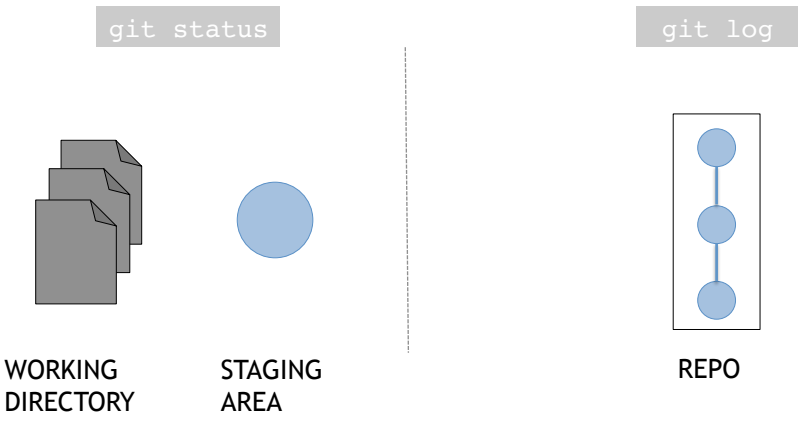
Comandos Git: git log

Uso del comando:

- `git log –p` <da la información más detallada posible de cada commit dando las diferencias con versiones anteriores.>
- `git log –author = “”`
- `git log –grep “ “`
- `git log <since>..<until>`

Comando Git: git log

Lista las snapshot que se han hecho, es decir, la historia del proyecto, se podrá filtrar, y realizar búsquedas por cambios específicos.



Deshacer cambios realizado

Git pone a nuestras disposición un conjunto de comandos para trabajar con las versiones anteriores de un proyecto de software.

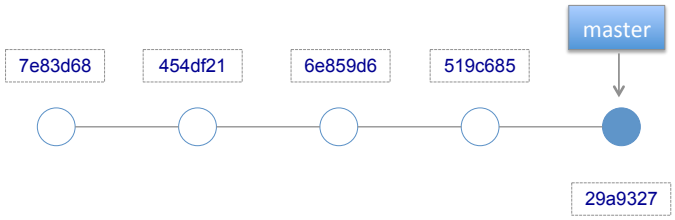
- `git checkout`
- `git revert`
- `git reset`
- `git clean`

Comandos Git: git checkout

Se usa para ver un estado anterior de un proyecto sin alterar el estado actual. Git checkout de un archivo permite ver una versión antigua de ese archivo, dejando el resto de su directorio de trabajo sin tocar.

El comando git checkout tiene tres funciones:

checkout files
checkout commit
checkout branches [siguiente clase].

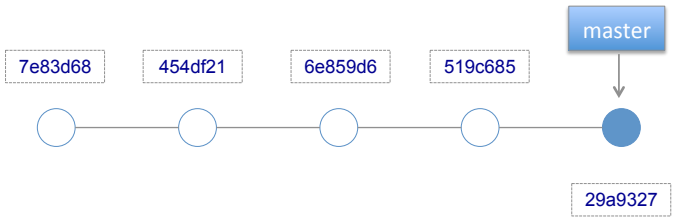


Comandos Git: git checkout

El carácter ~ se utiliza como una referencia relativa. Con respecto a una referencia de git, por ejemplo HEAD. (que marca la posición actual del repositorio).

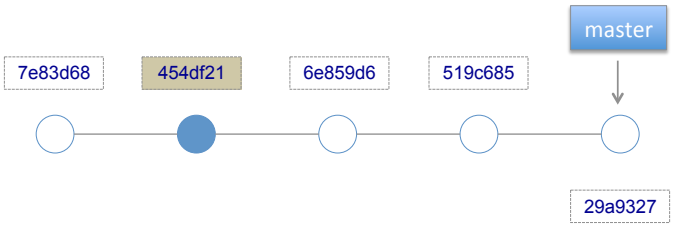
El comando git checkout tiene tres funciones:

- checkout files
- checkout commit
- checkout branches [siguiente clase] .

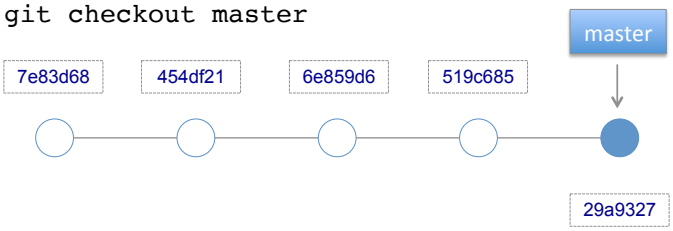


Comandos Git: git checkout

```
git checkout  45df21
```

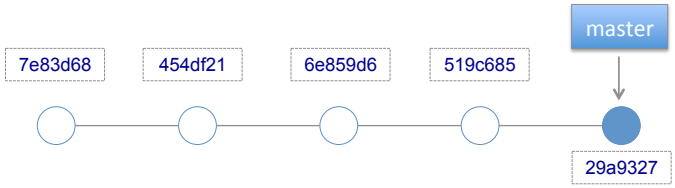



```
git checkout master
```



Comandos Git: git checkout

```
git checkout 45df21 nombreFichero.txt
```



El fichero vuelve a la versión previa al commit 454df21.
Se añade en el staging area.
HEAD sigue apuntando a master

¿ Qué hacer si simplemente queremos volver a master:

```
git checkout HEAD nombreFichero.txt
git checkout 29a9327 nombreFichero.txt
```

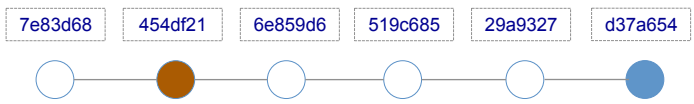
¿ Qué hacer si queremos modificar esa versión de fichero?
Como sea que lo tenemos en el staging área solo hay que hacer un nuevo commit.

Comando Git: git revert

Este comando deshace los commit

No borra el commit, lo que hace es añadir un nuevo commit con los cambios que han sido revertidos.


```
git revert 454df21
```



No elimina la historia del proyecto, se crea un nuevo commit, y mantenemos la historia anterior del proyecto.
Es una forma segura de deshacer cambios.

Comando Git: git reset

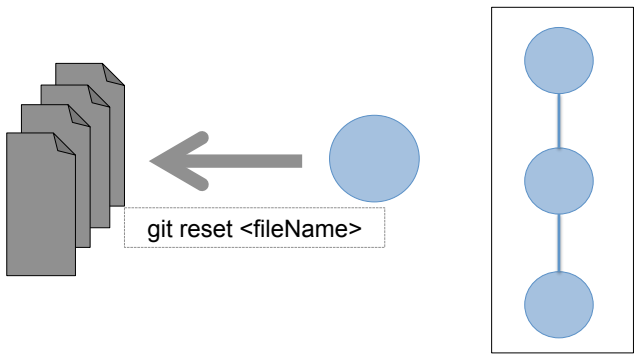
```
$ git reset <file>

$ git reset

$ git reset  --hard

$ git reset <commit>

$ git reset --hard <commit>
```

Comando Git: git reset

Versión: `git reset fileName`

Este comando deshace add de un archivo del staging area

Elimina un fichero del staging area, el efecto contrario `git add`.
El directorio de trabajo lo deja igual.

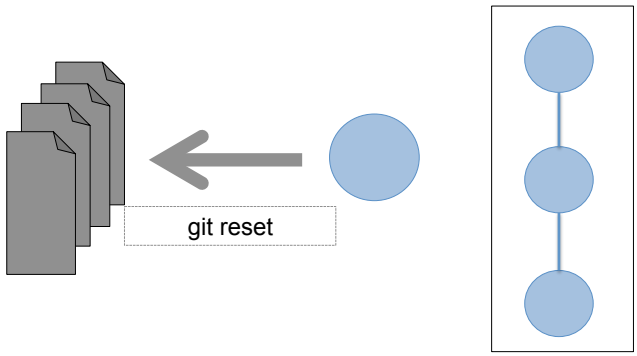


Comando Git: git reset

Versión: `git reset`

Este comando deshace todos add del staging area

Elimina todos los fichero del staging área, el efecto contrario `git add`. El directorio de trabajo lo deja igual. El staging area se queda en el mismo estado en el que se encontraba después del último commit.



Comando Git: git reset

Versión: `git reset --hard`

Este comando repone el directorio de trabajo y staging area al último commit.

Elimina todos ficheros del staging área y deja los archivos del directorio de trabajo con la versión del último commit.

```
HEAD is now at 29a9327 "mensage"
```

Borra todos los cambios sin confirmar, así que, asegúrese de que realmente quiere deshacerse de sus desarrollos locales antes de usarlo.

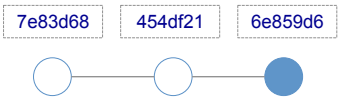
Comando Git: git reset

Versión: `git reset <commit>`

Este comando deshace los commit

Mueve el HEAD al commit. Re-establece el staging área para que coincida el estado de los archivos antes de git add. Y no modifica el directorio de trabajo. Esta situación permite reconstruir la historia usando commit más atómicos, más claros o deshacer cambios en archivos.

Después del git reset, el directorio de trabajo no se modifica.

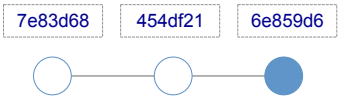


Comando Git: git reset

Versión: `git reset --hard 6e859d6`

Este comando deshace los commit.

Todos los cambios hechos desde 6e859d6 se eliminan del directorio de trabajo incluido lo que se haya modificado y todavía esté sin confirmar, pero solo de los archivos en seguimiento.



Comando Git: git reset

El flag --hard es útil cuando un experimento ha salido mal y queremos empezar desde cero.

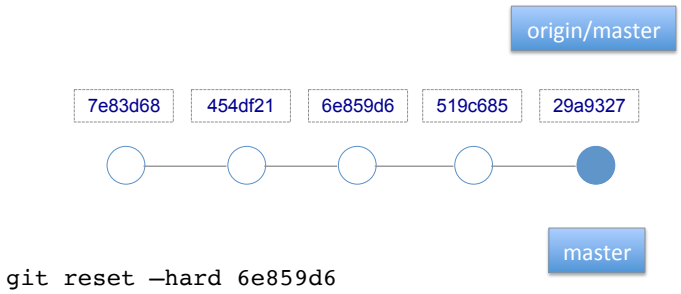
- git revert está diseñado para deshacer cambios que han sido publicados.
- git reset está diseñado para deshacer cambios no publicados.

Nunca debemos de usar git reset <commit> sobre un commit que ya ha sido publicado. Si ya lo hemos hecho público debemos de suponer que otros usuarios ya han actualizado sus repositorios.

Comando Git: git reset

Eliminar un commit que otro miembro del equipo de desarrollo ha seguido desarrollando tiene serios problemas para la colaboración. Cuando intentan sincronizar con su repositorio, se verá como un trozo de la historia del proyecto ha desaparecido abruptamente.

La secuencia siguiente muestra lo que sucede cuando se hace un git reset <commit> de un <commit> que se publico. La rama origin / master es la versión del repositorio central de su rama principal local.

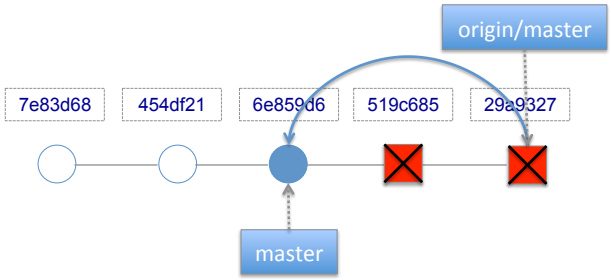



```
git reset –hard 6e859d6
```

Comando Git: git reset

Eliminar un commit que otro miembro del equipo de desarrollo ha seguido desarrollando tiene serios problemas para la colaboración. Cuando intentan sincronizar con su repositorio, se verá como un trozo de la historia del proyecto ha desaparecido abruptamente.

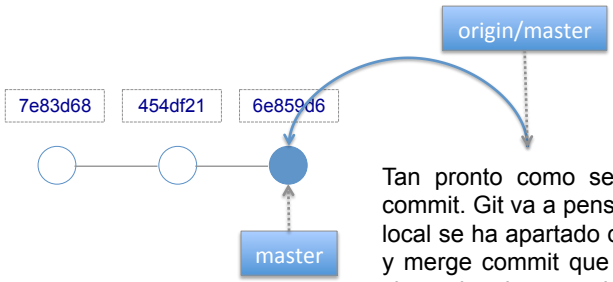
La secuencia siguiente muestra lo que sucede cuando se hace un git reset <commit> de un <commit> que se publicó. La rama origin / master es la versión del repositorio central de su rama principal local.



Comando Git: git reset

Eliminar un commit que otro miembro del equipo de desarrollo ha seguido desarrollando tiene serios problemas para la colaboración. Cuando intentan sincronizar con su repositorio, se verá como un trozo de la historia del proyecto ha desaparecido abruptamente.

La secuencia siguiente muestra lo que sucede cuando se hace un `git reset <commit>` de un `<commit>` que se publicó. La rama origin / master es la versión del repositorio central de la rama principal local.



Tan pronto como se añaden nuevos commit. Git va a pensar que su historia local se ha apartado de origin / master, y merge commit que se requiere para sincronizar los repositorios es probable que confunda y frustre el equipo de desarrollo.

Comando Git: git clean

Elimina archivos sin seguimiento del directorio de trabajo. Este comando no se puede deshacer.

Este comando se utiliza complementando a `git reset --hard` (elimina del directorio de trabajo los archivos a los que se les hace seguimiento) git clean elimina a los que no se les hace seguimiento.

Combinados, estos dos comandos permiten que el directorio de trabajo devuelva al estado exacto antes de una confirmación en concreto.

git clean -n	Mostrará los archivos que van a ser eliminado sin hacerlo realmente.
git clean -f	Elimina los archivos sin seguimiento del directorio actual. No elimina carpetas ni ficheros incluidos en .gitignore. (no es necesario –f si clean.requireForce esté puesto a false [no recomendable]).
git clean –f <path>	Elimina los archivos sin seguimiento, solo, en la path especificado.
git clean -df	Eliminar archivos y directorios sin sin seguimiento desde el directorio actual.
git clean -xf	Eliminar archivos sin seguimiento desde el directorio actual, así como los archivos que Git que normalmente pasan desapercibidos.

Re-Escribir la historia del repositorio

Una de las principales características de Git es asegurarse de que nunca se pierdan cambios que se han incluido en el repositorio.

También está diseñado para darle un control total sobre el flujo de trabajo de desarrollo.

Git ofrece comandos que permite **reescribir la historia** bajo la advertencia de que **su uso puede provocar la pérdida** de contenidos.

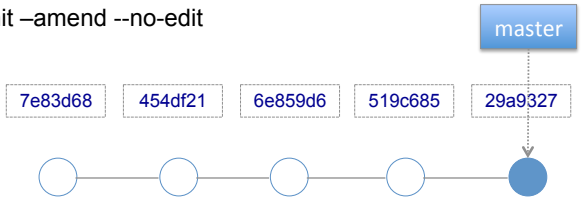
- `git commit –amend`
- `git rebase`
- `git rebase –i`

Reescribir la historia : git commit --amend

Este comando es el más adecuado para el arreglar el último commit realizado.

- **Permite combinar cambios añadidos al stage área con el último commit. En lugar de hacer un nuevo commit con un nuevo snapshot. Corregir prematuros commit.**
- También se puede utilizar para corregir simplemente el mensaje de confirmación anterior sin cambiar snapshot.

git commit –amend --no-edit



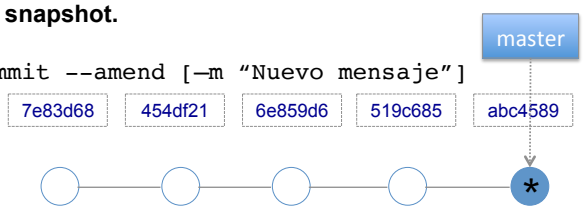
Reescribir la historia : git commit --amend

Este comando el más adecuado para el arreglar el último commit realizado.

Permite combinar cambios añadidos al stage área con el último commit en lugar de hacer un nuevo commit con nuevo snapshot. Corrige prematuros commit.

También se puede utilizar para corregir el mensaje del último commit sin cambiar snapshot.

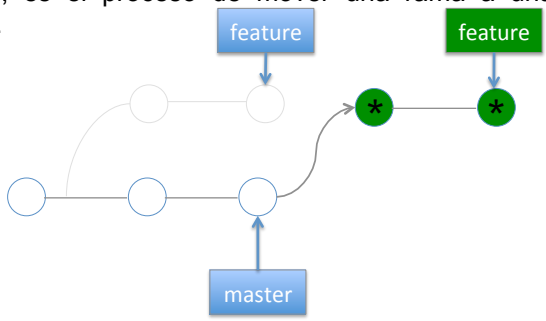
```
git commit --amend [–m "Nuevo mensaje"]
```



No modifica el último commit, lo sustituye completamente. Git lo verá como un nuevo commit. **Nunca ejecutar este comando con commit's que han sido publicados en un repositorio publico.**

Reescribir la historia del repositorio: git rebase

Rebase, es el proceso de mover una rama a una nueva base del commit.



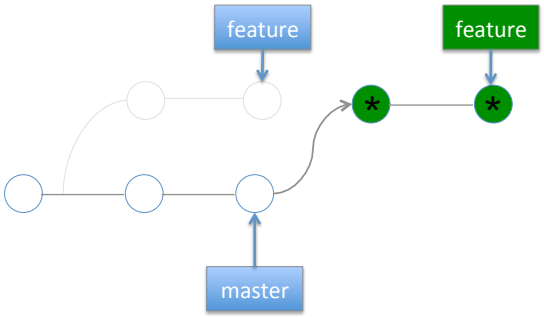
Lo que hace es mover una rama de un commit a otro. Git logra esto mediante la creación de nuevos commit's y su aplicación a la base. Esto es volver a escribir la historia del proyecto.

La rama tiene el mismo aspecto pero se compone nuevos commit.

Reescribir la historia del repositorio: git rebase

```
git rebase <base>
```

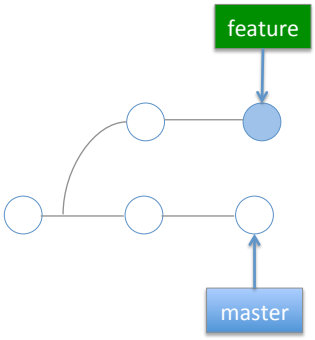
Rebase el actual branch (debemos estar posicionados en él) a <base> y <base> puede ser: cualquier tipo de referencia a un commit por ejemplo: ID commit; rama; tag (etiqueta); una referencia relativa a HEAD.



Reescribir la historia del repositorio: git rebase

La razón principal de cambio de base es mantener un historial de proyectos lineal.

La rama master ha progresado desde que se comenzó a trabaja con la nueva feature.



Tenemos dos alternativas:

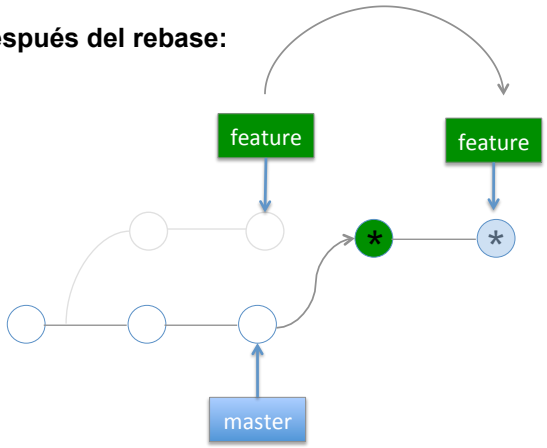
Hacer un merge : 3-way merge.

Hacer un rebase y merge: Genera un fast-forward merge y linealiza la historia de los commit's.

Reescribir la historia del repositorio: git rebase

Hacer rebase en el master + merge genera una historia lineal de nuestro repositorio utilizando como algoritmo de fusión el fast-forward.

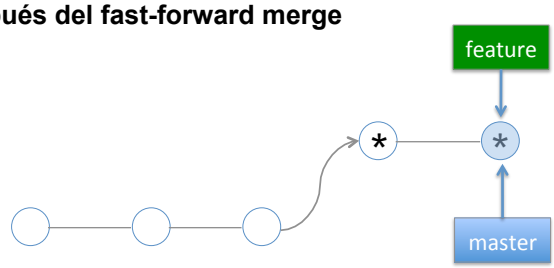
Después del rebase:



Reescribir la historia del repositorio: git rebase

Hacer rebase en el master + merge genera una historia lineal de nuestro repositorio utilizando como algoritmo de fusión el fast-forward.

Después del fast-forward merge



Es una forma común para integrar los cambios ascendentes en su repositorio local.
Utilizar git merge genera commit superfluos cada vez que deseamos ver como ha progresado el proyecto.
"Quiero basar mis cambios en lo que todo el mundo ya ha hecho."

Reescribir la historia del repositorio: git rebase

Si ya se ha publicado los cambios en un repositorio público: **No utilizar el rebase.**

Recordar si hemos publicado no utilizar:

- git commit --amend
- git reset,
- git rebase

Estos comando reemplazaría las antiguos commit por otras nuevos y se vería como esa parte de la historia del proyecto ha desaparecido de forma abrupta.

Reescribir la historia del repositorio: git rebase i

Al ejecutar el comando `git rebase -i` con el flag –i comienza un sesión interactiva

En lugar de mover ciegamente todas las confirmaciones a la nueva base, la sesión interactiva nos da la oportunidad de alterar commit individuales en el proceso.

Esto le permite limpiar la historia mediante la eliminación, la división, y la modificación de una serie confirmaciones existentes.

Reescribir la historia del repositorio: git rebase i

Uso:

```
$ git rebase –i <base>
```

Se realiza el rebase sobre la rama actual. Este comando abre un editor donde se podrá decidir, sobre cada de uno de los commit, que hacer.

Reescribir la historia del repositorio: git rebase i

Este comando da un control completo sobre la historia del proyecto.

El desarrollador está concentrado en el desarrollo y no en la historia. Realizar un borrador de historia que luego podrá perfeccionar.

A muchos desarrolladores les gusta usar un rebase interactivo:

Para pulir una rama de la característica antes de la fusión en la base de código principal.

- Eliminar commit insignificantes.
- Eliminar commit obsoletos.