



Git

Victor Herrero Cazurro



Contenidos

1. Introduccion	1
2. SCV	1
3. SCVC	2
3.1. Desventajas de SCVC	2
4. SCVD	2
5. SCVC vs SCVD	3
5.1. Almacenamiento de los datos	3
5.2. Velocidd	4
5.3. Independencia del servidor	5
5.4. Integridad	5
5.5. Git generalmente sólo añade información	5
5.6. Los tres estados de los datos en GIT	5
6. Instalacion	6
6.1. Instalacion en Windows	7
7. Flujos	10
7.1. Forking Workflow	10
7.2. Centralized Workflow	11
7.3. Git flow	11
7.3.1. Feature Branch	14
7.3.2. Release Branch	14
8. Git vs Svn	15
9. Areas del Respositorio	16
10. Ciclo de Vida	16
11. Configuracion	17
12. Inicializacion	18
13. Comandos Basicos	19
13.1. Status	19
13.2. Add	20
13.3. Commit	20
13.4. Log	21
13.5. Checkout	23
13.6. Show	24
13.7. Diff	24
13.8. Reset	25
13.9. Rm	27
13.10. Mv	29
14. Stash	29



15. Alias	31
16. Gitignore	31
17. Referencias	32
17.1. Referencias Simbolicas	32
17.2. Head	33
17.3. Master	34
17.4. Referencias relativas	35
17.5. Reglas de nombrado de referencias (Tag y Branch)	35
17.6. Referencias por mensaje de commit	35
18. Comandos Avanzados	36
18.1. Rev-parse	36
18.2. Ammend	36
18.3. Clean	36
18.4. Revert	37
18.5. Rebase	37
18.6. Blame	39
18.7. Bisect	39
19. Etiquetas	41
20. Ramas	42
20.1. Merge	43
21. Repositorios	45
21.1. Github	46
21.2. Bitbucket	47
21.3. Clone	47
21.4. Remote	48
21.5. Push	49
21.6. Fetch	50
21.7. Pull	50
21.8. Fork	51
21.9. Pull-request	51
22. Parches	52
23. Herramientas	53
23.1. Gitk	53
23.2. DiffMerge	54
23.3. SourceTree	54
23.4. SmartGIT	55
24. Comandos de fontaneria	55
24.1. Reflogs	55
24.2. Garbage Colector	57



24.3. Garbage Colector	57
25. Bibliografia	57



1. Introduccion

En 1972 aparece el primer sistema para el control del código fuente (SCCS).

En 1982 aparece un nuevo sistema Revision Control System (RCS) que automatiza algunas tareas, pero no permite trabajar con proyectos, solo con ficheros unitarios y tampoco concurrencia de usuarios.

Entre 1986 y 1990, se instaura CVS, que permite concurrencia y mejora el rendimiento de RCS.

En el 2000 aparece SVN, que introduce los commits transaccionales.

De forma paralela en el 2000 y ligado al mundo Linux aparece BitKeeper.

En 2005, BitKeeper se convierte en un producto de pago, y el mundo Linux lo abandona en favor de GIT como SCVD.

Linus Torvalds (creador de Linux), es el impulsor de GIT.

GIT persigue mejorar BitKeeper en varios aspectos:

- Velocidad. Se trabaja en local. No se copian ficheros en la creación de ramas, sino que se emplean punteros.
- Sencillez en su diseño.
- Desarrollo no lineal (ramas paralelas). El cambio de rama en proyectos grandes es muy rápido.
- Completamente distribuido

2. SCV

SCV son las siglas **Sistema Control de Versiones**, que corresponden al ingles VCS (Version Control Systems).

Es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo.

Un SCV permitirá retomar cualquier fichero en un estado anterior, a través de la versión del fichero.



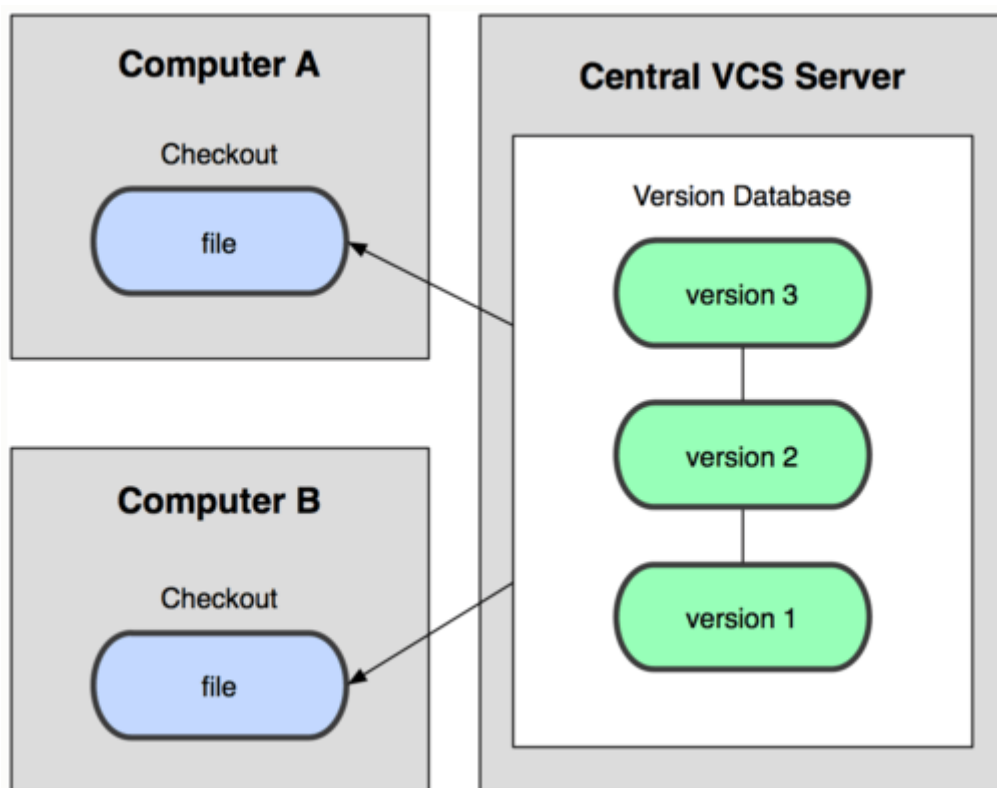
3. SCVC

SCVC son las siglas **Sistema Control de Versiones Centralizado**, en ingles CVCS (Centralized Version Control Systems).

Son la aproximación que durante muchos años se ha tenido ante el problema de la compartición de los ficheros por varias personas.

Ofrecen un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos de ese lugar central.

Seguro que se conocen herramientas como SVN o CVS.



3.1. Desventajas de SCVC

Si el servidor se cae ningún cliente se puede conectar y por tanto colaborar.

Si el disco duro del servidor se corrompe, y no se tiene un buen sistema de copias de seguridad, se pierde todo.

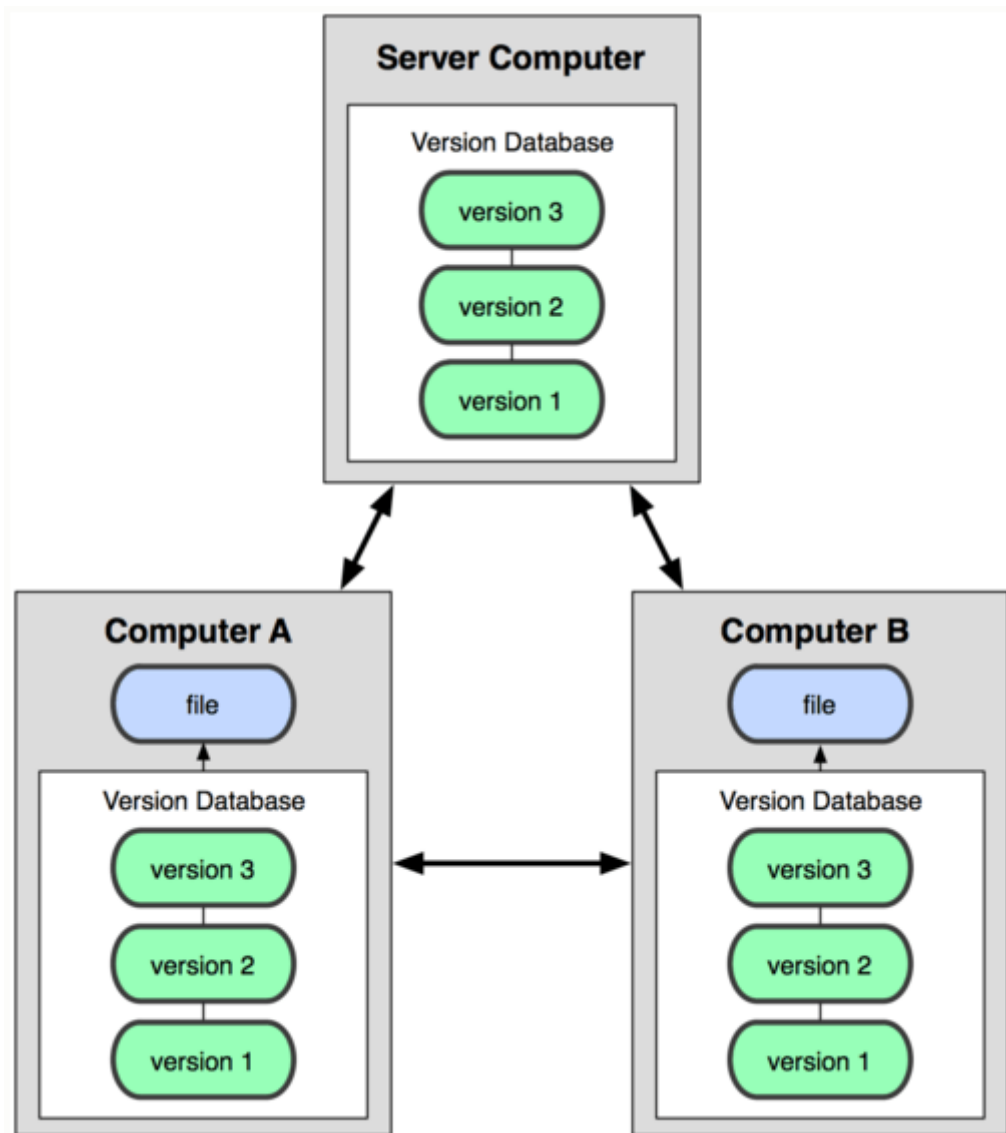
4. SCVD

SCVD son las siglas Sistema Control de Versiones Distribuido, en ingles DVCS

(Distributed Version Control Systems).

En estos sistemas, los clientes no sólo descargan la última instantánea de los archivos, sino que descargan el repositorio completo.

Así, si el servidor tiene algún problema, cualquiera de los clientes puede copiar en el servidor sus ficheros para restaurarlo.

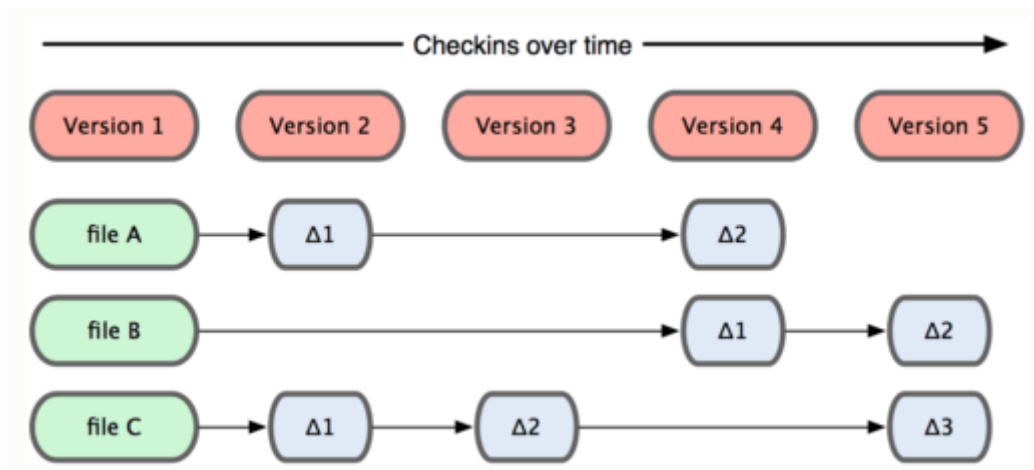


5. SCVC vs SCVD

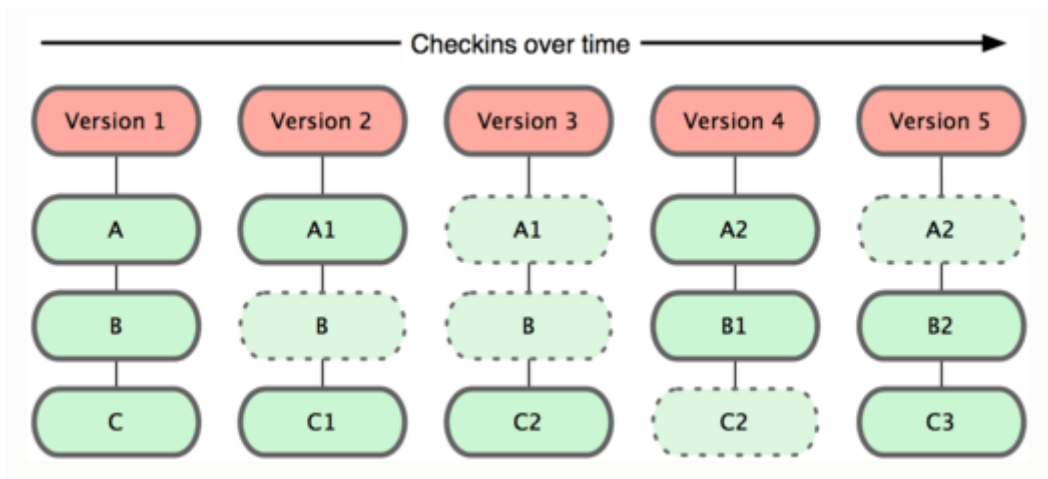
5.1. Almacenamiento de los datos

La mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos, se almacenan un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo, siempre referenciado al archivo base.





GIT almacena una instantánea de todos los archivos que han cambiado, los que no han cambiado no se guardan, sino que se referencia al archivo de la versión anterior.



Esta característica, tiene su pro, que reside en la velocidad de comparación de diferencias entre versiones, ya que al tener los ficheros enteros, únicamente hay que compararlos, en cambio con la anterior aproximación, había que ir aplicando a una versión base, los cambios de todas las versiones hasta conseguir la versión a comparar.

El contra, vendría por el espacio, pero GIT aplica algoritmos de compresión avanzados, que hacen que no exista tal contra.

5.2. Velocidd

La mayoría de las operaciones en GIT son locales, dado que en el repositorio Local, hay una copia de todo lo que hay en el repositorio Remoto.

Por este motivo, la velocidad de GIT es superior, dado que se ahorra los tiempos de transferencia.

5.3. Independencia del servidor

Cada cliente puede realizar los cambios sobre el repositorio sin necesidad de que el servidor esté disponible, ya que tiene todas las piezas necesarias.

Normalmente habrá un servidor, pero no es necesario, se podría compartir el proyecto GIT, entre distintos usuarios.

5.4. Integridad

Todo transferencia de información para su almacenado en GIT, es validada con checksum (suma de comprobación), de esta forma es imposible cambiar los contenidos de cualquier archivo o directorio en el repositorio sin que quede constancia en GIT, por lo que no es posible que produzca corrupción en los ficheros sin que se detecte.

Con esta funcionalidad se consigue por tanto integridad en la transferencia, evitando pérdidas de datos o corrupción en los mismos.

El mecanismo empleado en el checksum, es SHA-1.

5.5. Git generalmente sólo añade información

Casi todas las acciones en GIT, solo añaden información, por lo que es difícil de perder datos.

La única vía de pérdida de datos, es cuando se trabaja de forma local y no se hace push al servidor remoto con frecuencia.

5.6. Los tres estados de los datos en GIT

Los datos en GIT, pueden tener tres estados

- confirmado (committed). Los datos están almacenados de forma segura en local.
- modificado (modified). El fichero ha sido modificado, pero no preparado o confirmado.
- preparado (staged). Se ha marcado un archivo modificado para que forme parte de la próxima confirmación.

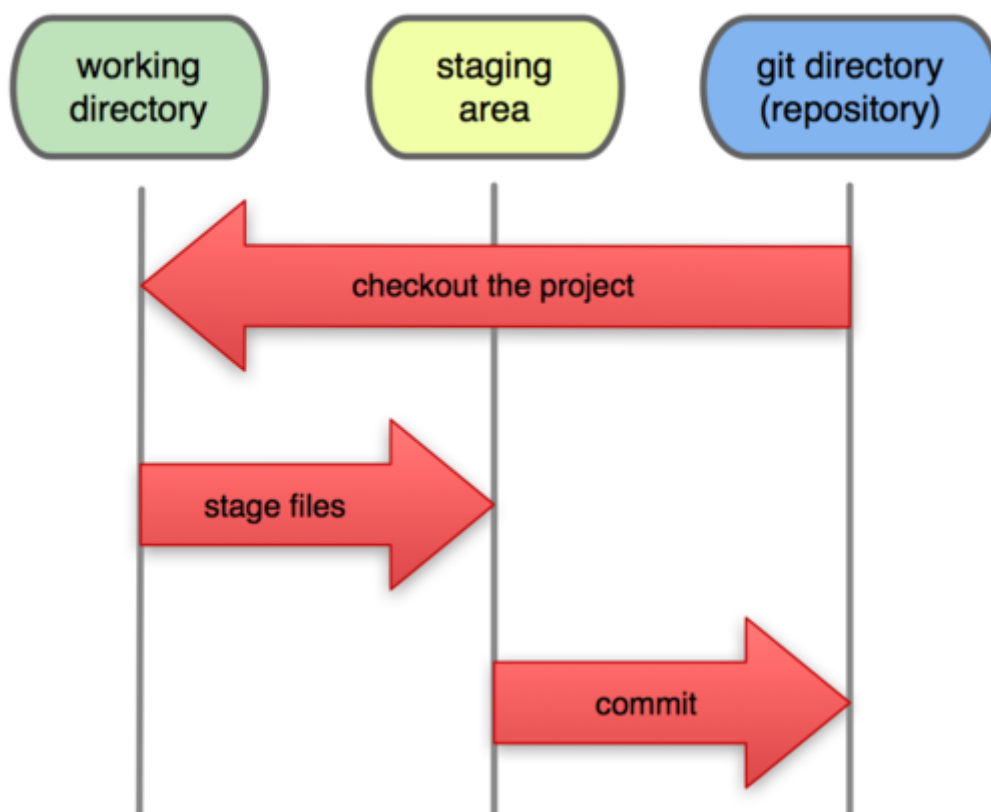
Estos tres estados, nos llevan a las tres secciones principales de los proyectos de



GIT

- Directorio de trabajo (Working directory). Es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de GIT y se descomprimen en disco para que se puedan usar o modificar.
- Área de preparación (Staging area). Es un archivo, generalmente contenido en tu directorio de GIT, que almacena información acerca de lo que va a ir en la próxima confirmación.
- Directorio de GIT o repositorio (GIT directory). Este es el directorio que se clona desde el repositorio, contiene una base de datos de GIT.

Local Operations



6. Instalacion

Instalación en Fedora Linux

```
>yum install git-core
```

Instalación en Debian Linux

```
>apt-get install git-core
```

Instalación en Windows

```
http://git-scm.com/download/win
```

Instalación en Mac OS

```
http://git-scm.com/download/mac
```

6.1. Instalacion en Windows

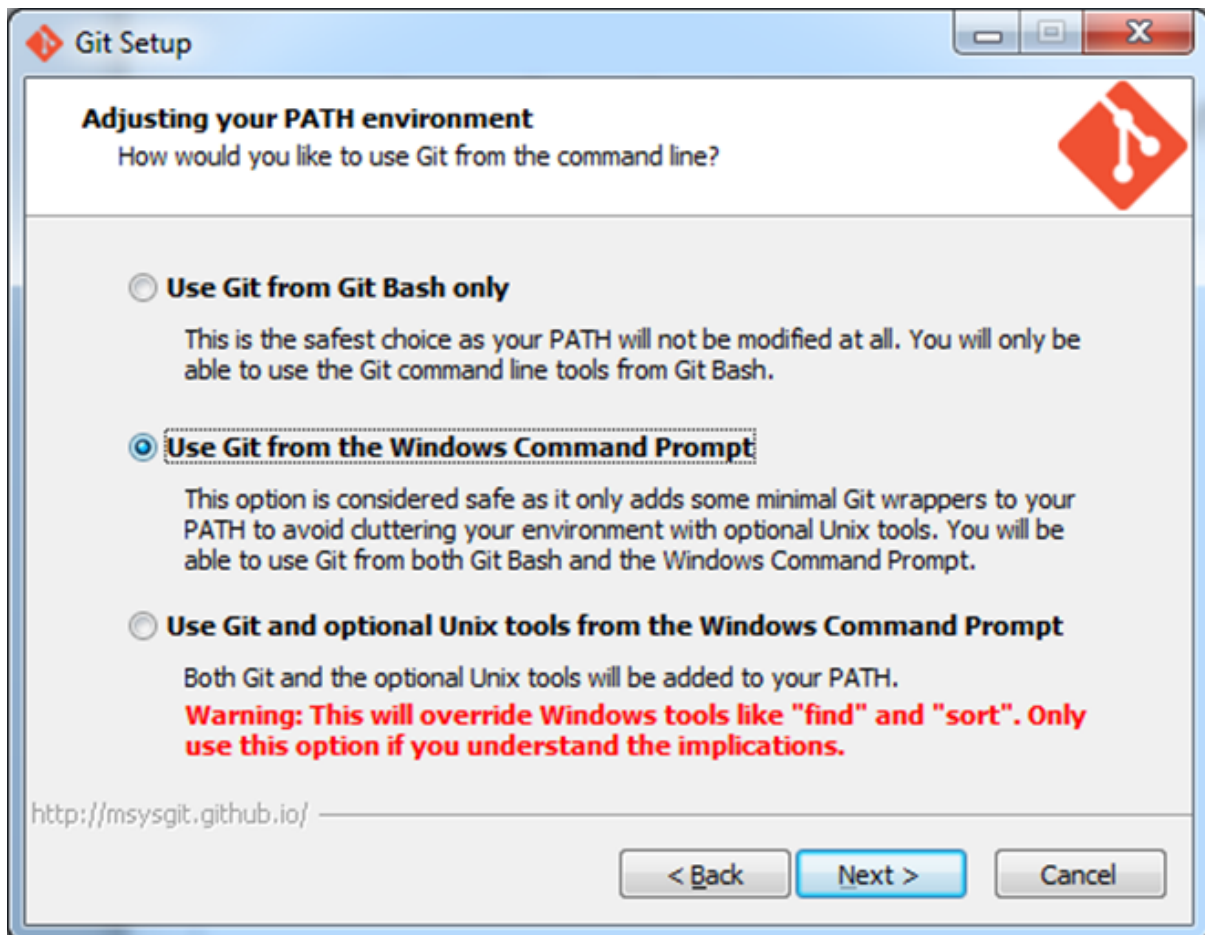
Dado qe Git nace en el mundo Linux, su instalación en sistemas basados en Linux no presenta problemas, pero en los sistemas windows, si presenta alguna consideración.

Modo de consola se desea instalar, las opciones son * Bash. Consola tipo Linux. * Windows Comand Prompt. Consola tipo windows, con las herramientas de GIT. * Unix tools from Windows Comand Prompt. Mezcla que incluye los comandos de Linux en la consola de Windows y que puede sobrescribir comandos de Windows.

NOTE

En un entorno Windows, se recomienda el uso de la segunda, aunque la primera seria igualmente valida. La tercera esta desaconsejada por poder provocar conflictos con otras herramientas.



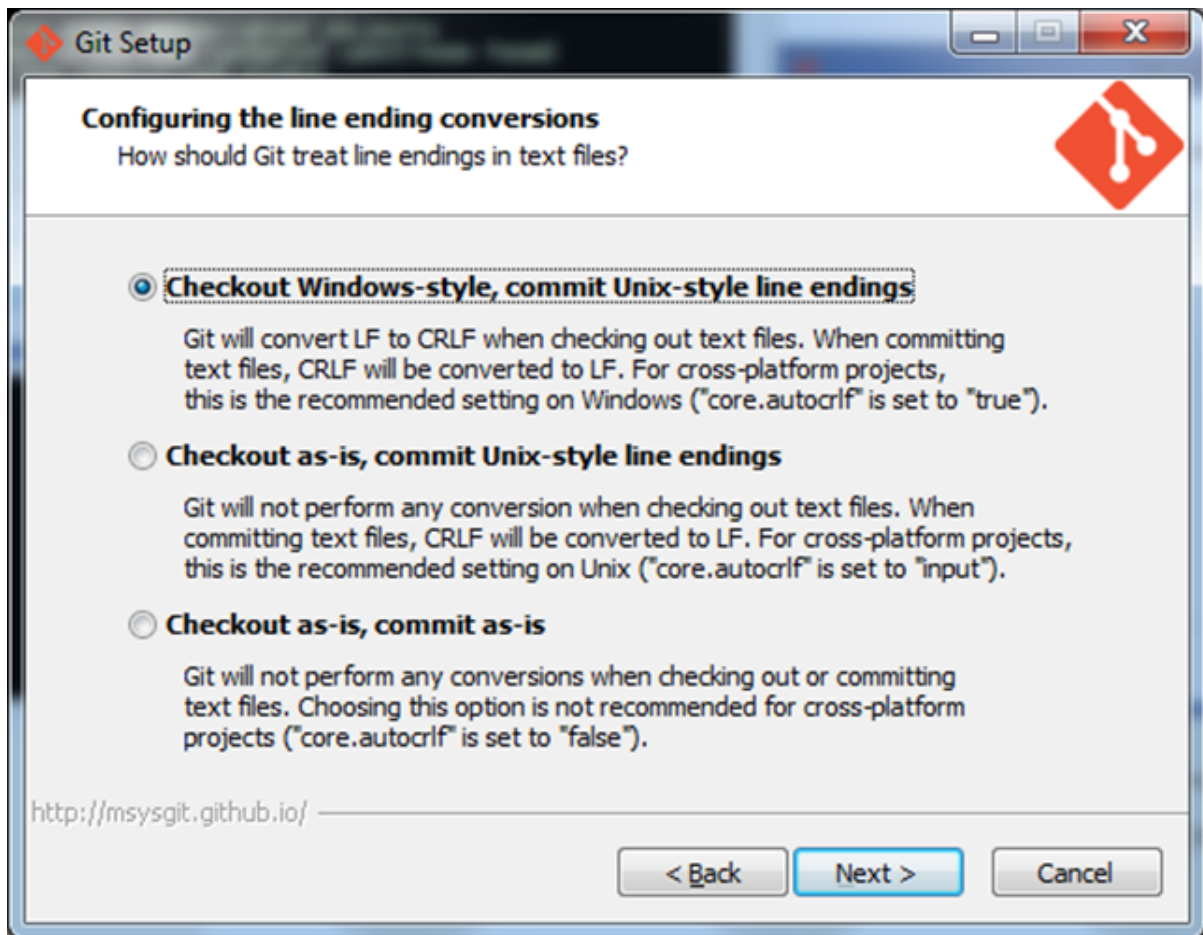


Como se quieren procesar los retornos de carro y los saltos de línea a la hora de hacer los commit (subidas hacia repositorio) y los checkout (descargas desde el repositorio)

- Checkout Windows Style - Commit Unix Style – GIT controla que los ficheros se guarden al estilo Unix y se descompriman al estilo Windows para trabajar con ellos.
- Checkout as-it - Commit Unix Style – Los ficheros se descomprimen tal cual estén y el guardado al estilo Unix.
- Checkout as-it - Commit as-it – No se realiza ninguna transformación.

NOTE

En Windows se recomienda la primera opción para despreocuparse de como procesar los ficheros.



Git Cheat Sheet

<http://git.or.cz/>

Remember: `git command --help`

Global Git configuration is stored in `$HOME/.gitconfig` (`git config --help`)

Create

From existing data
`cd ~/projects/myproject`
`git init`
`git add .`

From existing repo
`git clone ~/existing/repo ~/new/repo`
`git clone git://host.org/project.git`
`git clone ssh://you@host.org/proj.git`

Show

Files changed in working directory
`git status`

Changes to tracked files
`git diff`

What changed between \$ID1 and \$ID2
`git diff $ID1 $ID2`

History of changes
`git log`

History of changes for file with diffs
`git log -p $file $directory/`

Who changed what and when in a file
`git blame $file`

A commit identified by \$ID
`git show $ID`

A specific file from a specific \$ID
`git show $ID:$file`

All local branches
`git branch`

(star "*" marks the current branch)

Cheat Sheet Notation

\$id : notation used in this sheet to represent either a commit id, branch or a tag name
\$file : arbitrary file name
\$branch : arbitrary branch name

Concepts

Git Basics

master : default development branch
origin : default upstream repository
HEAD : current branch
HEAD* : parent of HEAD
HEAD~4 : the great-great-grandparent of HEAD

Revert

Return to the last committed state
`git reset --hard`
⚠ you cannot undo a hard reset

Revert the last commit
`git revert HEAD` Creates a new commit

Revert specific commit
`git revert $id` Creates a new commit

Fix the last commit
`git commit --amend`
(after editing the broken files)

Checkout the \$id version of a file
`git checkout $id $file`

Branch

Switch to the \$id branch
`git checkout $id`

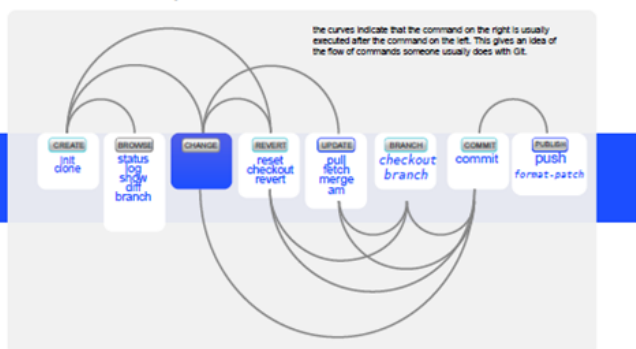
Merge branch1 into branch2
`git checkout $branch2`
`git merge branch1`

Create branch named \$branch based on the HEAD
`git branch $branch`

Create branch \$new_branch based on branch \$other and switch to it
`git checkout -b $new_branch $other`

Delete branch \$branch
`git branch -d $branch`

Commands Sequence



Update

Fetch latest changes from origin
`git fetch`
(but this does not merge them)

Pull latest changes from origin
`git pull`
(does a fetch followed by a merge)

Apply a patch that some sent you
`git am -3 patch mbox`
(in case of a conflict, resolve and use `git am --resolved`)

Publish

Commit all your local changes
`git commit -a`

Prepare a patch for other developers
`git format-patch origin`

Push changes to origin
`git push`

Mark a version / milestone
`git tag v1.0`

Useful Commands

Finding regressions

`git bisect start` (to start)
`git bisect good $id` (\$id is the last working version)
`git bisect bad $id` (\$id is a broken version)
`git bisect bad/good` (to mark it as bad or good)
`git bisect visualize` (to launch git and mark it)
`git bisect reset` (once you're done)

Check for errors and cleanup repository

`git fsck`
`git gc --prune`

Search working directory for foo()
`git grep "foo()"`

Resolve Merge Conflicts

To view the merge conflicts

`git diff` (complete conflict diff)
`git diff --base $file` (against base file)
`git diff --ours $file` (against your changes)
`git diff --theirs $file` (against other changes)

To discard conflicting patch

`git reset --hard`
`git rebase --skip`

After resolving conflicts, merge with

`git add $conflicting_file` (do for all resolved files)
`git rebase --continue`

Don't forget to push your changes!

7. Flujos

7.1. Forking Workflow

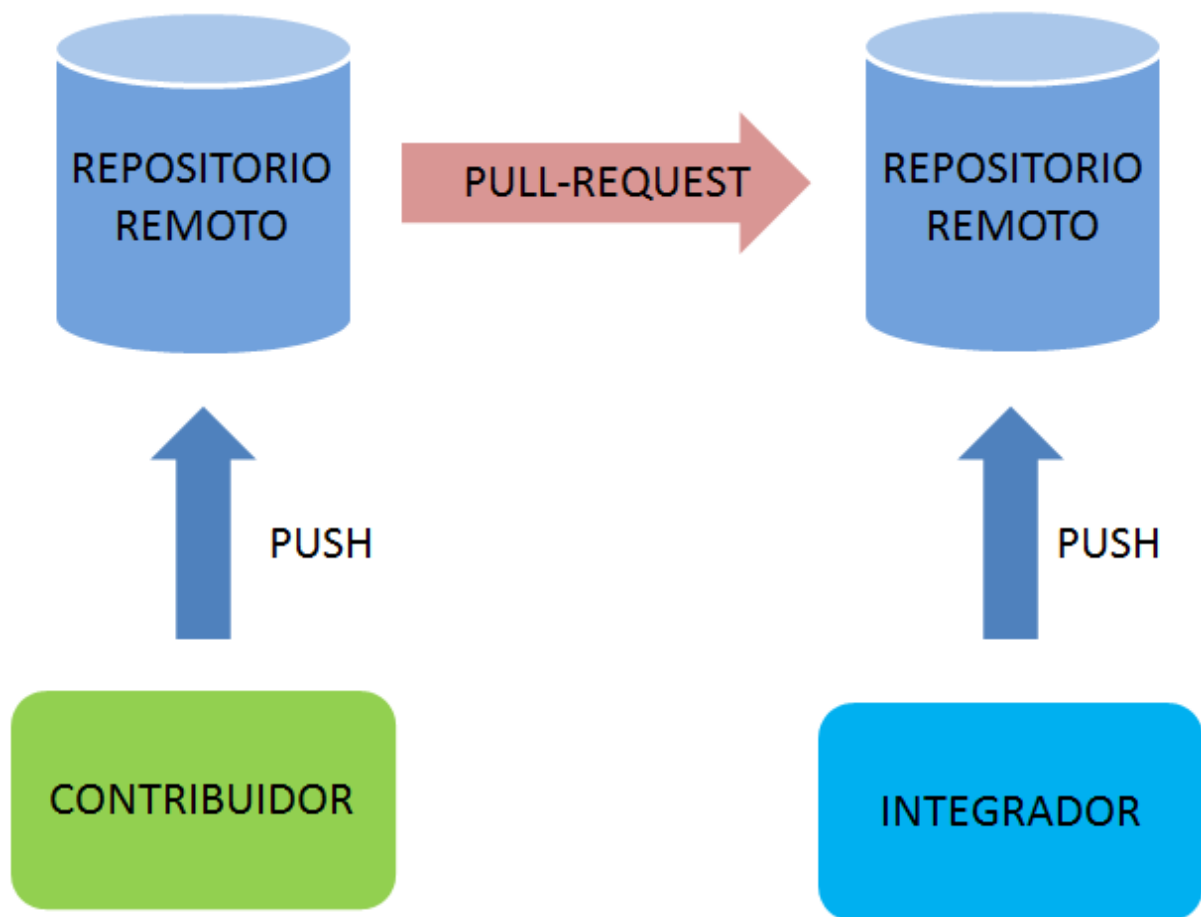
Es un flujo basado en Forks.

Permite que el usuario propietario del repositorio valide cada una de las aportaciones al repositorio.

Se implementa en plataformas como Github.

El escenario de uso seria:

- Dados 2 usuarios distintos de Github, llamémosles Colaborador e Integrador.
- El Integrador tiene un proyecto en Github.
- El Colaborador, se hace un Fork (clona) del proyecto del Integrador en su cuenta de usuario del servidor de repositorios.
- Una vez clonado, el Colaborador se descarga el proyecto clonado a Local.
- El Colaborador evolucionará el repositorio clonado y lo publicará en el servidor de repositorios.
- El Colaborador lanzará un Pull-Request sobre el repositorio original del Integrador.
- El Integrador se bajara el código del Colaborador a Local, a una rama temporal, y lo revisará.
- Si el Integrador acepta el cambio, lo mezcla con la rama evolucionada de su repositorio original y lo publica en el servidor de repositorios.



7.2. Centralized Workflow

El procedimiento, constaría de un branch por desarrollador/funcionalidad en Local.

El desarrollador realiza commit sobre su rama en Local.

Y la sube con push al Repositorio Remoto esos commit.

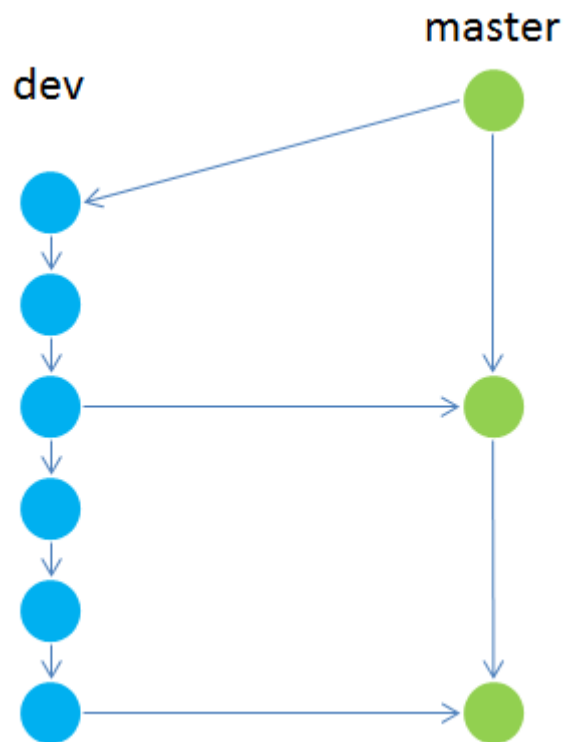
En el Repositorio Remoto, crea un pull-request con la rama principal como base y la rama del desarrollo como rama con la que comparar.

images::centralized_workflow.png[]

7.3. Git flow

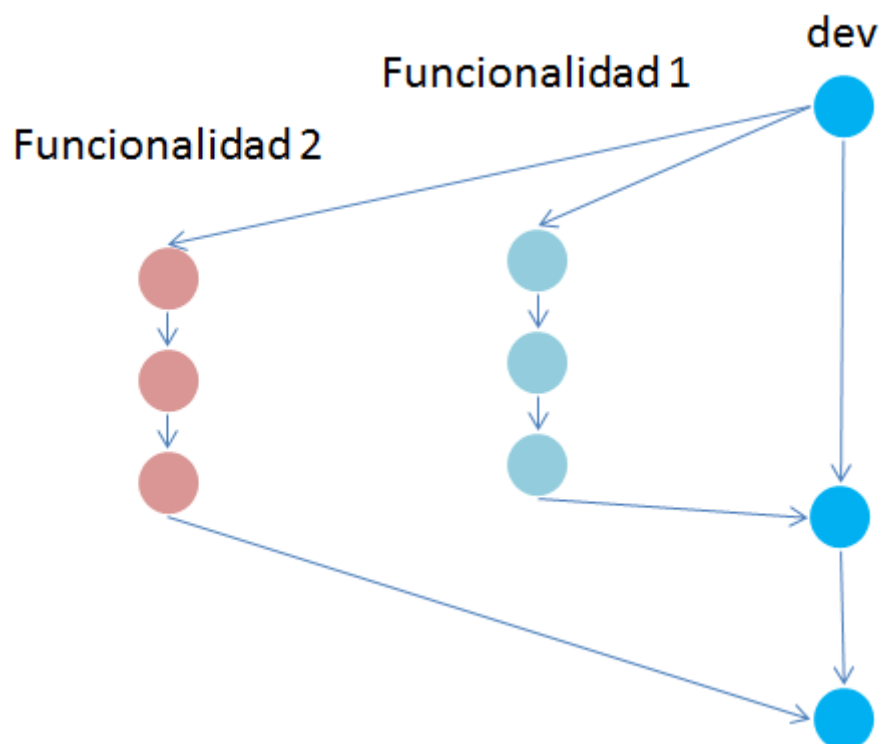
Dos ramas principales

- **develop.** Ira teniendo los commit necesarios para completar la siguiente versión a desarrollar.
- **master.** Tendrá las versiones estables del código, cada version con un Tag que indique dicha versión.

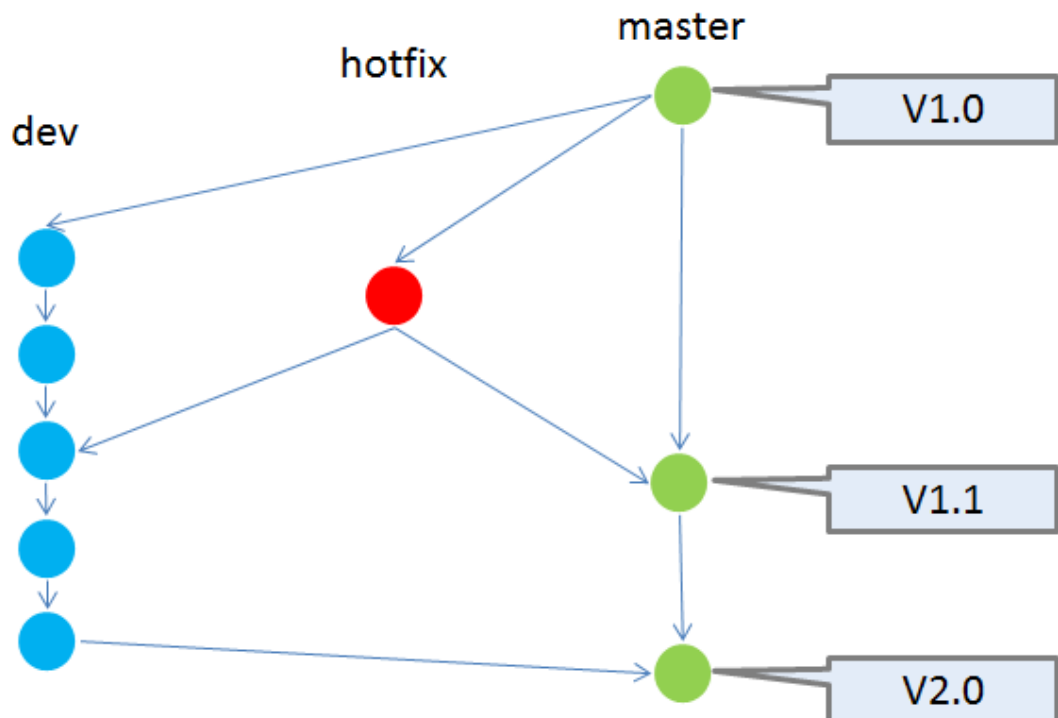


Otras ramas secundarias

- **feature-x**: Ramas donde se irán desarrollando las distintas funcionalidades que compongan una **release**. Estas ramas, nacen de **develop** y se van fusionando con **develop** cuando están acabadas.



- **bugfix-x**: Ramas donde se irán desarrollando las correcciones de los bug encontrados durante la fase de desarrollo, sale y se mergean con **develop**.
- **hotfix-x**: Ramas donde se irán desarrollando las correcciones de los bug encontrados en producción, salen de **master** y se mergean tanto en **develop** como en **master**.



- **release-x**: Está destinada a ser un paso intermedio entre que en **develop** se acaba el desarrollo de una versión y se lleva a **master** por ser estable, los commit sobre esta rama serán **Release Candidate**. En esta rama, únicamente se incluirían bugs críticos (no todos como en dev), para no lastrar el tiempo empleado en lanzar la release. Esta rama deberá estar abierta el menor tiempo posible.

Para la generación de esta estructura de ramas GitFlow, se lanza el comando

```
> git flow init
```

Con este comando se preguntan los nombres de las dos ramas principales y los prefijos para las ramas secundarias



```
How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? [] v
Hooks and filters directory? [D:/GitLab/Attlassian-Bamboo-
CI/workspace/Demo/.git/hooks]
```

7.3.1. Feature Branch

Estas ramas estan pensadas para albergar el desarrollo de una nueva funcionalidad de forma independiente al resto.

Cuando se quiera crear una nueva feature, se debe crear una rama de feature, esta se debera llamar **feature/xxxx**, por ejemplo **feature/pagina_inicio**

Se puede crear con el comando

```
> git flow feature start <nombre de la rama sin prefijo>
```

Cuando se termina el desarrollo en la rama de feature, los cambios se han de llevar a la de desarrollo, para ello se puede lanzar el comando

```
> git flow feature finish <nombre de la rama sin prefijo>
```

7.3.2. Release Branch

Estas ramas están pensadas para estabilizar una puesta en producción, es decir desacoplar el desarrollo, que se hace en la rama develop con produccion, que es lo que se tiene en master.

La idea es que cuando se tengan el numero de features en develop deseadas, se cree una rama para la siguiente release, en esa rama se trata de estabilizar la entrega, por lo que una vez bifurcada no se admiten mas features, unicamente correccion de errores, hotfix, los commit en esa rama de release, seran Release Candidate, cuando la release es estable, se mergeará con master, etiquetando el commit de master como nueva release y con develop, para incorporar posibles

correcciones que se hayan hecho en el proceso de estabilizar la release.

Para iniciar una nueva rama de release, se lanza en comando

```
> git flow release start <version de la release>
```

Una vez se estabiliza la release, es hora de fusionar con master y develop, para ello se lanza el comando

```
> git checkout master  
> git checkout merge release/<version de la release>  
> git flow release finish '<version de la release>'
```

8. Git vs Svn

Repositorio distribuido (Git) Trabajar sin conexión Cambios en local. Ramificación local al gusto del desarrollador. No hay dependencia del servidor. No hay punto débil en el servidor. Creación de ramas fácil y sencillo. Mas rapido Repositorio mas pequeño. Ramas mas naturales. Commits intermedios Mejor información en la mezcla y las ramas.

Repositorio centralizado (SVN) Todos conectados al repositorio central. No se puede trabajar sin conexión Los cambios afectan a todos. Creación de ramas costoso. Control extra de los commit para que no incluyan inestabilidad. Numeración de las versiones mas faciles de recordar. Descarga parcial del repositorio. Multiplataforma.

Tomando como referencia el Kernel de linux que trabaja con Git, este cumple con:

- Tiene 22000 archivos.
- Se hacen 4.5 merges al día.
- Se tardan 7 segundos aproximadamente en mostrar las diferencias de todo el kernel, unos 74000 commits.

Tomando como referencia el repositorio de Mozilla Project

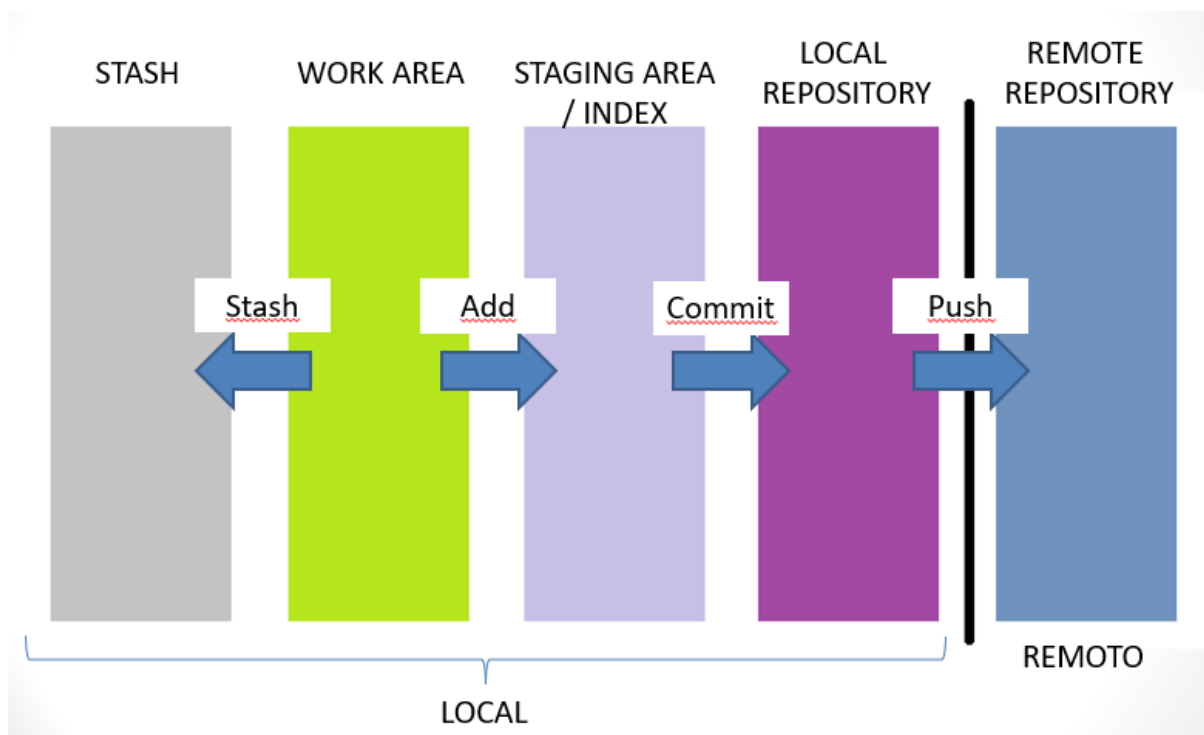
- En CVS ocupa 3Gb.
- En SVN 12Gb.
- En Git 300Mb.



9. Areas del Respositorio

En Git se definen varias areas en las que pueden estar los ficheros y que estan relacionadas con el ciclo de vida de los mismo en Git. Esta areas son

- **Stash:** Zona de aparcamiento de los cambios, sirve para apartar cambios del flujo de trabajo y que no estorben en los commit.
- **Work area:** Zona de trabajo, reflejan el estado actual de los ficheros que componen el proyecto en Local, puede reflejar un estado sincronizado o con cambios pendiente de consolidar
- **Staging area:** Zona previa al commit, que marca los cambios que se quieren consolidar en el proximo commit.
- **Local repository:** Refleja el estado consolidado de los ficheros en local.
- **Remote repository:** Refleja el estado consolidad de los ficheros en remoto.



10. Ciclo de Vida

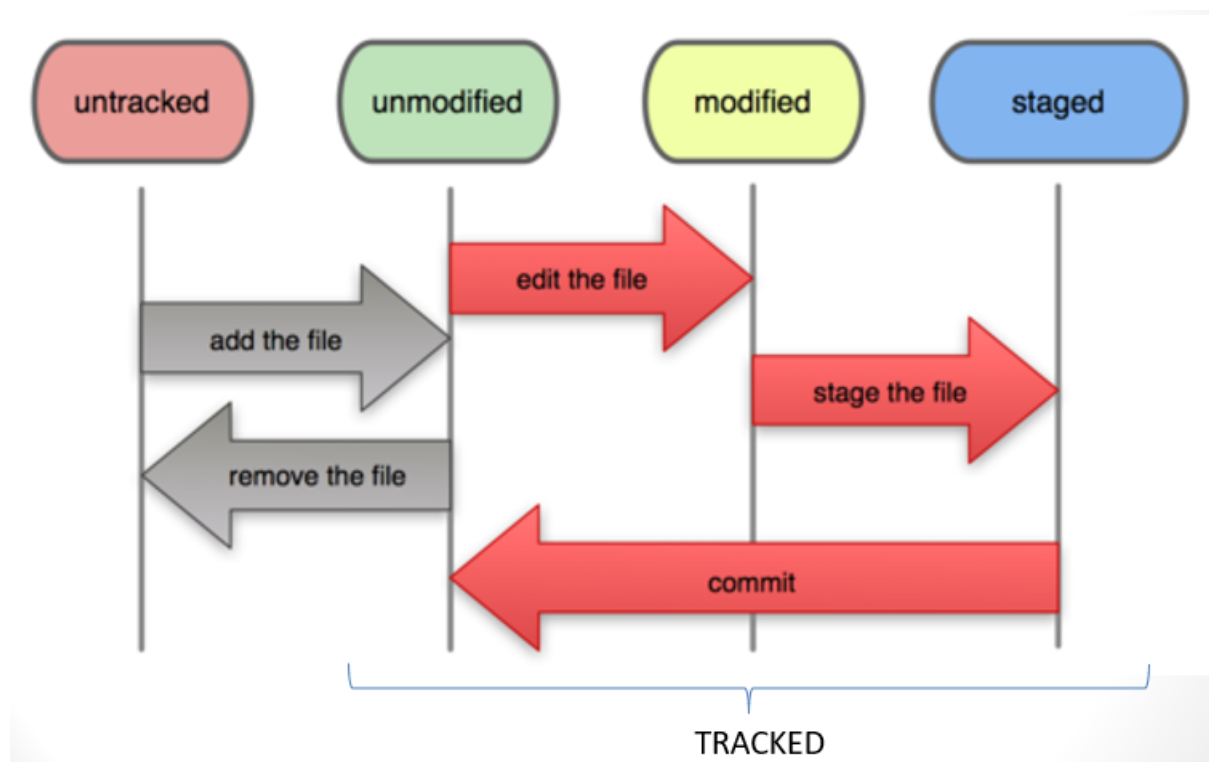
Los ficheros en GIT pueden pasar por varios estados, dependiendo de las operaciones que se ejecuten sobre ellos.

Hay dos estados principales

- **Untracked.** No gestionados por GIT. Sucede cuando se añade un nuevo fichero al Working Area, pero no se ha marcado para incluirse en el repositorio.
- **Tracked.** gestionados por GIT.

Dentro del segundo estado (Tracked), existen varios subestados

- **Unmodified.** Fichero ya persistido alguna vez, que esta sincronizado con el repositorio.
- **Modified.** Fichero ya persistido alguna vez, que ha sido modificado con respecto a lo que hay en el repositorio.
- **Staged.** Fichero marcado para ser almacenado en el repositorio en el siguiente commit.



11. Configuración

Para configurar GIT, se tiene el comando config

```
> git config
```

Las configuraciones de GIT se pueden establecer a tres niveles distintos

- **Sistema.** Fichero **C:\Program Files (x86)\Git\etc\gitconfig**



```
> git config --system
```

- **Usuario.** Fichero **C:\Users\Victor\gitconfig**

```
> git config --global
```

- **Proyecto.** Fichero **<directorio del repositorio>\.git\config**

```
> git config --local
```

Los niveles son jerárquicos, pudiendo sobrescribir configuraciones, siendo el que mas prioridad tiene el de proyecto.

Una vez establecidos los niveles de configuración, lo primero es establecer con que usuario se realizan los Commit al repositorio, dado que es una configuración de usuario, se establecerá con **--global**.

```
> git config --global user.name "John Doe"  
> git config --global user.email johndoe@example.com
```

Para listar las configuraciones establecidas

```
> git config --list
```

NOTE

muchos de los modificadores de los comandos de git, tienen un formato reducido, como en el caso --list, que seria -l

Se pueden configurar cosas como el editor a emplear para introducir los comentarios de los commit.

```
> git config --global core.editor
```

12. Inicializacion

Existen dos formas de obtener un proyecto GIT

- Crearlo de forma local.
- Obtener un clon de un repositorio remoto.

Para crear el proyecto en local

```
> git init <nombre del directorio que alberga el repositorio>
```

Para obtener un clon de un repositorio remoto

```
> git clone <url del repositorio remoto> <directorio donde se quiere  
situar en local>
```

Con estos comandos se crea dentro del directorio sobre el que se ha ejecutado, la carpeta `.git`, que contiene el repositorio.

13. Comandos Basicos

Veamos algunos de los comandos de Git mas basicos

13.1. Status

Este comando permite la comprobación del estado de los ficheros.

```
> git status
```

Mostrará el estado de cada uno de los ficheros que están dentro del directorio y subdirectorios del proyecto GIT.

La consola emplea el rojo para aquellos ficheros que sean nuevos o estén modificados, pero que estén **unstaged** (no vayan a participar del proximo commit).

La consola emplea el verde para aquellos ficheros que estén en staged (que vayan a participar del proximo commit).

No se mostrarán los que estén en seguimiento sin modificaciones respecto al commit de referencia.

Se indica la rama (branch) en la que se está.



13.2. Add

Para que los ficheros sean gestionados por GIT, hay que ejecutar el comando

```
> git add <patron de seleccion de ficheros>
```

Con lo que el fichero pasa de **Untracked** o **Unstaged** a **Staged**.

Si se pasa el nombre de un directorio, se ejecuta de forma recursiva el comando add sobre todos los ficheros contenidos.

Si entre que se ejecuta el comando add sobre un fichero y se ejecuta el commit, se modifica dicho fichero, este fichero aparecerá como staged y unstaged de forma simultanea, en la zona staged aparecerá con la modificación antes del add y en unstaged con la modificación posterior al add.

Se aceptan comodines en las expresiones (*)

```
> git add *.html
```

También se pueden añadir todos los ficheros del Working Area al Staging Area.

```
> git add .
```

13.3. Commit

Un commit, es la forma de almacenar cambios en un repositorio, permite trasladar todo lo que esta en Staging Area, al repositorio local.

Los commit son atómicos, o se hacen o no.

Solo hay dos formas de introducir commits en el repositorio

- Con el comando commit.
- Con el comando merge.

De ejecutarse tal cual, se abrirá el editor de textos configurado, para introducir el comentario del commit, dado que siempre es necesario incluir un comentario en el commit para conocer que cambios se han producido en el código fuente al hacer

dicho commit.

```
> git commit
```

Si se desea hacer el commit e incluir el comentario en una sola línea

```
> git commit -m "Comentario"
```

Cuando se emplea el comando commit sin modificadores, se incluye en el comentario el resultado de ejecutar el comando **status**.

Si se quiere tener también el resultado de la ejecución del comando **diff**, se ha de incluir el modificador -v

```
> git commit -v
```

Este comando permite pasar por Staging Area de forma automática empleando el modificador -a, siempre que los ficheros han sean ya ficheros Tracked.

```
> git commit -a
```

Con este modificador se incluyen en el commit, los que están en Staging Area y también los que no están en Staging Area.

Se pueden hacer commit solo de algún fichero que están en el Staging Area, en lugar de hacerlo de todos

```
> git commit -m "Mensaje" <nombre del fichero>
```

13.4. Log

Este comando permite visualizar el histórico de cambios que se han producido en un repositorio local.

```
> git log
```

Es equivalente a poner



```
> git log HEAD
```

Luego en realidad muestra todos los commit de la rama actual anteriores a la referencia indicada.

Sin parámetros, lista los commits en orden cronológico inverso, con los siguientes parámetros

- La suma de comprobación SHA-1.
- El nombre del autor.
- La dirección de correo del autor.
- La fecha del commit.
- El mensaje de confirmación.

También se le puede indicar rangos con ..

```
> git log master~2..master~4
```

En este caso se muestran los commit entre 2 commit antes de master y 4 commit antes de master.

También se le puede indicar los commits en los que ha participado un fichero concreto

```
> git log -- index.html
```

Las modificadores que permite el comando son

- **-p**: Muestra las diferencias entre cada versión
- **-<numero>**: Lista los últimos <numero> de commits.
- **--oneline**: Muestra el log, reducido en una línea
- **--no-merge**: Quita del listado de commits, aquellos producidos por un merge
- **--decorate**: Muestra los punteros.
- **--graph**: Muestra la estructura grafica de las ramas.
- **--all**: Muestra todas las ramas.

- **--stat**: Muestra tras cada confirmación una lista de archivos, indicando cuántos han sido modificados y cuántas líneas han sido añadidas y eliminadas para cada uno de ellos.

También se incluyen modificadores temporales * **--since**, **--after**: Filtra por fecha, mostrando desde una fecha * **--until**, **--before**: Filtra por fecha, mostrando los anteriores a una fecha

El formato de las fechas será {yyyy-MM-dd}

13.5. Checkout

Comando que permite posicionarse en una rama (branch)

```
> git checkout master
```

O commit

```
> git checkout e5ce841
```

Se emplean como identificador del commit, los 7 primeros dígitos del SHA1.

Además, de no existir la rama, permite crearla

```
> git checkout -b <nombre de la rama>
```

Y crearla basándose en otra rama existente o commit

```
> git checkout -b <nombre de la nueva rama> <nombre rama existente o id de commit>
```

El efecto que se produce al hacer un checkout, es que se cargan en el Working Area, los ficheros asociados al commit (o rama) cargado con checkout, posicionando HEAD en dicho commit (o rama).

Los ficheros untracked, no se ven afectados por este comando, ya que no tienen un estado conocido por git.

También permite volver un fichero al último commit.



```
> git checkout -- <nombre del fichero>
```

El fichero ha de estar en Working Area, si esta en Staging Area, hay que recurrir a **reset**.

```
> git reset HEAD <ruta del fichero>
```

También se puede traer el fichero en el estado que estaba en otro commit, no es necesario que sea el último.

```
> git checkout <identificador del commit> <nombre del fichero>
```

En este caso el fichero aparece como modificado y como **staged**, manteniendo el HEAD en la rama que se estaba antes de la ejecución del comando.

13.6. Show

Comando que permite visualizar el contenido de un commit

```
> git show <identificador del commit>
```

El identificador del commit será el SHA1 completo, el corto o también se pueden emplear los nombres de ramas o etiquetas.

13.7. Diff

Permite comprobar que cambios se han producido dentro de los ficheros.

Ese comando compara lo que hay en el Working Area, con lo que hay en el Repositorio Local, mostrando los cambios dentro de los ficheros.

```
> git diff
```

Sin modificadores, solo indica los cambios realizados sobre los ficheros del Working Area, que todavía no se han añadido al Staging Area.

Los archivos nuevos no se muestran.

También se pueden ver los cambios del Staging Area, que irán en el próximo commit, comparados con los del último commit.

```
> git diff --cached  
> git diff --staged      #Version de GIT mayor que 1.6.1
```

En este caso aunque sea un fichero nuevo lo que se quiere comitar, ya aparecerá (con --staged), ya que cuando está en el Staging Area ya esta siendo trackeado por git.

Como resumen

- **git diff** muestra los cambios no preparados, sin incluir los ficheros nuevos.
- **git diff --cached** muestra los cambios preparados, incluyendo los ficheros nuevos.

Tambien se pueden ver lo cambios ente dos commit en particular

```
> git diff <identificador de commit mas antiguo> <identificador de  
commit mas nuevo>
```

La salida de este comando genera un parche, por lo que si se vuelca sobre un fichero y se distribuye, puede emplearse para trasladar commits a otros repositorios. Para generar el parche bastaría con

```
> git diff > cambio.patch
```

Y para aplicarlo en otro repositorio, previa copia

```
> git apply cambio.patch
```

13.8. Reset

Permite eliminar commit en una rama.

No se recomienda el uso de reset después de hacer push, ya que podría quedar incoherente el Repositorio Local con respecto al Remoto.



También permite sacar cambios del Staging Area.

```
> git reset HEAD <nombre de fichero>
```

Si hay tags en los commit que se quieren eliminar, se han de eliminar previamente, ya que sino no se puede realizar el reset.

El comando tiene los siguientes modificadores:

- **soft:** Borra los commit de la rama actual, dejando los cambios de los commit borrados en el Staging Area.

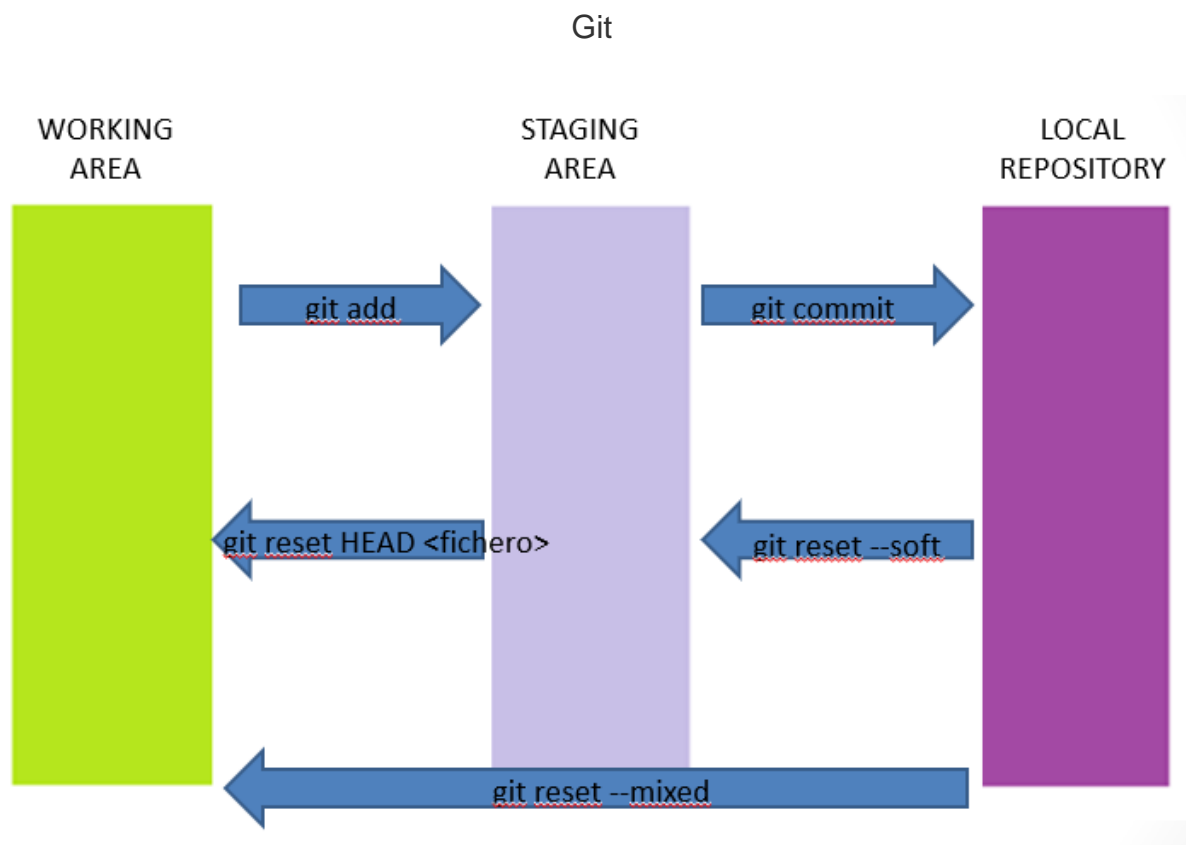
```
> git reset --soft <identificador de commit>
```

- **mixed:** Borra los commit de la rama actual, dejando los cambios de los commit borrados, en el Working Area.

```
> git reset --mixed <identificador de commit>
```

- **hard:** Elimina los commit, y los datos asociados de la rama actual, no se almacenan en ningún sitio.

```
> git reset --hard <identificador de commit>
```



Con este comando, se pueden aplicar varias técnicas, como

- **Squasing**. Que realiza el aplanado de commit, es decir deja varios commit en uno.

```
> git reset --soft <identificador del commit anterior al primero que se quiere aplanar>
> git commit -m "mensaje unificado para los commit"
```

- **Splitting**. Divide un commit en varios.

```
> git reset --soft HEAD^
> git commit -m "mensaje primer commit" <ficheros del primer commit>
> git commit -m "mensaje segundo commit" <ficheros del segundo commit>
```

13.9. Rm

Para sacar ficheros de Staging Area, existen dos vías dependiendo del estado del fichero. Si todavía no se ha hecho ningún commit del fichero, se puede quitar del staging area con la opción **--cached** del comando `rm`.

```
> git rm --cached <ruta del fichero>
```



Este comando en realidad, no solo lo saca del Staging Area, sino que lo pasa a untraked.

En cambio si el fichero ya ha sido persistido alguna vez en el repositorio local, se ha de emplear el comando reset.

```
> git reset HEAD <ruta del fichero>
```

Este comando, realmente sacaría ficheros en los dos estados anteriores (nuevo y modificado).

Para el borrado de ficheros, se pueden seguir varios caminos

Si se borra del disco con las herramientas del SO, el al estar gestionado por GIT, pasa a un estado modificado, si se ejecuta el comando

```
> git rm <ruta del fichero>
```

La próxima vez que se haga un commit, el fichero se eliminará del repositorio local.

Si el fichero esta en Staging Area, es decir, se había modificado y preparado previamente a su borrado, GIT solicitará para borrarlo la opción -f como medida de seguridad

```
> git rm -f <ruta del fichero>
```

Como hemos visto antes, este comando permite también que un fichero deje de ser seguido por GIT, esto es el fichero permanece en el disco duro, pero GIT no lo gestiona, para ello

```
> git rm --cached <nombre fichero>
```

El comando rm, acepta patrones


```
> git rm log/*.log      # Elimina todos los archivos .log, dentro
                        # del carpeta log

> git rm /*~            # Elimina todos los archivos terminados
                        # en ~
```

13.10. Mv

Este comando permite el renombrado de ficheros en GIT, de forma sencilla.

Sin el comando tendríamos que hacer algo como

```
> mv README.txt README  # Comando de SO para
                        # renombrar
> git rm README.txt      # Borrar de GIT el fichero con
                        # nombre antiguo
> git add README         # Añadir a GIT el fichero con el
                        # nombre nuevo
```

Con el comando **mv**, haríamos únicamente

```
> git mv README.txt README
```

14. Stash

Zona para almacenar de forma temporal los commits.

Se emplea para persistir el estado del trabajo de forma temporal para quedar el **Working Area** limpio, sin las ultimas modificaciones realizadas, que todavía no son finales y por tanto no pueden ir al Repositorio Local con un commit.

Para introducir en esa zona los commits, se tiene el comando **stash**.

```
> git stash
```

El comando **list**, permite ver los commit en la zona de **Stash**, el equivalente al comando **log**



```
> git stash list
```

El modificador **-h**, permite ver la ayuda del comando.

El comando **pop**, extrae el ultimo commit temporal almacenado en **Stash** dejándolo en **Working Area**, y lo borra del Stash.

```
> git stash pop
```

El comando **apply**, permite extraer un commit que no sea el ultimo incluido dejándolo en **Working Area**, pero no lo borra del **Stash**.

```
> git stash apply stash@{1}
```

El comando **drop**, permite borrar un commit temporal

```
> git stash drop stash@{1}
```

El comando **save**, permite incluir un mensaje en el commit temporal, de no incluirse, se pone como comentario, el del commit actual.

```
> git stash save "Mensaje para identificar el commit temporal"
```

Si se desea volver los ficheros también al estado en el que fueron incluidos en **stash**, se ha de emplear el modificador **--index**

```
> git stash apply --index
```

De otra manera, los cambios siempre vuelven a **Working Area**.

Los cambios almacenados en **stash**, también se puede recuperar en un rama nueva

```
> git stash branch <nombre de la nueva rama>
```

15. Alias

Desde la versión 1.4.0, se permite la creación de alias de los comandos de GIT. Los alias, no son mas que simplificaciones de comandos, para facilitar su escritura. No se recomienda su uso al empezar a manejar GIT, ya que puede inducir a error. Son especialmente recomendables con aquellos comandos con parámetros que se repiten habitualmente.

La sintaxis seria

```
> git config --global alias.<nombre del alias> "comando"
```

Por ejemplo uno de los mas habituales

```
> git config --global alias.logall "log --oneline --all --decorate"
```

A partir del momento de esta definición, los siguientes comandos son equivalentes

```
> git log --oneline --all --decorate  
> git logall
```

16. Gitignore

Fichero que permite ignorar algunos ficheros para que no sean manejados por GIT.

En el caso de java, por ejemplo los .class, .jar, etc.

O en el caso de proyectos eclipse **.Metadata/**, **.settings/**, **.classpath** o **.project**

El fichero **.gitignore** se ha de situar en el raíz del repositorio.

En github, hay un [repositorio](#) con una lista de ficheros gitignore con preconfiguraciones para añadir a los proyectos

Se pueden emplear caracteres especiales

- **\#**: Comentario
- **!**: Negación



- *: comodín

Se pueden poner path relativos con los directorios.

Algunos ejemplos

```
# Esto es un comentario, y no se considera

*.a           # No se tienen en cuenta ficheros con extensión a

!lib.a        # Si se tiene en cuenta el fichero lib.a

/TODO         # Se ignora el fichero TODO en el directorio raiz, en
              # el resto no se ignora

build/        # Se ignoran todos los ficheros en el directorio /build

doc/*.txt     # Se ignora el fichero doc/notes.txt, pero no
              # doc/server/arch.txt por ejemplo
```

17. Referencias

GIT trabaja con objetos, los cuales tienen un ID único que es un SHA1, con este ID, se puede hacer referencia a los objetos.

Cada commit es un objeto.

Los nombres de ramas, ramas remotas o etiquetas son referencias a commit.

Para hacer referencia a un commit, se puede emplear el ID completo o una subcadena del mismo formado por los 7 primeros dígitos del ID.

17.1. Referencias Simbólicas

También conocidas como symrefs, son punteros a punteros GIT, no apuntan al SHA-1 del objeto, sino al nombre del puntero.

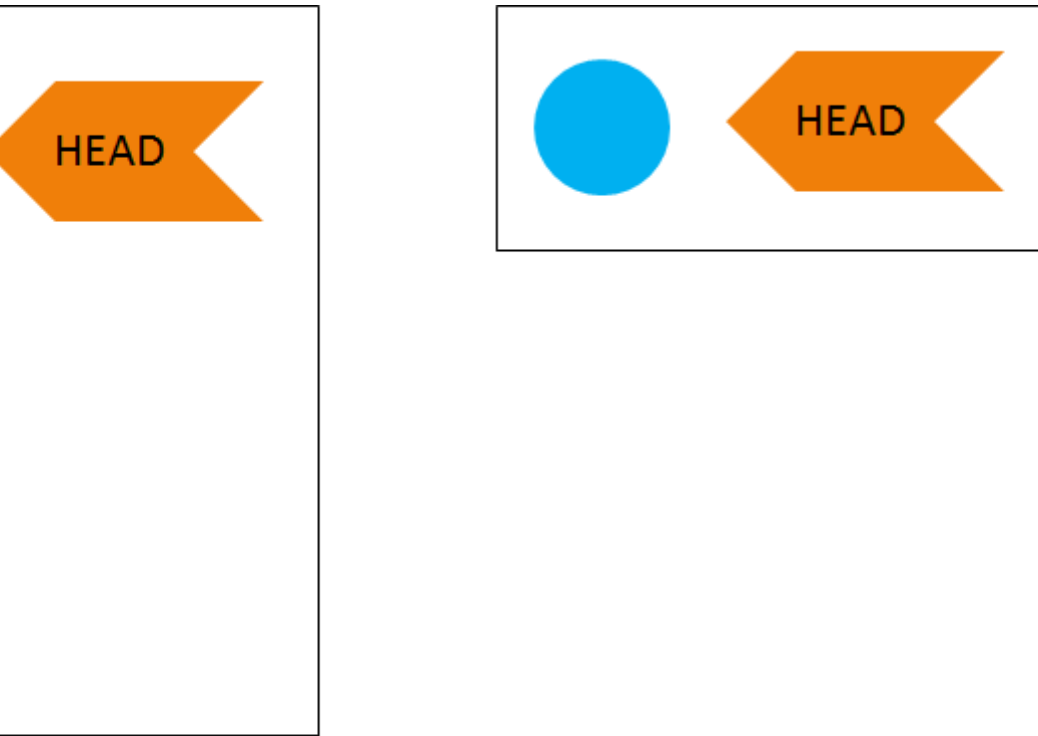
Las principales son

- **HEAD**. Apunta a un commit, o a una rama como master.
- **ORIG_HEAD**. Apunta al anterior commit al que apuntaba HEAD.

17.2. Head

Es un puntero que apunta por defecto al ultimo commit de la rama con la que se esta trabajando.





Es único por repositorio.

Se puede mover para que apunte a otro commit.

Se almacena en un fichero llamado HEAD del directorio .git, que tendrá como contenido

```
ref: refs/heads/master
```

Hay que tener cuidado con hacer commits cuando HEAD no apunta a una rama, dado que estaremos creando commit potencialmente huérfanos.

Para hacer un nuevo commit a partir de un commit antiguo, lo primero será crear una rama en ese commit antiguo y posicionar en esa rama el HEAD.

17.3. Master

Puntero a la rama principal, que si no se establece lo contrario, se llama master. Este puntero siempre apunta al último commit de la rama.

17.4. Referencias relativas

Se pueden emplear referencias relativas para referencias los commits, existiendo dos opciones.

- Con el acento circunflejo `\^`: Pudiéndose concatenar los `\^`, indicando cada `\^` un commit anterior a la referencia, en el caso anterior, seria el justamente anterior al último de la rama master, ya que master apunta al último.

```
> git show master^
```

- Con la virgulilla `\~`

```
> git show master~4
```

En este caso se hace referencia al cuarto commit para atrás.

17.5. Reglas de nombrado de referencias (Tag y Branch)

Para el nombrado de tags y branch, hay que seguir las siguientes premisas

- Se puede emplear la `V`, pero no al final.
- No están permitidos :
 - `\.` (punto)
 - `\~` (virgulilla)
 - `\^` (circunflejo)
 - `\:` (dos puntos)
 - `\?` (interrogación)
 - `*` (asterisco)
 - `\[` (corchete)

17.6. Referencias por mensaje de commit

Si no se desea manejar los SHA-1 para referenciar a un commit, se pueden emplear también los mensajes del commit para hacerle referencia, empleando la expresión de control



```
> git checkout :/"issue-1"
```

Con este comando se hace checkout del primer commit que se encuentre con la cadena **issue-1**, dentro del mensaje del commit.

18. Comandos Avanzados

18.1. Rev-parse

Comando de bajo nivel, que permite conocer cual es el SHA-1 del ultimo commit de una rama

```
> git rev-parse <identificador de la rama>
```

18.2. Amend

El modificador **--amend** del comando **commit**, permite añadir mas cambios al último commit añadido.

```
> git commit --amend
```

Este comando permite modificar el comentario asociado al commit, partiendo de dicho mensaje, una edición.

En realidad hace un commit nuevo, que sustituye en la linea temporal el anterior, quedando el anterior commit desreferenciado (huerfano)

18.3. Clean

El comando clean permite limpiar el Working Area, de ficheros que no estén manejados por GIT.

El motivo para emplear este comando, será la eliminación de archivos temporales generados por herramientas intermedias, que meten ruido en el Working Area a la hora de preparar un commit o bien ficheros generados que no se deseen.

El comando sin modificadores, no hace nada


```
> git clean
```

Por defecto, no se eliminan aquellos ficheros que están incluidos en .gitignore.

Modificadores

- **-d**: Incluye los subdirectorios.
- **-f**: Fuerza la eliminación.
- **-x**: Incluye entre los ficheros a borrar los incluidos en .gitignore.
- **-i**: Entra en el modo interactivo.
- **-n**: No borra nada, pero muestra lo que haría, la lista de tareas a realizar, sin aplicarlas.

18.4. Revert

Permite deshacer un commit.

Aplica los pasos inversos a lo que indica el commit, creando un nuevo commit con ellos.

```
> git revert <identificador del commit>
```

Si se revierte la creación de un fichero, se crea un nuevo commit, en el que se borra el fichero.

18.5. Rebase

Un rebase, se produce, cuando en dos ramas que van avanzando en paralelo, se quieren fusionar, no haciendo un merge, sino incluyendo los commit de una rama en la otra.

El caso mas típico para un rebase, es cuando mientras un desarrollador va incluyendo nuevas funcionalidades en su rama de desarrollo, otros desarrolladores, van actualizando la rama principal (master), y el desarrollador o bien porque necesita esos desarrollos, o bien porque no quiere extender el tiempo en el que las ramas avancen en paralelo para no complicar el futuro merge, decide hacer un rebase.



El procedimiento será colocarse en la rama que se desarrolla

```
> git checkout <identificador de rama>
```

Realizar el rebase sobre la rama master, o sobre aquella que se quiera incluir dentro de esta rama de desarrollo

```
> git rebase master
```

Cuando hay conflictos, se emplea

```
> git rebase master --continue
```

Se consigue de esta forma, incluir en la rama de desarrollo, los commit que en ese momento no están en ella y si en la rama master, tomándose dichos commit como la base (primeros commit) de la rama, luego a efectos prácticos, es como si la rama de desarrollo, no empezase donde empezó, sino en el ultimo commit que tenga en el momento del rebase la rama master.

Estado de las ramas avanzando en paralelo.

Graph	Description
	ramarebase Quinto commit(ramarebase)
	master Cuarto commit(master)
	Tercer commit(master)
	Segundo commit(ramarebase)
	Primer commit(ramarebase)
	README

Estado de las ramas con el rebase realizado

Graph	Description
	ramarebase Quinto commit(ramarebase)
	Segundo commit(ramarebase)
	Primer commit(ramarebase)
	master Cuarto commit(master)
	Tercer commit(master)
	README

La secuencia a realizar cuando la rama master del Repositorio Remoto es modificada, es

```
> git checkout master  
> git pull origin master  
> git checkout <rama de desarrollo>  
> git rebase master
```

Esta secuencia se tendrá que repetir tantas veces como sea necesario, y dependerá de lo que se modifique la rama master.

La principal ventaja de realizar esta secuencia, será la minimización de la complejidad de los merge.

18.6. Blame

Se emplea para inspeccionar un fichero, conociendo a que commit corresponde cada línea introducida, mostrando también el usuario que realizó la modificación.

```
> git blame <fichero>
```

Esta comando es útil cuando detectado un problema en un fichero, se desea saber cuando (commit) y quien (usuario) introdujo el bug.

Cuando los ficheros son muy grandes y no se sabe en que punto se ha producido el problema buscado, se puede emplear el modificador **-L**

```
> git blame -L <línea desde>,<línea hasta> <nombre de fichero>
```

18.7. Bisect

Cuando los ficheros son muy grandes y no se sabe, ni de forma aproximada, en que línea esta el problema, se puede realizar una búsqueda binaria.

Se emplea básicamente para buscar cuando se introdujo un bug en el código.

Lo único que se necesita saber, es como reproducir el bug (test) y un commit en el que no estuviera el bug y otro en el que si.

El comando para empezar la búsqueda



```
> git bisect start
```

Se ha de marcar el ultimo commit sobre el que se desea buscar, en el que se ha detectado el error, pero que no se sabe si se ha introducido en ese commit dicho error.

```
> git bisect bad <identificador de commit>
```

Es habitual que el commit malo, sea el actual, ya que se acabará de descubrir, por lo que no es necesario indicar el identificador del commit.

Se ha de marcar el primer commit sobre el que se desea buscar, el commit mas reciente sobre el que se sabe feacientemente que no exisita el error, puede ser el primer commit de la rama o un commit en el que la funcionalidad que falla no estuviera todavia implementada.

```
> git bisect good <identificador de commit>
```

El resultado de la operación, es un informe, con el numero de commit realizados entre esos dos commit y el identificador del commit central.

A partir de ese punto, se puede comprobar si para el commit central, se reproduce el bug, o no.

- Si se reproduce. El bug se introdujo antes, por lo que el commit central, se define como el ultimo (malo)

```
> git bisect bad
```

- Si no se reproduce. El bug se introdujo después, por lo que el commit central, se define como el primero (bueno)

```
> git bisect good
```

El procedimiento se repite hasta que solo haya un commit, y será en ese donde este el problema.

Cuando se finaliza la búsqueda se recompone el índice HEAD

```
> git bisect reset
```

19. Etiquetas

Son punteros a un commit específico.

No son modificables, es decir son punteros fijos.

Si se quiere mover, hay que borrar el tag y luego volverla a crear.

Se pueden tener ramas y etiquetas que se llamen igual.

Para listar los tag

```
> git tag
```

Para crear un nuevo tag en el commit actual

```
> git tag -a "nombre de la etiqueta" -m "Descripción del Tag"
```

Para crear un nuevo tag en otro commit

```
> git tag -a "nombre de la etiqueta" -m "Descripción del Tag"  
<identificador del commit>
```

Para borrar un tag

```
> git tag -d <nombre de la etiqueta>
```

Existen dos tipos de tags.

- **Ligeros.** Solo son un puntero, sin información extra.
- **Anotados:** Los que se crean con el modificador -a. Contienen información de quien lo creo, además de poder ser firmados y verificados.

Las etiquetas al estar asociadas a una rama, se descargan directamente en caso de realizar un clone de un Repositorio Remoto.

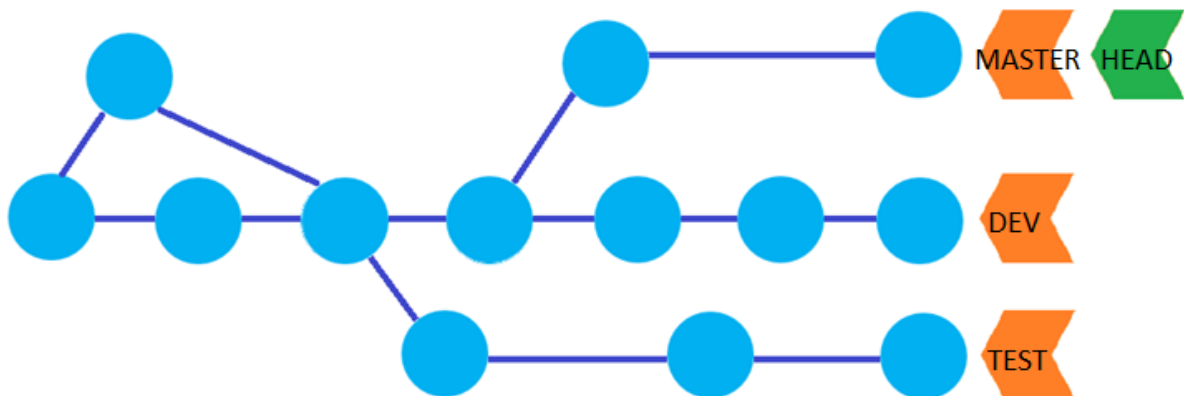


20. Ramas

Una rama es una línea de desarrollo distinta.

Permiten diferencias la evolución de un desarrollo en distintos caminos.

Estos caminos pueden volver a confluir en algún punto.



Se pueden tener tantas ramas como se quieran.

Las ramas locales, no tienen porque subirse al repositorio remoto.

Solo puede estar activa una rama, esto se consigue haciendo que HEAD apunte a la rama.

branch es el comando que permite mostrar las ramas del proyecto, y en la que estamos trabajando actualmente (marcada con *).

```
> git branch
```

Las ramas Track o de solo lectura (las que se obtienen con el comando fetch), no aparecen listadas con este comando, para listarlas.

```
> git branch -a
```

También permite crear nuevas ramas en el commit actual.

```
> git branch <nombre de la rama>
```

O en un commit concreto

```
> git branch <nombre de la rama> <identificador del commit>
```

Con este comando se pueden eliminar ramas, habrá dos formas de borrarla, dependiendo del estado de la rama.

- Si ha sido fusionada con otra (merge)

```
> git branch -d <nombre de la rama>
```

- Si no ha sido fusionada con otra, habrá que forzar el borrado

```
> git branch -D <nombre de la rama>
```

Cuando se borra una rama, en realidad se borra solo el puntero, los commit permanecen.

Se pueden filtrar la ramas visualizadas para ver

- Las fusionadas con la rama activa

```
> git branch --merged
```

- Las no fusionadas con la rama activa

```
> git branch --no-merged
```

El comando ** permite visualizar los commit de una o varias ramas.

```
> git show-branch <nombre de la rama>
```

Pero no es muy claro, es mejor el comando log.

20.1. Merge

Permite fusionar los cambios que hay en dos o mas ramas distintas



```
> git checkout <nombre de la rama actual>  
> git merge <nombre de la rama con la que se quiere fusionar la actual>
```

Realiza una fusión a tres bandas entre los dos commit a fusionar y el ancestro común.

La fusión se queda en la rama actual sobre la que se está trabajando.

Las ramas fusionadas, deben estar en el mismo repositorio.

Se recomienda que el Working Area y el Staging Area estén limpios antes de hacer el merge.

Los Merge puede dar dos resultados

- Ningún conflicto, por lo que se realiza de forma automática la fusión haciéndose un commit.
- Conflicto, no se puede realizar la fusión de forma automática por lo que la persona tiene que resolver dichos conflictos, estos habrán quedado indicados en los ficheros con conflictos, con una sintaxis similar a

```
<<<<<<< HEAD  
  
=====  
  
>>>>>>> 123retg4556123e123f
```

Donde lo primero hasta ===== es lo que hay en la rama origen (HEAD) y después lo que hay en la otra rama.

Una vez editado el fichero y resuelto el conflicto, GIT detecta el resultado como un cambio en el repositorio, por lo que hay que proceder con el de la misma forma que cuando el cambio se produce por el desarrollo normal.

```
> git add .  
> git commit -m "Solucionado el Merge"
```

Cuando se hace un Merge entre dos ramas, donde solo ha habido avance en una de las ramas, es decir la representación sigue siendo lineal

Graph	Description
	Segundo commit en la NuevaRama
	Primer commit en la NuevaRama
	Segundo commit en la rama master
	Primer Commit con README

No se produce un nuevo commit, sino que únicamente se mueve el puntero master hasta la posición de NuevaRama.

Graph	Description
	Segundo commit en la NuevaRama
	Primer commit en la NuevaRama
	Segundo commit en la rama master
	Primer Commit con README

Si se quiere forzar a que se haga el commit, y que los commit de NuevaRama, no se trasladen también a master, hay que incluir el modificador `--no-ff`

```
> git merge --no-ff
```

Obteniendo el siguiente resultado

Graph	Description
	Merge branch 'NuevaRama'
	Segundo commit en NuevaRama
	Primer commit en NuevaRama
	Segundo commit en la rama master
	Primer commit con README

21. Repositorios

Existen dos tipos de repositorios

- **Bare:** Repositorio normalmente Remoto, sobre el que no se puede realizar commits, solo push, fetch, pull, etc.
- **Development:** Repositorio Local típico de desarrollo, donde realizar los commits. Es recomendable siempre que la colaboración entre distintos desarrolladores, se

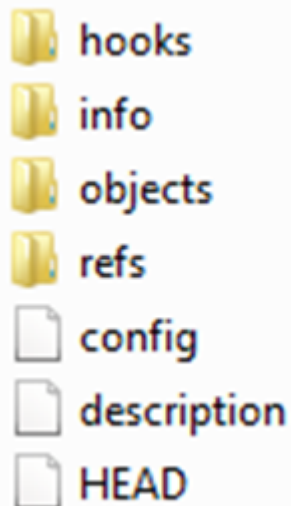


haga a través de un repositorio bare.

Los repositorios **Bare** se crean con

```
> git init <Nombre del Repositorio Remoto> --bare
```

Este comando crea una carpeta <Nombre del Repositorio Remoto>, con una estructura de carpetas propia de un Repositorio Remoto.



Una vez creado dicho Repositorio, se puede incluir como Repositorio Remoto en cualquier proyecto de GIT.

```
> git remote add <nombre del repositorio remoto> "path del repositorio remoto"
```

Solo falta dar acceso a la carpeta del repositorio bare al resto de desarrolladores, para que interaccionen con el.

21.1. Github

Es un hosting GIT que ofrece repositorios públicos gratuitos y privados de pago. se puede acceder desde [aquí](#)

Dentro de Github, se pueden crear

- Repositorios.
- Organizaciones (Grupos de usuarios).

Al crear un repositorio, se puede definir

- Publico.
- Privado (Conlleva incluir datos del pago).
- Incluir fichero README por defecto.
- Añadir el fichero .gitignore.
- Añadir una licencia.

Una vez creado, se genera una URL por la cual se puede clonar el repositorio.

Es el Hosting Git mas empleado, en el mundo del Software Libre.

Permite definir colaboradores del repositorio, con permisos para hacer push directamente en el repositorio.

Permite hacer Forks del proyecto, para realizar cambios en un repositorio paralelo y proponer para fusionar los cambios a través Pull-Request.

21.2. Bitbucket

Producto de Atlassian, empresa referente en herramientas de ecosistemas agiles. Se puede acceder [aquí](#)

Ofrece cuentas gratuitas con un número ilimitado de repositorios privados, limitando eso si, el numero de colaboradores a 5 por repositorio en el caso de las cuentas gratuitas.

Se pueden crear

- Repositorios.
- Equipos.

21.3. Clone

Para crear el proyecto como un clon de uno remoto

```
>git clone <url del repositorio> <nombre del directorio del proyecto>
```

Este comando crea un directorio de proyecto, dentro crea el directorio .git y descarga todos los ficheros del proyecto en todas sus versiones.



Si no se establece el nombre del repositorio, se coge el del repositorio remoto.

Cuando se clona un repositorio remoto, en local, se crea un alias del repositorio llamado por defecto **origin**.

También se crea un puntero en el Repositorio Local, que apunta al commit de la ultima sincronización con el Repositorio Remoto, llamado **<alias del repositorio remoto>/<rama>**

Cuando se clona un repositorio, por defecto se clona solo la rama **master**.

Para clonar otra rama, se emplean los mismos comandos que tenemos para el trabajo con ramas en local, pero añadiendo la referencia al Repositorio Remoto.

```
>git checkout -b <nombre de la rama local> <alias repositorio remoto>/<rama remota>
```

Que nos crea la rama y nos posiciona en ella

```
>git branch <nombre de la rama local> <alias repositorio remoto>/<rama remota>
```

Que solo crea la rama, sin posicionarse en ella.

21.4. Remote

Es el comando que permite interaccionar con los repositorios locales que están conectados con uno remoto.

Para listar los repositorios remotos, con url y las acciones disponibles sobre dichos repositorios.

```
>git remote -v
```

Para acceder a la ayuda del comando

```
>git remote -h
```

Para renombrar el alias del repositorio remoto

```
>git remote rename <nombre antiguo> <nombre nuevo>
```

Para añadir un repositorio remoto a un proyecto GIT ya existente

```
>git remote add <alias del repositorio remoto> <url del repositorio>
```

Con este comando podemos tener varios repositorios remotos asociados a un mismo proyecto.

Para borrar un repositorio remoto del proyecto local

```
>git remote rm <alias del repositorio remoto>
```

Las url de los repositorios, pueden incluir el usuario y el password, si fuesen necesarios

```
http://<user>:<password>@<host>:<port>/<path>
```

Se puede inspeccionar el repositorio remoto

```
>git remote show <alias del repositorio remoto>
```

21.5. Push

Permite subir (empujar) a una rama concreta en un Repositorio Remoto, el commit actual.

```
>git push -u <alias del repositorio remoto> <rama local>:<rama remota>
```

De no indicarse el Alias, se toma por defecto origin.

De no establecer la rama se toma por defecto master.

De solo indicar una rama, se toma ese nombre tanto para la local como para la remota.

Si se intenta hacer un push sobre un Repositorio Remoto que tiene otro commit, no



contemplado en el Repositorio Local desde el que se esta haciendo el push, este es desestimado, es decir, hay que tener el Repositorio Local actualizado respecto el Remoto para poder empujar (push) nuevos cambios (commits), por lo que los merge se han de realizar en local.

Se pueden subir todas las etiquetas al repositorio remoto

```
>git push <alias del repositorio remoto> --tags
```

O solo alguna concreta

```
>git push <alias del repositorio remoto> <nombre del tag>
```

Para borrar una rama remota

```
>git push <alias del repositorio remoto> :<nombre de la rama>
```

Nótese que delante de los :, no hay nada, es decir se esta subiendo la rama local <vacío> a la rama remota

21.6. Fetch

Permite sincronizarse con el Repositorio Remoto, para actualizar en el Repositorio Local cambios que haya introducido otro usuario del Repositorio Remoto.

```
>git fetch <Alias del Repositorio Remoto> <Rama Remota a sincronizar>
```

Lo que se descarga, lo hace en una rama distinta, de tipo Track (solo lectura) cuyo nombre es **<Alias del Repositorio Remoto>/<Rama Remota a sincronizar>**

Por lo que la fusión (Merge) queda pendiente.

21.7. Pull

Realiza un Fetch y además un Merge.

```
>git pull <Alias del Repositorio Remoto> <rama remota>:<rama local>
```

Emplearemos este comando cuando se tenga claro que hay que mezclar lo que hay en el Repositorio Remoto en el Repositorio Local.

Lanzar este comando siempre desde la rama a empujar.

21.8. Fork

Los Fork, se producen en algunos repositorios remotos, que implementan este mecanismo, como Github, cuando se clona un repositorio de un usuario A en otro usuario B.

Este mecanismo permite que el usuario B, que no es propietario del repositorio original, pueda seguir evolucionándolo de forma paralela.

Habitualmente este mecanismo se acompaña de los **Pull-Request**.

21.9. Pull-request

Los **Pull-Request**, son peticiones que realiza un usuario B, a un usuario A, donde el usuario B ha evolucionado una copia de un repositorio original del usuario A.

La petición consiste en que el usuario B, quiere colaborar con el A, en la elaboración del repositorio.

Es un mecanismo que permite dos cosas:

- La primera es que el usuario B evoluciona un repositorio, sin tener permisos explícitos para ello
- Y la segunda es que el usuario A, tiene en todo momento el control sobre lo que se incorpora a su repositorio, dado que es el quien finalmente realiza el commit.

El procedimiento será el siguiente

- El usuario B
 - Realiza el fork del repositorio original, lo cual se realiza en el servidor de repositorios remotos (Github)
 - Se descarga en local su repositorio, la copia.
 - Evoluciona el repositorio con los commit que desee.
 - Publica los commit en su repositorio remoto.



- ↪ Realiza la petición de Pull-Request al usuario A.
- El usuario A
 - ↪ Crea una rama en local con el nombre del usuario B que realizo el Pull-Request.
 - ↪ Descarga la rama evolucionada desde el repositorio del usuario B hacia la rama en local recién creada.
 - ↪ Valida la evolución y en caso de aceptarla la mezcla con la rama original de su repositorio.
 - ↪ Publica los cambios en el repositorio original, con lo que se cierra el Pull-Request.

22. Parches

Un parche, es un conjunto de cambios generados en un Repositorio Local que se desean aplicar a un Repositorio Remoto, es decir son un conjunto de commit, que están en Local y se desean trasladar a Remoto.

Para generarlo se ejecuta

```
> git format-patch -M <rama remota de referencia>
```

Es comando genera un archivo **.patch**, para cada commit, entre el HEAD del proyecto Local y la Rama Remota tomada como referencia.

Estos ficheros se distribuirán al servidor/es Remotos y allí se aplicarán los parches.

Para aplicar los parches, lo normal, es hacerlo sobre una rama nueva y posteriormente realizar el merge.

Las opciones son en comando apply

```
> git checkout -b <rama para aplicar los parches>  
> git apply <path del archivo patch>
```

Este se puede emplear para parches generados con **diff** y con **format-patch**.

Este comando es atómico, si se pueden realizar todos los cambios, los hace, y sino lo aborta todo.

Modifica el Working Area incluyendo los cambios descritos en el parche, pero no hace commit.

Si antes de aplicar se desea chequear si hay conflictos, se puede ejecutar con el modificador **--check**

Y el comando am

```
> git checkout -b <rama para aplicar los parches>
> git am <path del archivo patch>
```

Este comando realiza el commit, con la información obtenida del archivo patch, siempre que este haya sido generado con **format-patch**.

En caso de problemas, este comando permite realizar tres cosas.

- Resolverlo, como si fuera un merge, ya que se incluyen marcadores en los ficheros de la misma forma, y posteriormente marcarlo como resuelto

```
> git add <fichero con el conflicto resuelto>
> git am --resolved
```

- Saltarse el parche

```
> git am --skip
```

- Resetear al estado anterior

```
> git am --abort
```

Al aplicar un solo parche las dos ultimas opciones son iguales.

23. Herramientas

23.1. Gitk

Herramienta visual que viene incluida en la distribución de Git, permite visualizar los commit, las ramas, las etiquetas.



También da acceso a otra aplicación Git Gui, desde donde se puede controlar el ciclo de vida de los ficheros de forma gráfica.

23.2. DiffMerge

Herramienta que permite realizar de forma visual los Merge entre ficheros.

La pagina oficial de la herramienta, se encuentra [aquí](#)

La instalación es sencilla, simplemente seguir las instrucciones.

Para que GIT, emplee esta herramienta para las diferencias, hay que configurar

```
> git config --global diff.tool diffmerge  
  
> git config --global difftool.diffmerge.cmd "C:/Program\  
Files/SourceGear/Common/DiffMerge/sgdm.exe \"%LOCAL%\" \"%REMOTE%\""
```

NOTE

A destacar el empleo de secuencia de escape para el espacio en blanco \ * y las comillas interiores *\"

Para que GIT, emplee esta herramienta para los Merge, hay que configurar

```
> git config --global merge.tool diffmerge  
  
> git config --global mergetool.diffmerge.trustExitCode true  
  
> git config --global mergetool.diffmerge.cmd "C:/Program\  
Files/SourceGear/Common/DiffMerge/sgdm.exe -merge -result=\"%MERGED%\"  
\"%LOCAL%\" \"%BASE%\" \"%REMOTE%\""
```

23.3. SourceTree

Herramienta de Atlassian gratuita para trabajar con GIT en Windows y en Mac.

Esta herramienta es un cliente para trabajar con repositorios locales Git o Mercurial y remotos como Github, Bitbucket o Stash.

Permite

- revisar los cambios de los repositorios.

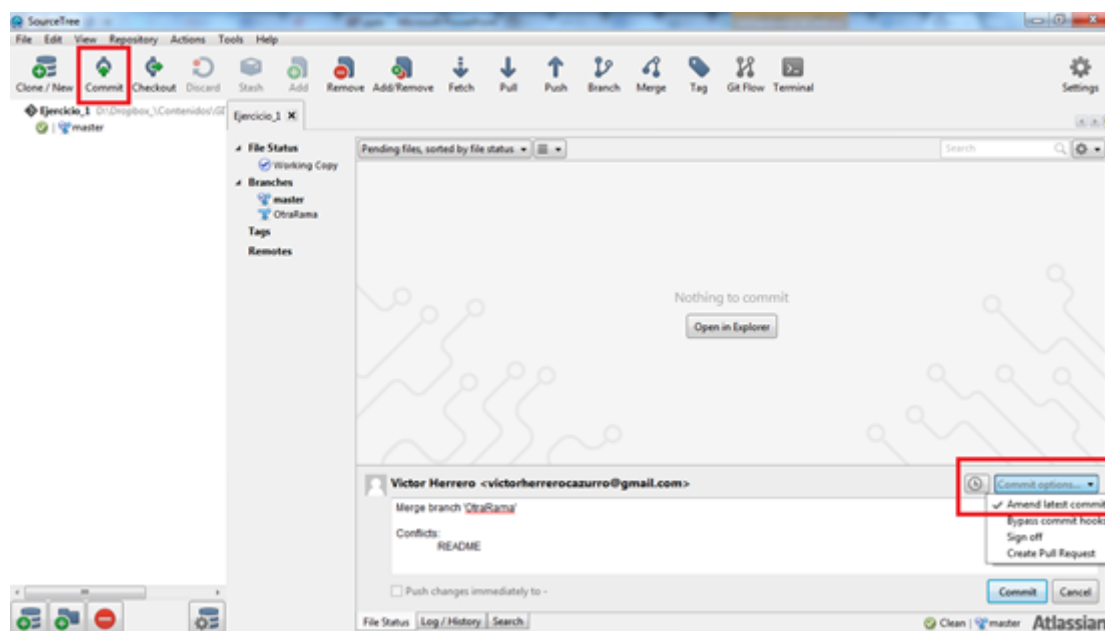
- manejar de forma sencilla los distintos branch.
- visualizar claramente cada modificación en los gráficos de cada rama.

Por ejemplo, para hacer un **amend** en SourceTree, recordemos que por línea de comandos sería.

```
> git commit --amend
```

Seleccionar el botón de commit.

En la ventana de mensaje en un seleccionable marcar **amend**



23.4. SmartGIT

Herramienta similar a la anterior, que es gratuita para proyectos personales, no comerciales, que como característica diferencia, permite visualizar los commit que no están referenciados por un rama, es decir los "perdidos" (Lost Heads)

24. Comandos de fontanería

24.1. Reflogs

Comando que muestra todos los movimientos de referencias que se han producido.

El resultado que ofrece, puede ser difícil de leer.



Hay un reflog general (de HEAD)

```
> git reflog
```

y un reflog por cada rama

```
> git reflog <nombre de rama>
```

Al borrar una rama, se pierde el reflog.

Para borrar un reflog

```
> git reflog expire --expire-unreachable-now <nombre de rama> --all
```

- **expire**: indica que se quieren borrar
- **unreachable**: indica que no esta siendo referenciado por nadie
- **now**: indica que expire desde ahora, sin esta opcion, guarda los de los ultimos 90 dias por seguridad
- **all**: indica que se aplique a todos los reflog, el general y el de los branch

No se hace un push del reflog, cada repositorio tiene el suyo.

Es posible que en un momento dado cuando se modifican las referencias, se **pierdan** commit.

```
> git reset -hard <identificador de commit>
```

En estos casos, es interesante el uso del comando **reflog**.

```
> git reflog
```

Este comando lista las tareas realizadas en el repositorio, indicando el identificador del commit a donde llevo dicho comando, lo cual permitiría volver a un commit no referenciado.

Para volver al commit, lo habitual es hacerlo en un branch nuevo

```
> git branch <nombre de nueva rama> <identificador del commit perdido>
```

24.2. Garbage Colector

Elimina objetos del disco duro

Empaqueta y optimiza el espacio en disco.

Cando se hace un push, se ejecuta el recolector de basura, para evitar que al empujar al servidor, se envíen datos de mas.

La condicion para que **gc** borre el commit es que no existan referencias a commit de tipo branch, tag o entrada del reflog.

```
> git gc
```

La ejecución del recolector de basura no es instantanea.

24.3. Garbage Colector

Muestra los commit que están sin resolver, que son aquellos que no están referencias a commit de tipo branch, tag o entrada del reflog.

```
> git fsck --lost-found
```

Muestra los últimos de las ramas

Para verlos todos

```
> git fsck --lost-found --unreachable
```

25. Bibliografia

Documentación Oficial de GIT <http://git-scm.com/book/en/v2>

Tutoriales de Atlassian <https://www.atlassian.com/git/tutorials>

Pro-Git (Ed. Apress) <http://www.apress.com/9781484200773>



