

CONTROL DE VERSIONES

¿Qué es y para qué sirve?

Git es un sistema de control de versiones libre y distribuido diseñado para gestionar proyectos. Su objetivo es controlar y gestionar una enorme cantidad de ficheros de forma fácil y eficiente.

Se basa en repositorios que se inicializan en un directorio concreto y tienen toda la información de los cambios realizados.

Partes de un repositorio

El repositorio remoto puede ubicarse en GitHub, GitLab, Bitbucket, SourceForge, entre otros. Es una copia del repositorio de git que se encuentra alojado en un servidor o en un sitio distinto al local.

El repositorio local se divide en:

- **Directorio de trabajo** (working directory): donde se almacena el contenido del repositorio.
- **Área de preparación** (staging area o index o cached): donde están indicados los cambios de confirmación.
- **Repositorio local** (local repository): donde están todas las versiones de manera local.

Estados de un fichero

- **untracked**: Son ficheros que existen en el área de trabajo pero no existen para Git. *No han sido añadidos al seguimiento del repositorio.*
- **staged**: Son ficheros que se han añadido al *staging area* y están listos para ser incluidos en el siguiente commit. *Añadidos con `git add`.*
- **committed**: Son ficheros que se guardaron en el último commit y que no han sido modificados desde entonces. *Están en el repositorio y no tienen cambios*

pendientes.

- **modified**: Son ficheros que se han modificado desde el último commit pero que aún no se han añadido al *staging area*. *Han sido editados en el área de trabajo, pero Git aún no los ha preparado para el commit.*
- **ignored**: Son ficheros que Git ignora debido a las reglas definidas en el archivo `.gitignore`.

Comandos básicos para gestionar ficheros en Git

1. **Añadir un fichero al staging area:**

```
git add fichero
```

2. **Confirmar los cambios del staging area al repositorio:**

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

3. **Incluir en el commit los cambios en los ficheros *staged* y también en los *modified*:**

```
git commit -a -m "mensaje"
```

4. **Quitar un fichero del staging area (opuesto a `git add`):**

```
git restore --staged fichero
# Alternativa menos usada:
git reset fichero
```

5. **Quitar un fichero del staging area pero mantenerlo en el área de trabajo:**

```
git rm --cached fichero
```

6. **Borrar un fichero del staging area y del área de trabajo:**

```
git rm fichero
```

7. **Restaurar un fichero modificado en el área de trabajo al estado del último commit:**

```
git restore fichero
# Alternativa menos usada:
git checkout fichero
```

Comprobación del estado de los ficheros:

Para ver el estado de los ficheros en las diferentes áreas (untracked, staged, modified):

```
git status
```

El comando muestra:

- Archivos en la staging area (cambios a ser confirmados).
- Archivos modificados pero no añadidos al staging area.
- Archivos sin seguimiento (untracked).

Ejemplo de salida:

```
En la rama main
Cambios a ser confirmados:
  (usa "git restore --staged <archivo>..." para sacarlos del área de stage)
  nuevos archivos: facturas.html

Cambios no rastreados para el commit:
  (usa "git add <archivo>..." para actualizar lo que será confirmado)
  (usa "git restore <archivo>..." para descartar los cambios en el directorio d
  modificados:      main.js

Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlos en el área de stage)
  cliente.html
```

Notas importantes:

- Si modificas un fichero después de usar `git add` y no vuelves a añadirlo con `git add`, el commit solo incluirá los cambios que estaban en el staging area cuando ejecutaste `git add`.

- Para evitar este problema, se puede usar el comando `git commit -a`, que automáticamente añade los ficheros modificados al staging area antes de confirmar los cambios.

Sincronización de Git y Ramas

Una misma rama en git puede estar en varios sitios a la vez. El repositorio es el histórico donde están todos los commits que se han hecho.

Explicación de las ramas suponiendo que estamos en `master`:

- Carpeta local o `área de trabajo` o árbol de trabajo: Es donde están los ficheros con lo que trabajamos nosotros directamente mientras programamos.
- `Repositorio local`: Es el repositorio que tenemos en nuestro ordenador.
 - `Rama master`: Es donde se guardan los ficheros al hacer commit
 - `Rama origin/master`: Es una copia que hay del repositorio remoto.
- `Repositorio remoto`: Es un repositorio que está en otra máquina.
 - `Rama master`: Es donde queremos copiar los commits de nuestra rama master. Al hacer referencia a una rama de un repositorio remoto, se usará la expresión `origin master` con un espacio en medio.

Si queremos movernos entre ramas:

```
git switch master
git checkout master # Comando más antiguo, se recomienda switch
```

Si queremos crear una nueva rama:

```
git branch <nombreRama>
```

Si queremos crear una rama y automáticamente movernos a ella:

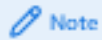
```
git switch -c <nombreRama>
git checkout -b <nombreRama> # Comando antiguo, se recomienda switch
```

Si queremos movernos a un commit en concreto con `switch`:

```
git switch --detach <hashDelCommit>
```

Si queremos movernos a una rama de tipo `origin/rama` debemos hacerlo con:

```
git switch --detach origin/rama
```



DETACH nos informa de que no se puede crear commit al lugar al que nos movemos. Es decir, no podemos trabajar en ese lugar

Si queremos publicar los cambios de una rama:

```
git push origin master  
git push
```

Si queremos obtener los cambios del remoto:

```
git fetch origin master  
git fetch  
  
# FETCH TIENE LA OPCIÓN --PRUNE PARA BORRAR LAS CARPETAS QUE HAN SIDO BORRADAS |  
git fetch --prune
```

Si queremos obtener y aplicar en nuestra área de trabajo los cambios del remoto:

```
git pull
```



`git pull` hace un fetch y un merge, lo que puede producir conflictos que posteriormente haya que solucionar y puede que el histórico se modifique.

Para `merge` lo que hay en la rama `origin/master` en la rama `master`:

```
git merge origin/master
```

Si queremos eliminar una rama en local:

```
git branch -d <rama>
```

Si queremos eliminar una rama en remoto:

```
git push origin -d <rama>
```

Para fusionar ramas tenemos dos opciones:

```
git merge <nombreRama>  
git rebase <nombreRama>
```

¿Diferencia entre Merge y Rebase?

Merge conserva la historia original de la rama fusionada y **rebase** reescribe la historia de la rama para que parezca que los commits se aplicaron de forma lineal. En otras palabras, **merge** fusiona e incorpora los cambios especificados a la rama actual (*donde está el HEAD*) mientras que **rebase** mueve el HEAD a la rama especificada.



HEAD es donde estamos ubicados.

Remotes

En Git un "remote" hace referencia a un servidor de Git donde subimos el código. Por defecto cuando hacemos git clone se crea un remote llamado origin.

Si queremos ver la lista de remotes:

```
git remote -v
```

Para añadir un remote:

```
git remote add otroservidor <URL>
```

Borrado de commits

`GIT RESET` modifica el estado actual de un fichero.

Tienes diferentes opciones:

- `--soft`: Vuelve a un commit y conserva los cambios en el directorio de trabajo y en el stage.

```
git reset --soft <hashCommit>
```


- `--mixed`: Los cambios se conservan en el stage.

```
git reset --mixed <hashCommit>
```

- `--hard`: Los cambios no se conservan.

```
git reset --hard <hashCommit>
```

	Borra los commits siguientes	Se modifica área de trabajo	Modifica staged area
--hard	Si	Si	Si
--mixed	Si	No	Si
--soft	Si	No	No

 **Note**

La opción `HEAD~1` hace referencia al último commit.


Deshacer commits

`GIT REVERT` deshace un commit y crea uno nuevo volviendo al estado del commit especificado.

```
git revert HEAD~1
git revert <hashCommit>
```

- Prepara un nuevo commit que deshace el último commit pero sin realmente hacer el commit. Lo que hace es modificar los ficheros y añadirlos a la staged area. Después se hace el commit. La opción `--no-commit` es útil porque nos permite revisar que va a hacer el commit y nos permite hacer alguna modificación más. Si no estamos interesados en hacer el commit lo podemos abortar con `git revert --abort` o usar el ya conocido `git restore --staged --worktree fichero`.

```
git revert --no-commit HEAD~1
git commit -am "Mensaje de commit" # -a para añadir a stage & -m para el mensaje
```

 **Note**

`git restore --staged --worktree fichero` restaura el archivo `fichero` tanto en el área de preparación como en el directorio de trabajo a su estado en el último commit, eliminando cualquier cambio realizado.

Configuración

`.gitkeep`

Por defecto Git no sube las carpetas vacías, por lo que si queremos que las suba hay que añadir algún fichero. Por convención se suele crear un fichero llamado `.gitkeep` y al ya haber algún fichero, git subirá esa carpeta.

`.gitignore`

El fichero `.gitignore` permite indicar que carpetas no se deben subir a git. El fichero debe estar en el raíz del proyecto de git.

Almacenando contraseñas

Normalmente trabajamos siempre en nuestro ordenador y no queremos todas las veces volver a poner la contraseña. En ese caso se puede decir a git que la almacene y no nos la vuelva a pedir:

- No te vuelve a pedir la contraseña en 15 min

```
git config --global credential.helper cache
```

- Almacena la contraseña en el disco en texto plano y nunca mas te la vuelve a pedir.

```
git config --global credential.helper store
```

Stash

Es un almacén que permite guardar los cambios de los que todavía no se quiere hacer un commit.

- Guardar solo lo modificado (estén o no en el staged area):

```
git stash push
```

- Guarda todos los ficheros aunque NO estén en la stage area. Es decir, también los ficheros nuevos. Y al hacer el pop se restaura el estado en el staged area.

```
git stash push --include-untracked
```

	Por defecto	--include-untracked	--all
Los ficheros modificados (estén o no en el <i>staged area</i>)	✓	✓	✓
Los ficheros nuevos		✓	✓
Los ficheros los ignorados			✓

- Y si luego queremos recuperar los cambios se hace con:

```
git stash pop --index # Si no se añade --index, no se restaurará el staged area
```

Log

El comando git log permite ver el histórico de commits.

- Ver los commits en una única línea cada commit.

```
git --no-pager log --pretty=oneline
```

- Ver los commits indicando que campos queremos mostrar e indicando el formato de la fecha.

```
git --no-pager log --pretty=tformat:"%h %cn %cd %s" --date=format:"%d/%m/%Y %H:
```

Pretty format

```
git log --pretty=format:"%H"
```

See the next tables on format variables.

Commit

%s	commit subject
%F	commit subject, filename style
%b	commit body
%d	ref names
%e	encoding

Author and committer

Author

Name	author
%an	author, respecting mailmap
%aI	author email, respecting mailmap
%aD	author date (rfc2822)
%ar	author date (relative)
%at	author date (unix timestamp)
%ai	author date (iso8601)

Committer

Name	committer name
%cn	committer name, respecting mailmap
%ce	committer email
%cE	committer email, respecting mailmap
%cD	committer date (rfc2822)
%cr	committer date (relative)
%ct	committer date (unix timestamp)
%ci	committer date (iso8601)

Hash

Commit	
%H	commit hash
%h	(abbrev) commit hash
Tree	
%T	tree hash
%t	(abbrev) tree hash
Parent	
%P	parent hash
%p	(abbrev) parent hash

Presets

Date		%a	Sun	Weekday
%a/%d/%Y	06/05/2013	%A	Sunday	
%A, %B %e, %Y	Sunday, June 5, 2013	%w	0-6 (Sunday is 0)	
%b %e %a	Jun 5 Sun	%y	13	Year
Time		%Y	2013	
%H:%M	23:05	%b	Jan	Month
%I:%M %p	11:05 PM	%B	January	
Used by Ruby, UNIX date, and many more.		%m	01.12	
		%d	01.31	Day
		%e	1.31	

Time

%l	1	Hour
%H	00.23	24h Hour
%I	01.12	12h Hour
%M	00.50	Minute
%S	00.60	Second
%p	AM	AM or PM
%Z	+08	Time zone
%j	001.306	Day of the year
%c	%	Literal % character

Mensajes de commit

Los mensajes de commit deben seguir el siguiente formato:

```
type(#issue):titulos  
  
Explicación (opcional)
```

Siendo

- **type**
 - **feat**: Si se añade una nueva funcionalidad (feature)
 - **fix**: Si se arregla (fix) un error
 - **docs**: Si solo cambia cosas de la documentación
 - **style**: Si solo se cambia el estilo del código como tabuladores, puntos y comas, formateo, etc.
 - **refactor**: Si se cambia el código para mejorar su calidad pero sin modificar la funcionalidad. Eso se llamada refactorizar.
 - **test**: Si se cambian cosas relacionadas con test automáticos.
 - **chore**: Si se cambian cosas relacionadas con el despliegue.
- **#issue** : Es el N° de la incidencia a la que hace referencia.

Ejemplos:

- Se arregla el fallo 45 que se llama "Falla si la fecha está vacía"

```
fix(#45):Falla si la fecha está vacía
```

- Se añade la nueva funcionalidad llamada "Mostrar el listado de los pacientes" con N° 456

```
feat(#456):Mostrar el listado de los pacientes
```