GIT 1 - Control de versiones

▼ **Índice**

- Sistema de control de versiones
  - Tipos de sistemas de control de versiones
    - Local
    - Centralizado
    - Distribuido
  - GIT
    - Instantáneas, no diferencias.
    - **Tiene integridad (checksum).**
    - **Grafo dirigido acíclico.**
    - Ciclo de vida de un archivo
- Trabajando en local
  - Guardando cambios en el repositorio
  - Ramas
  - Viendo el histórico
  - Log
  - Viendo las diferencias
  - Integrando los cambios de ramas
    - Merge
    - Rebase
  - Rebase vs Merge
    - **Merge**
    - Rebase
- Trabajando con repositorios remotos
  - Remotos
    - Clonar un repositorio
    - Mostrando tus repositorios remotos
    - Añadiendo un repositorio remoto
  - Actualizando el entorno de desarrollo
    - Fetch
    - Pull

        - Push
        - Eliminar rama remota
        - Forma natural de trabajo
- Deshaciendo cosas
    - Deshaciendo la modificación de un archivo
    - Deshaciendo la preparación de un archivo
    - Limpia los archivos que no están bajo el control de la versión en tu directorio de trabajo
    - Revertir un commit
    - Deshacer un commit
    - Head vs Detached Head
- Reescribiendo la historia
    - Modificar la última confirmación de cambios
    - Reorganizando el trabajo realizado
    - Deshacer un commit
- Otros conceptos útiles
    - Tags
    - Cherry pick
    - Hooks
    - Alias
    - Configuración
    - Stash
    - Otros
- Flujos de trabajo:
    - Mensajes de commits
- Bibliografía

Sistema de control de versiones

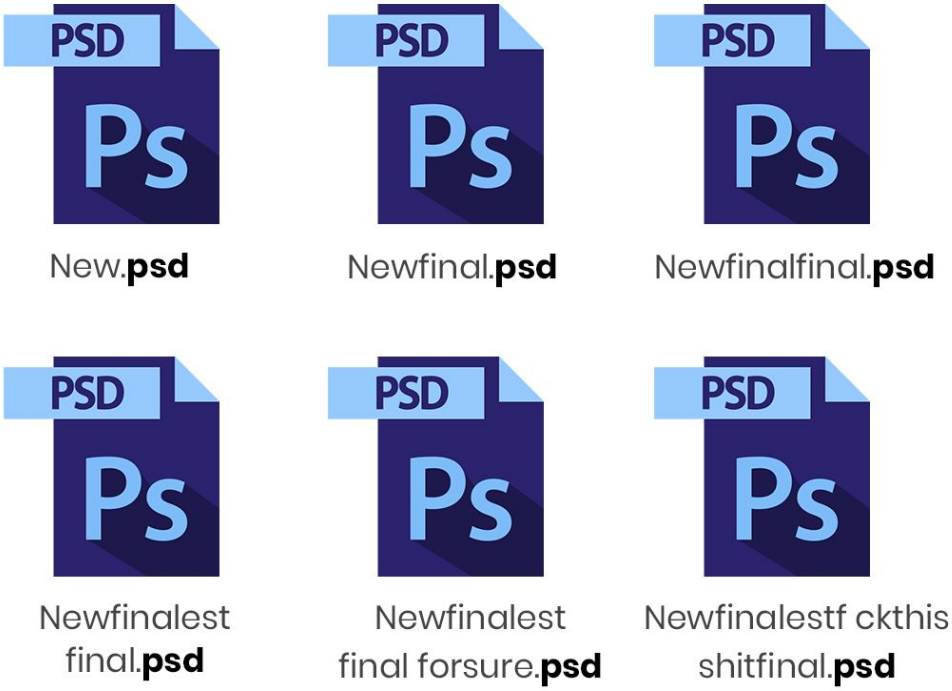
Herramienta que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.

Dicho sistema te permite regresar a tus versiones anteriores de los archivos, regresar a una versión anterior de tu proyecto, comparar cambios en los ficheros a lo largo del tiempo, también nos permitirá recuperar fácilmente archivos en caso de perdida.

Tipos de sistemas de control de versiones

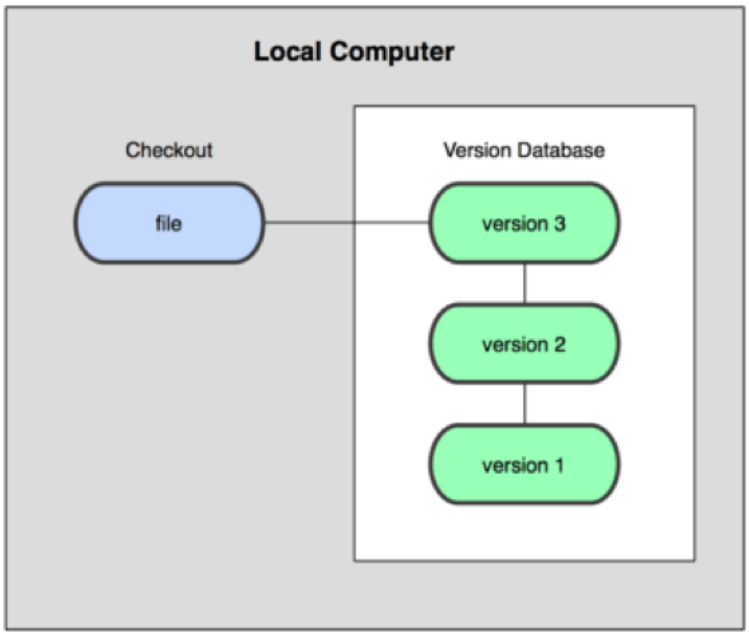
Local

Un método de control de versiones, usado por muchas personas, es copiar los archivos a otro directorio (quizás indicando la fecha y hora en que lo hicieron, si son ingeniosos). Este método es muy común porque es muy sencillo, pero también es tremendamente propenso a errores. Es fácil olvidar en qué directorio te encuentras y guardar accidentalmente en el archivo equivocado o sobre-escribir archivos que no querías.



A raíz de esto nacen los VCS(Version Control System) locales, que contenían una base de datos en la que se iban registrando todos los cambios realizados a los archivos

Los cambios son guardados localmente y no se comparten con nadie. Esta arquitectura es la antecesora de las dos siguientes.



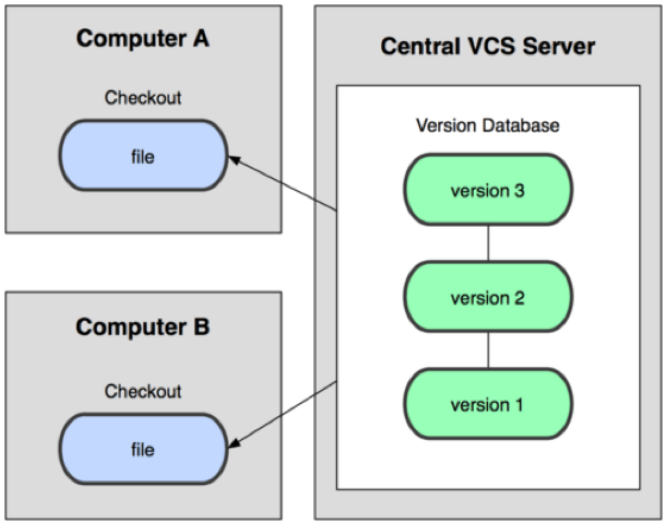
Centralizado

Problema: El VCS local no nos permite colaborar con otras personas, porque como su propio nombre indica es local. ¿Que pasa cuando varios desarrolladores han de colaborar en un mismo proyecto?
Como **solución** aparecieron los VSC centralizados

Todos nuestros fuentes y sus versiones están almacenados en un único directorio de un servidor(**repositorio centralizado).** Todos los desarrolladores que quieran trabajar con esos fuentes, deben pedirle al sistema de control de versiones una copia local para trabajar. En ella realizan todos sus cambios y cuando están listos y funcionando, le dicen al sistema de control de versiones que guarde los fuentes modificados como una nueva versión. Ejemplos: CVS y Subversión.

Problemas:

- Si el servidor se cae, los desarrolladores no van a poder salvar las versiones.
- Si se corrompe la base de datos, se pierde todo el historico de cambios.



Distribuido

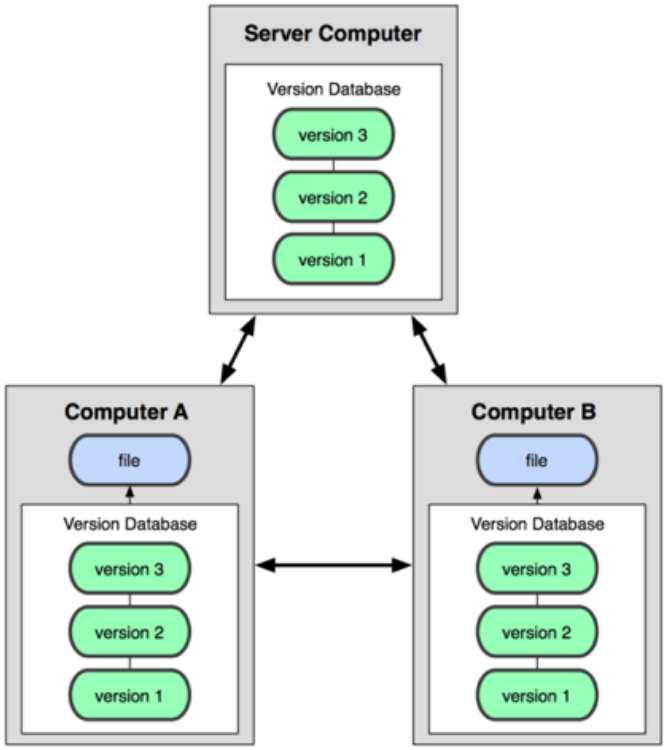
No hay un repositorio central. Todos los desarrolladores tienen su propia copia del repositorio, con todas las versiones y toda la historia.

De esta manera, si un servidor deja de funcionar y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios disponibles en los clientes puede ser copiado al servidor con el fin de restaurarlo. Cada clon es realmente una copia completa de todos los datos. Podemos seguir trabajando en caso de que se caiga el servidor

- **Casi cualquier operación es local.**
    - Unido a lo comentado anteriormente entre las diferencias entre el VCS centralizado vs distribuido. La mayoría de las operaciones en Git sólo necesitan archivos y recursos

locales para funcionar. Por lo general no se necesita información de ningún otro computador de tu red.

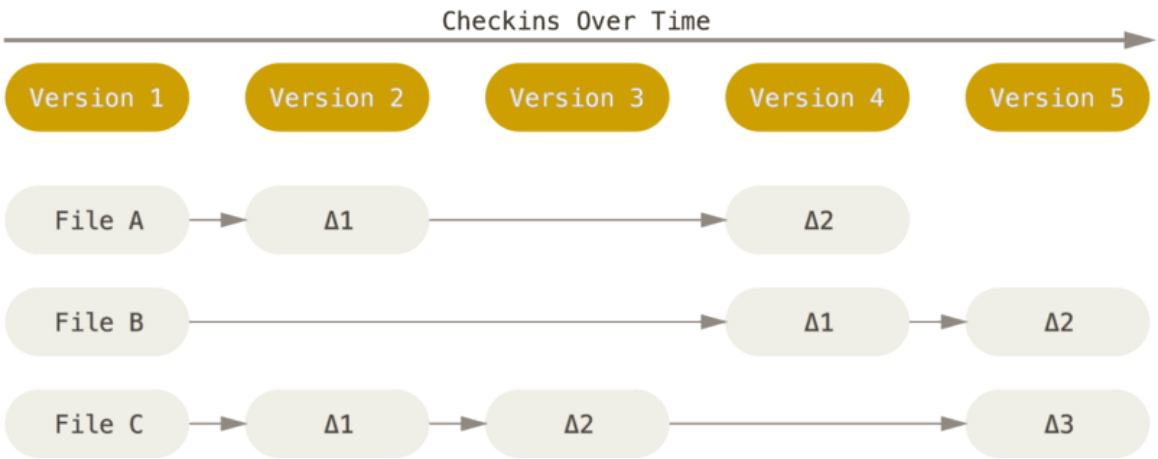
- En Subversion y CVS puedes editar archivos, pero no puedes confirmar los cambios en el caso de que el servidor este caído.



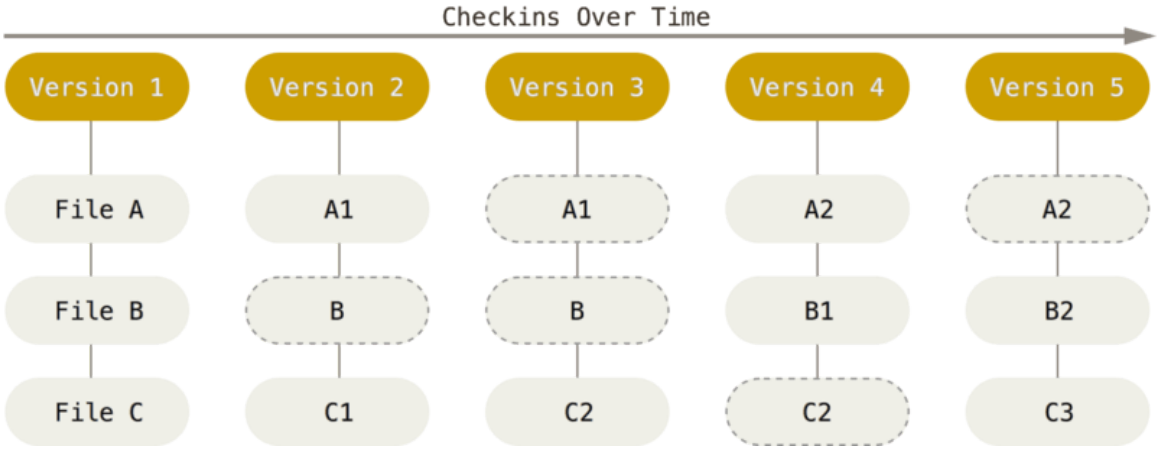
GIT

Instantáneas, no diferencias.

- La mayoría de los otros sistemas manejan la información que almacenan como un conjunto de archivos y las modificaciones hechas a cada uno de ellos a través del tiempo.



- Git maneja sus datos como un conjunto de copias instantáneas. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, básicamente toma una foto del aspecto de todos tus archivos en ese momento y guarda una referencia a esa copia instantánea. Para ser eficiente, si los archivos no se han modificado Git no almacena el archivo de nuevo, sino un enlace al archivo anterior idéntico que ya tiene almacenado.



Tiene integridad (checksum).

- Todo en Git es verificado mediante una suma de comprobación (checksum) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma.
- Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git sea capaz de detectarlo.

Grafo dirigido acíclico.

Commits (confirmaciones o instantáneas)

Un commit se puede asemejar a una fotografía (*capturar la apariencia visual de un objeto, persona, paisaje en un momento determinado*). Llevado a nuestro campo, podemos afirmar que un commit es una fotografía o instantánea que muestra el estado de tu proyecto en un momento determinado (en el momento en el que creaste el commit o confirmación) la cual se almacena en un histórico.

▼ Ejemplo con imágenes

KATHY COLLINS / GETTY IMAGES

¿Para qué sirven los commits?

Cuando usamos un sistema de control de versiones en un proyecto, uno de los objetivos que perseguimos es mantener un registro de los cambios que se han producido en el código a través del tiempo. No sólo queremos tener un backup de nuestros archivos, queremos poder mirar hacia atrás y ver cómo ha ido mutando o avanzando nuestro código. Para ello, Git nos va a mostrar un listado de los cambios que ha sufrido el código, es decir, una línea temporal donde iremos viendo diferentes puntos o hitos, cada uno de ellos acompañados de un mensaje descriptivo. Cada uno de esos hitos será un commit.

Ramas

Una rama representa una línea independiente de desarrollo. Las ramas sirven como una abstracción de los procesos de cambio, preparación y confirmación. Puedes concebirlas como una forma de solicitar un nuevo directorio de trabajo, un nuevo entorno de ensayo o un nuevo historial de proyecto. En Git las ramas son una forma de seguir desarrollando y programando nuevas funcionalidades o modificaciones del software sin afectar a la parte principal del proyecto.

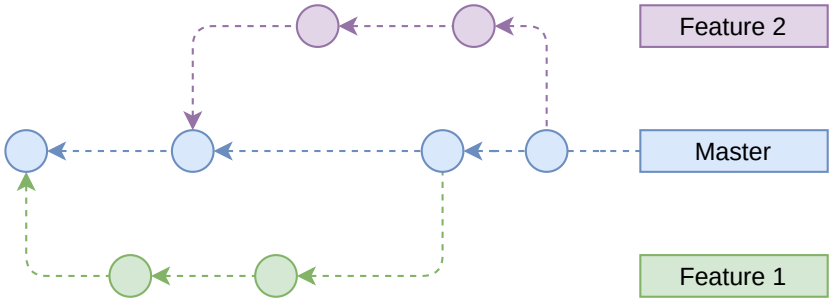
¿Para qué sirven las ramas?

Imaginemos que tenemos una pagina web publicada y el código de esa web está en la rama master (main) de nuestro repositorio. Si trabajamos directamente sobre la rama master, en este caso estaremos cambiando o introduciendo los cambios a la web. Si por un casual introducimos un bug, ese bug estará publicado directamente en la web y la gente podrá ver dichos fallos. Como somos humanos, tenemos fallos y no queremos que la gente vea esos fallos. Por cosas como estas surge el concepto de las ramas.

Lo cual nos permite crear un nuevo directorio de trabajo, donde iremos desarrollando la nueva funcionalidad y una vez dicha funcionalidad esté terminada y probada (para evitar posibles errores) la volcamos sobre la rama principal.

Las nuevas confirmaciones se registran en el historial de la rama actual, lo que crea una

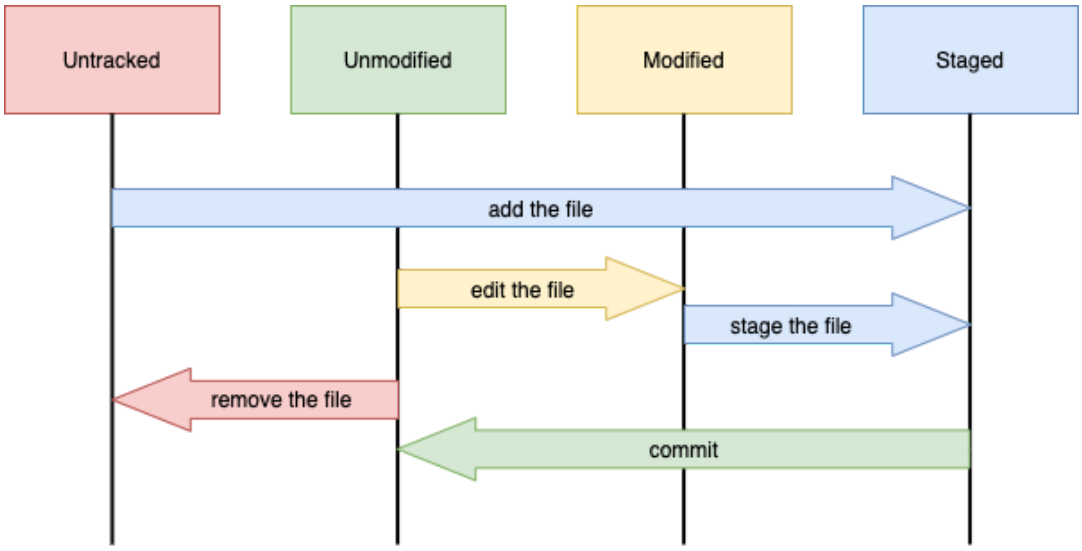
bifurcación en el historial del proyecto.



¿HEAD?

¿Cómo sabe Git en que rama estás trabajando? Pues…, mediante un apuntador especial denominado `HEAD`. Es un apuntador a la rama local en la que te encuentras en ese momento. `HEAD` contiene el identificador de la rama actual, que a su vez contiene el identificador del último commit realizado en la rama.

Ciclo de vida de un archivo

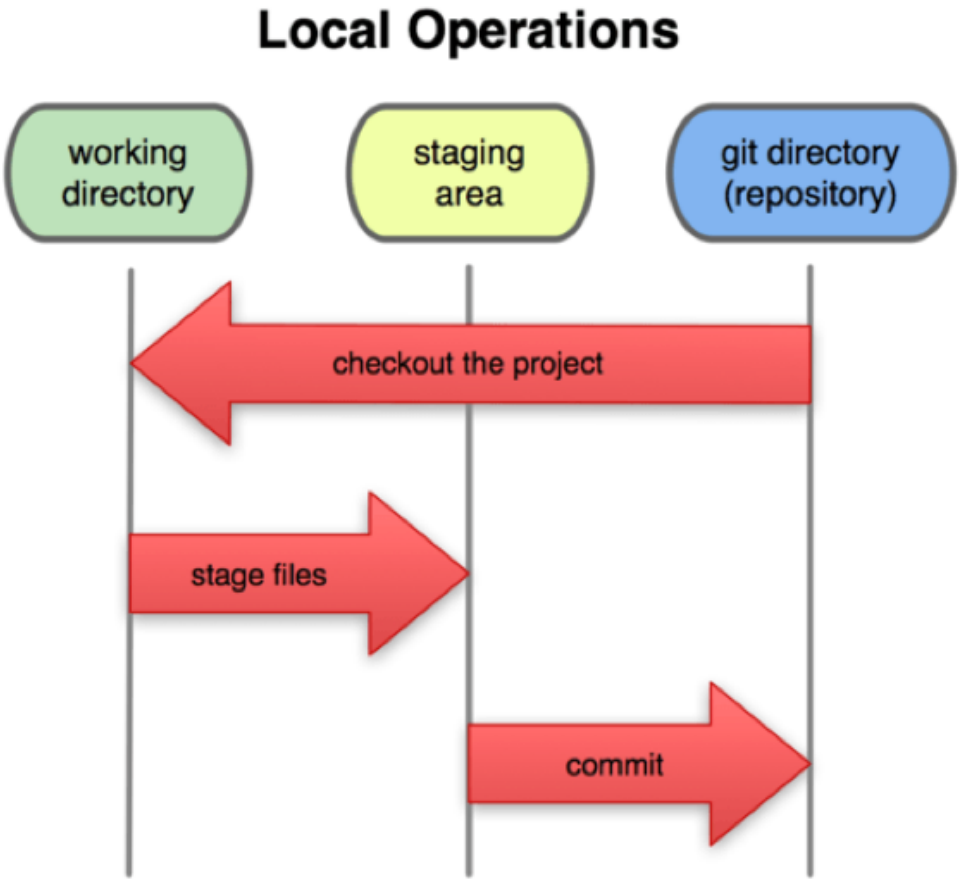


Tres estados:

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

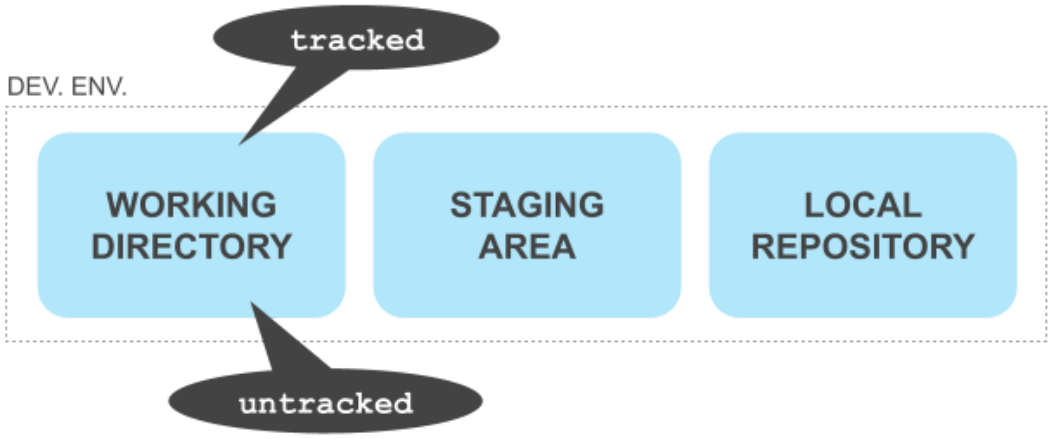
Tres áreas de trabajo:

- **Git directory**: almacena los metadatos y la base de datos de objetos para tu proyecto.
- **Working directory**: copia de una versión del proyecto.
- **Staging area**: almacena información acerca de lo que va a ir en tu próxima confirmación.



Trabajando en local

Guardando cambios en el repositorio



▼ Crear repositorio local

```
git init
```

▼ Preparar archivo modificado y ver qué pasa

```
vim README
```

▼ Seguimiento de nuevos archivos (añadir ficheros a git)

```
git add README

# git add <path>
# git add [--all][-A][.]
```

▼ Comprobar el estado de tus ficheros

```
git status [-s]
```

▼ Confirmar cambios

```
git commit -m "First commit" # genera un commit
git commit -am "First commit" # añade los cambios al staged area y genera un commit
```

▼ Eliminando archivos (git rm vs rm)

```
git rm <file>
```

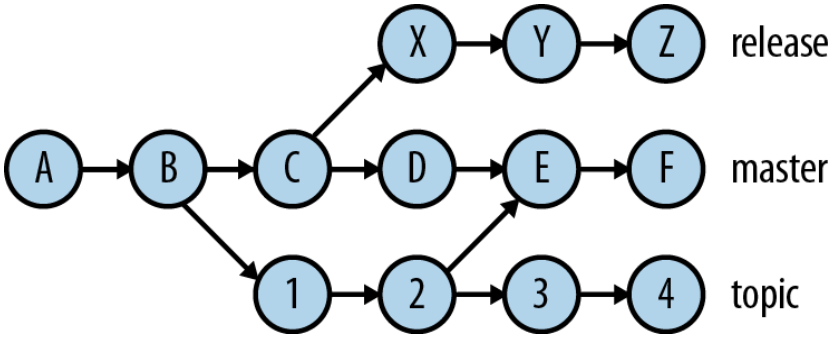
▼ Eliminar un fichero pero mantener una copia en local

```
git rm <file> --cached
```

▼ Ignorando archivos

```
vim .gitignore
#Ver https://www.gitignore.io/
```

Ramas



▼ Crear una rama

```
git branch <branchname>
```

▼ Borrar una rama

```
git branch -d <branchname>
```

▼ Re-nombrar una rama

```
git branch -m [branchname] new-branchname
```

▼ Moverse entre ramas

```
git checkout branchname
```

▼ Crear y moverse a la rama

```
git checkout -b branchname
```

Viendo el histórico

Log

▼ **Podemos ver el histórico de Instantáneas(commits) que hemos ido creando a lo largo del tiempo**

```
git log
```

▼ **Lol - Histórico de instantáneas resumido**

Log resumido

```
git config --global --add alias.lol "log --graph --decorate --pretty=oneline --abbrev-commit --all"
```

Viendo las diferencias

▼ **Mostrar los cambios de un fichero en el working directory**

```
git diff <file>
```

▼ **Mostrar los cambios de un fichero en staging area**

```
git diff --staged <file>
```

▼ **Mostrar los cambios de un fichero respecto a commit**

```
git diff <commit>
```

▼ **Comparar dos ramas**

```
git diff <branch1>..<branch2>
```

▼ **Diferentes configuraciones**

```
git diff [--numstat][--name-only]...
```

⚠️ **Práctica**
- Crear y añadir ficheros
- Hacer **commits** con diferentes cambios: Nuevos ficheros, borrar ficheros, editar ficheros.
- Crear diferentes **ramas** y commitear en ellas
- Moverse entre **ramas**
- Borrar y renombrar **ramas**
- Ver los **logs** de commits
- Ver **diferencias** entre commits y ramas

Integrando los cambios de ramas

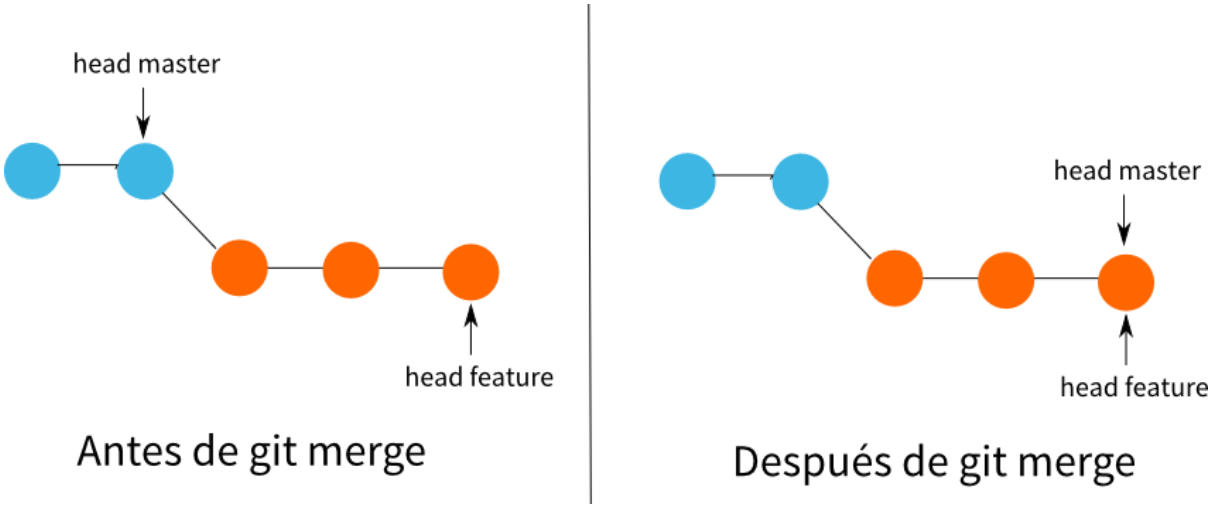
https://git-school.github.io/visualizing-git/

Merge

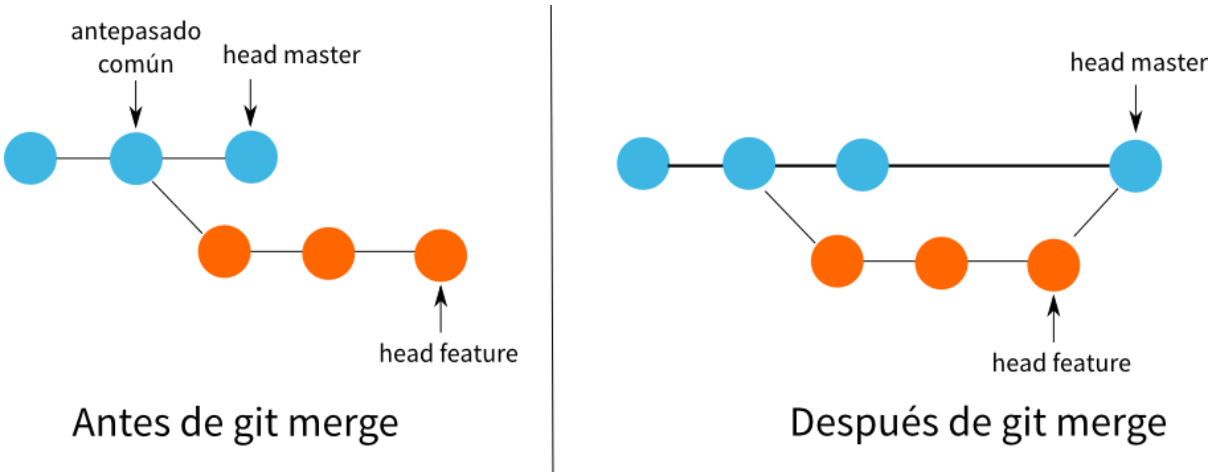
La fusión o merge es la forma que tiene Git de volver a unir un historial bifurcado. El comando git merge permite tomar las líneas o ramas independientes de desarrollo creadas por git branch e integrarlas en una sola rama.

```
git merge [--ff][--no-ff][--ff-only
```

Un fast-forward merge es cuando al momento de hacer merge con la rama master no se ha añadido ningún commit luego de crear la rama feature, es decir que el HEAD de master es el antepasado de la rama feature. Por tanto, en este caso no se genera un nuevo commit para agregar los commits de la rama de feature, en vez de ello, el HEAD de master se actualiza al HEAD de la rama feature, sin crear un commit de merge adicional, de allí su nombre, fast-forward o avance rápido, como se muestra en la imagen:



Por otro lado, si la rama master ha divergido después de haber creado la rama feature ya no es posible un fast-forward merge, debido a que el commit de la rama donde actualmente se está (master) no es un antepasado directo de la rama a fusionar (feature) por tanto, git realiza un merge a tres bandas, es decir, que genera un commit para fusionar las dos ramas.



Merge con conflictos

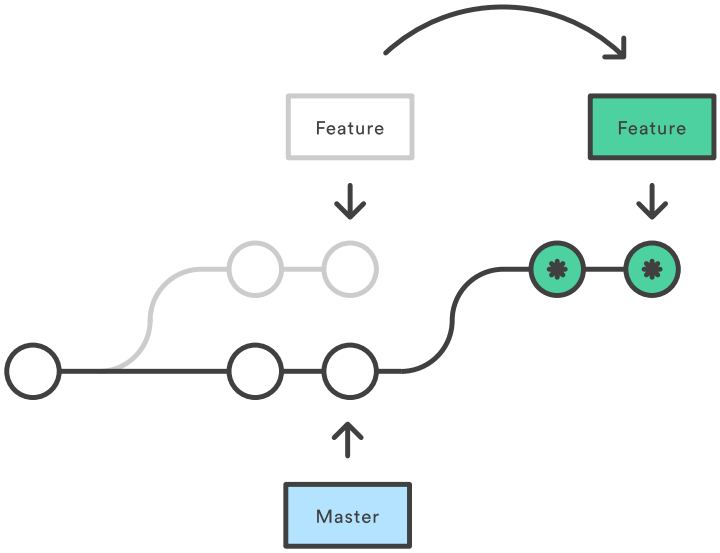
- Posibilidad de abortar un merge (durante el proceso de resolución de conflictos)

```
git merge --abort
```

Rebase

Rebase es una de las dos utilidades Git que se especializa en integrar cambios de una rama a otra.

La reorganización es el proceso de mover o combinar una secuencia de commits en un nuevo commit base. La reorganización es muy útil y se visualiza fácilmente en el contexto de un flujo de trabajo de ramas de funciones. El proceso general se puede visualizar de la siguiente manera:



Desde una perspectiva del contenido, la reorganización consiste en cambiar la base de tu rama de un commit a otro haciendo que parezca que has creado la rama desde otro diferente. Internamente, Git lo hace creando nuevos commits y aplicándolas a la base especificada. Es muy importante entender que, aunque la rama parece la misma, se compone de nuevos commits por completo.

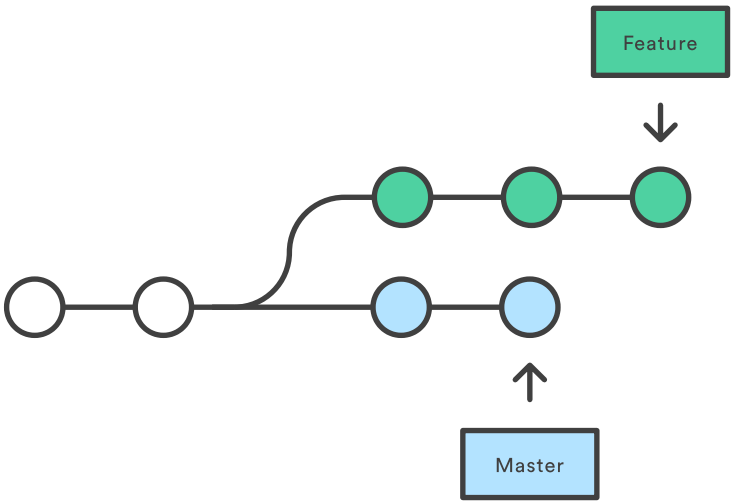
```
git rebase <branch-name>
```

Rebase vs Merge

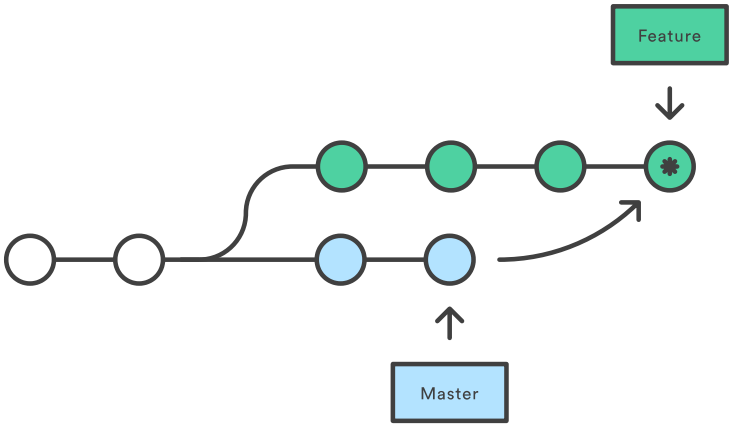
Ambos comandos están diseñados para integrar cambios de una rama a otra, pero lo hacen de forma muy distinta.

Merge

A forked commit history



Merging master into the feature branch



Merge Commit

La fusión o merge está bien porque es una operación no destructiva. Las ramas existentes no cambian en ningún aspecto.

En el ejemplo de la imagen se fusiona la rama principal(master) con la rama de función(feature) creando una confirmación de fusión que une los historiales de ambas ramas.

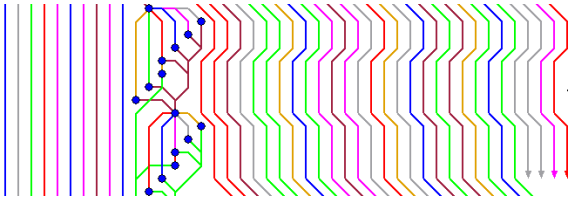
Ventajas

- Simple y Familiar.
- Mantiene la historia completa y en orden cronológico.

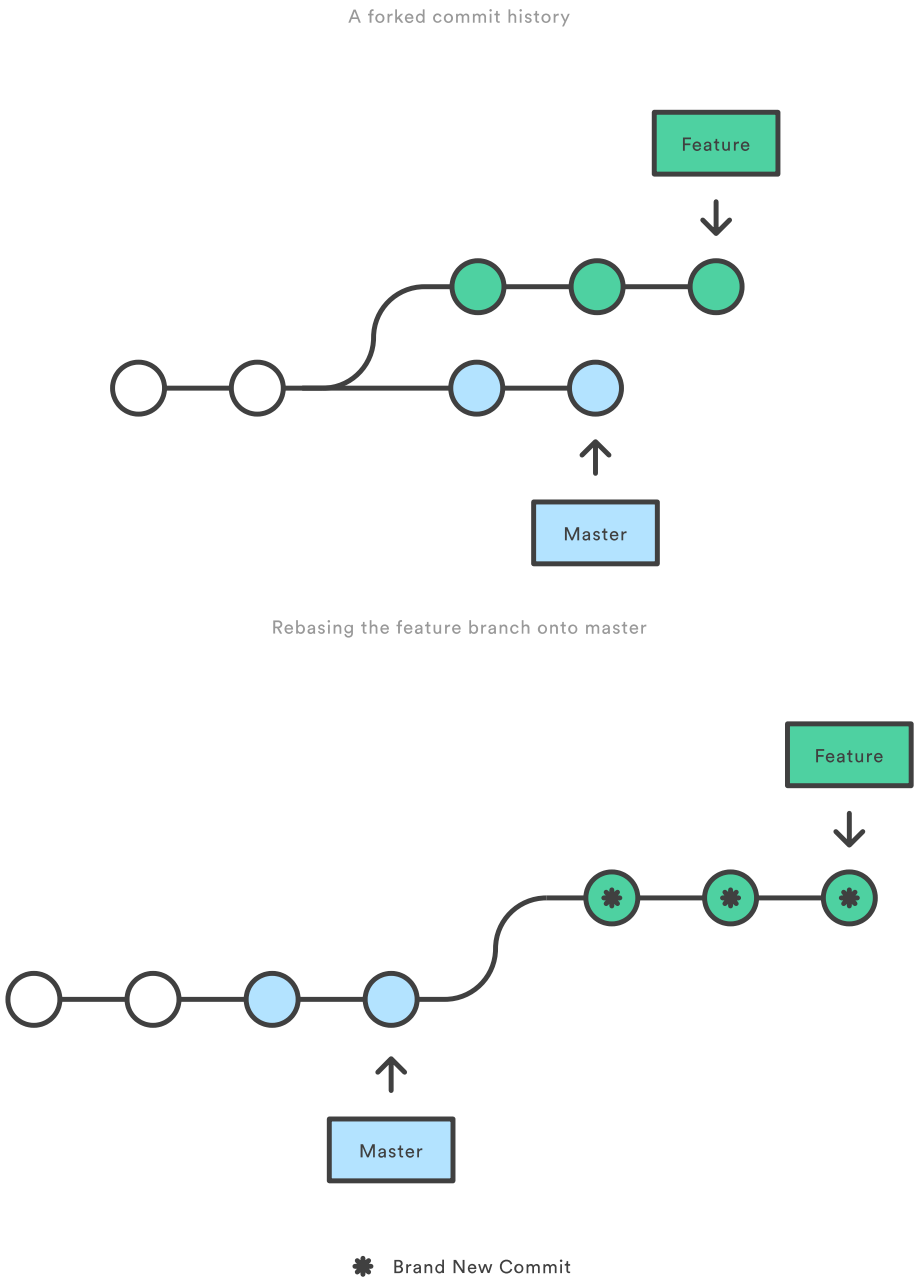
Desventajas

- Si la rama master es muy activa, esto puede contaminar bastante el historial de

tu rama de función, ya que se generará una confirmación de fusión externa cada vez que necesites incorporar cambios de nivel superior.



Rebase



La reorganización o rebase reescribe el historial del proyecto creando nuevas confirmaciones para cada confirmación en la rama original.

Ventajas

- Historial de proyecto muy limpio, perfectamente lineal.
- Navegación por el proyecto más sencilla

Desventajas

- Requiere utilizar **Force Push** al trabajar con ramas remotas **[Reescribir la historia]**
- El uso de rebase para mantener actualizada requiere resolver conflictos similares una y otra vez. Mientras se fusiona, una vez que resuelva los conflictos, estará listo. Debe resolver el conflicto en el orden en que se crearon para continuar con el rebase.

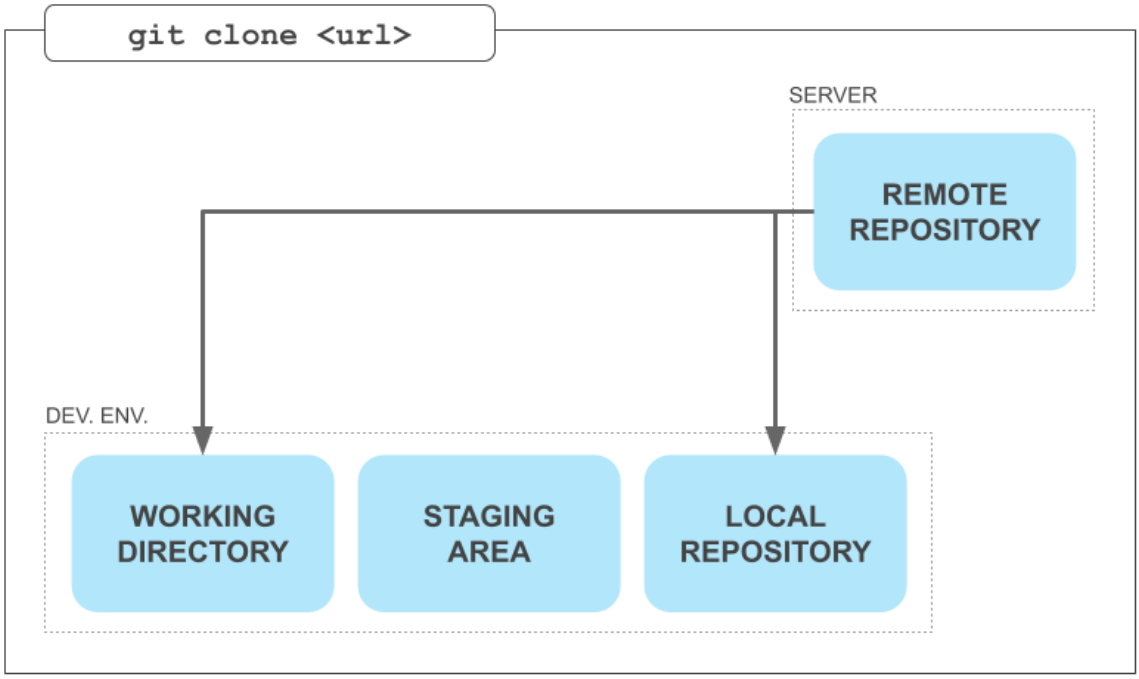
⚠️ **Práctica**
- Hacer un **merge**
- Hacer un **rebase**
- Forzar un **conflicto y abortar** el merge
- Forzar un **conflicto y resolverlo**

Trabajando con repositorios remotos

Remotos

Clonar un repositorio

```
git clone <remote-repositorie-url> <carpeta-destino>
```



https://rachelcarmena.github.io/2018/12/12/how-to-teach-git.html

💡 La primera vez que clonas un repositorio, todos tus archivos estarán bajo seguimiento y sin modificaciones, ya que los acabas de copiar y no has modificado nada.

Mostrando tus repositorios remotos

```
git remote -v
```

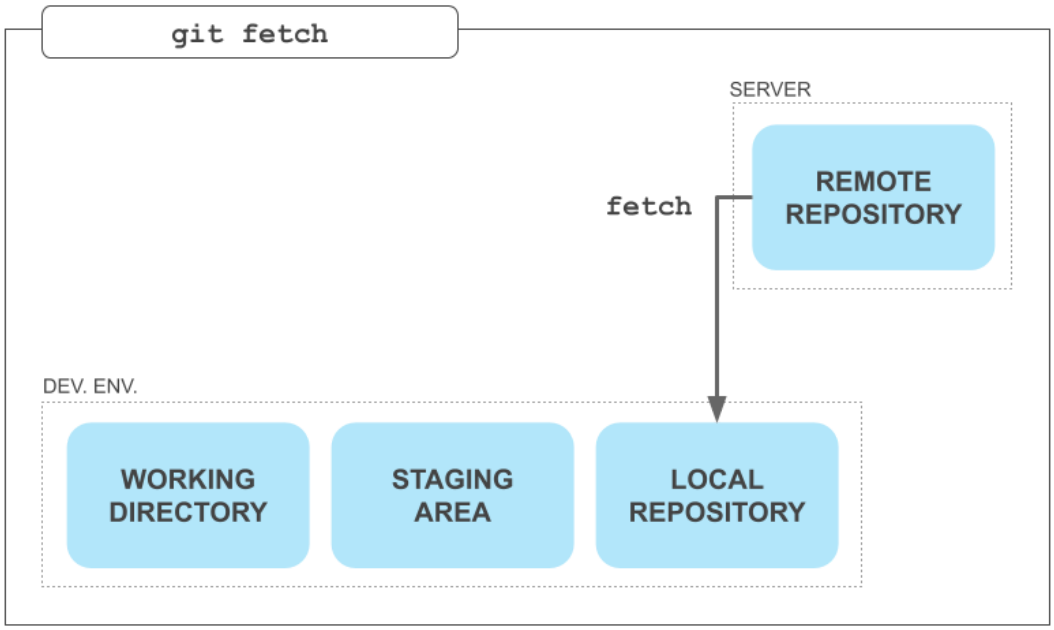
Añadiendo un repositorio remoto

```
git remote add <remote-name> <remote-repositorie-url>
```

Actualizando el entorno de desarrollo

Fetch

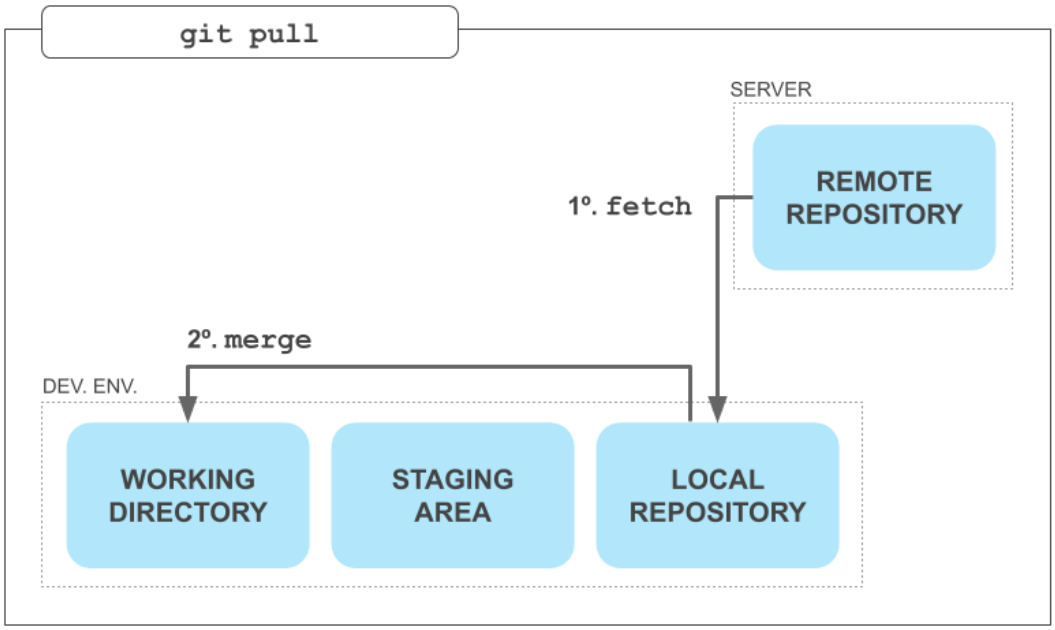
```
git fetch
```



https://rachelcarmena.github.io/2018/12/12/how-to-teach-git.html

Pull

```
git pull
```



https://rachelcarmena.github.io/2018/12/12/how-to-teach-git.html

Push

```
git push <remote> <branchname> [--force][-u]
```

Eliminar rama remota

```
git push <remote> :<branchname>
```

Forma natural de trabajo

Se puede hacer **pull** de cualquier rama en cualquier rama o **push** de cualquier rama en cualquier rama. Un **pull** se ha visto no es más que un *fetch+merge* con *fast-forward*. También puede ser *no-fast-forward* si se le indica `[--no-ff]` o incluso rebase `[--rebase]`

Lo normal es que haya una rama local unívocamente asociada a otra rama remota. En ese caso GIT ofrece facilidades:

```
## Rama local a publicar
git push <remote> <branchname> -u
git pull
git push

## Rama remota a descargar
git checkout <remote>/<branchname> --track
git pull
git push
```

⚠️ **Práctica (por parejas)**
- Crear repositorio vacío
- Añadir remoto y subir repositorio local (persona1), clonar repositorio en local (persona2)
- Añadir commits nuevos y pushear (persona1) y hacer pull (persona2)
- Pushear nueva rama (persona1) y hacer fetch (persona2)
- Eliminar rama remota

Deshaciendo cosas

Deshaciendo la modificación de un archivo

```
git checkout -- <filename>
```

Este comando reemplaza los cambios en tu directorio de trabajo (sin estar en el stage) con el último contenido de HEAD. Si no estaba trackeado no hace nada.

Deshaciendo la preparación de un archivo

```
git reset HEAD <file>
git reset [--mixed][--soft][--hard]
```

--mixed: deja los cambios en el working directory (tienes que estar en el stage). Es el que hace por defecto.

--soft: deja los cambios en el stage

--hard: elimina todo (de todo)

Limpia los archivos que no están bajo el control de la versión en tu directorio de trabajo

```
git clean [-f][-d][-n]
git clean [-f][-d][-n] <file>
```

-n: realizará un simulacro de git clean. Podremos ver los archivos que se van a eliminar sin eliminarlos

-f: indica la eliminación real de los archivos sin seguimento del directorio actual (obligatorio para realizar el borrado)

-d: indica a git clean que también se han de eliminar los directorios sin seguimiento, ya que este comando ignora los directorios de forma predeterminada.

Revertir un commit

```
git revert <commit>
```

El comando git revert es una operación para deshacer de forma progresiva que ofrece una forma segura de deshacer los cambios. En vez de eliminar confirmaciones o dejarlas huérfanas en el historial de confirmaciones, la reversión creará una nueva confirmación que invierte los cambios especificados.

Deshacer un commit

```
git reset HEAD^
git reset HEAD~3
git reset <commit>
git reset [--mixed][--soft][--hard]
```

--mixed: deja los cambios en el working directory (tienes q hacer el add). Es el que hace por defecto.

--soft: deja en el stage area los cambios listos para ser comiteados

--hard: elimina todo (de todo)

Head vs Detached Head

Mover el puntero activo HEAD a un estado específico

▼ ¿Qué es el HEAD?

Referencia que apunta a la rama donde te encuentras en cada momento, la rama activa. No es el último commit. Será tu último commit depende de cuál sea tu rama activa. El HEAD apunta a la rama y la rama apunta al último commit. Es una doble referencia.

▼ ¿Qué es un DETACHED HEAD?

Hay HEAD pero no apunta a la cabecera de una rama. Cualquier confirmación que hagas aquí se perderá si te vas de ahí y no hay rama que le apunte.

Reescribiendo la historia

Modificar la última confirmación de cambios

```
git commit --amend
```

Reorganizando el trabajo realizado

```
git rebase [-ip]
```

Deshacer un commit

```
git reset
```

▼ *¿Qué pasa si pusheo, hago un reset --hard a otro commit y vuelvo a pushear?*

Si he comiteado y pusheado un commit que quiero deshacer, mejor utilizar git revert. Como “criterio”, mejor no “no reescribir la historia” de todo aquello que ya ha sido pusheado.

Otros conceptos útiles

Tags

Similar a una rama son punteros a commits. Sirven para taggear puntos específicos como importantes. Normalmente para marcar releases (p.e. los tags se convierten en releases en github). La rama se mueve al añadir nuevos commits, el tag sin embargo se queda estático.

Dos tipos de tags: lightweight y annotated. Por defecto lightweight, annotated con `-a.` El annotated requiere un mensaje como un commit (`-m`) y tiene un objeto con un hash. El lightweight es simplemente una rama que no se mueve.

Se puede hacer checkout a un tag pero el puntero se queda en detached HEAD porque no apunta a ninguna rama.

▼ **Crear un tag**

```
git tag v1.0
```

▼ **Listar tags**

```
git tag
```

▼ **Eliminar tag local**

```
git tag -d v1.0
```

▼ **Publicar tags**

```
git push origin --tags
```

▼ **Eliminar tags remotos**

```
git push origin :v1.0
```

Cherry pick

Trasladar los cambios de uno o varios commits en otra rama en orden. Genera nuevos commits con los nuevos cambios.

```
git cherry-pick A..B #A^..B para incluir A [--no-commit]
```

Hooks

Permite lanzar scripts cuando suceden ciertas acciones importantes. Están todos en `.git/hooks`. Hay un ejemplo para cada uno.

Hooks disponibles:

- **Cliente:** *pre-commit* (`–no-verify`), prepare-commit-msg, commit-msg, post-commit, pre-rebase, post-checkout, post-merge
- **Servidor:** pre-receive, post-receive, update

> No se clonan, son locales

Ejemplo: husky, pre-commit

Alias

Nos sirven para crear atajos de comandos, podemos guardar diferentes alias de forma global y quedarán guardados en la configuración de git.

```
git config --global alias.co checkout
```

```
git config --global -e #muestra el archivo config con los alias creados
```

Configuración

Git permite configurar opciones a nivel de repositorio o globales para todos los repositorios locales

▼ **Especificar el nombre de usuario**

```
git config --global user.name "FIRST_NAME LAST_NAME"

git config user.name "FIRST_NAME LAST_NAME"
```

▼ **Especificar email**

```
git config --global user.email "MY_NAME@example.com"
```

▼ **Otras opciones**

```
git config {--global} opción "VALOR"
```

Stash

Guardado rápido provisional. Usarlo con orden, se puede convertir en un cajón de sastre.

▼ **Guardar un stash**

```
git stash [--include-untracked]
```

▼ **Listar stash**

```
git stash list
```

▼ **Aplicar el último stash**

```
git stash apply
```

▼ **Aplicar cambios de un stash y lo elimina inmediatamente de la pila**

```
git stash pop
git stash pop stash@{0}

git stash apply --index
```

▼ **Eliminar un stash**

```
git stash drop stash@{0}
```

▼ **Eliminar todos los stash guardados**

```
git stash clear
```

Otros

- Comandos interactivo (rebase, clean, etc.)
- Reflog: histórico de commits
- Bisect

Flujos de trabajo:

- Git Flow: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
- Trunk Based Development: https://www.atlassian.com/continuous-delivery/continuous-integration/trunk-based-development
    - Gestión de ramas simplificada.
    - CI/CD

Mensajes de commits

https://chris.beams.io/posts/git-commit/

Bibliografía

- https://git-scm.com/
- https://git-school.github.io/visualizing-git/
- https://rachelcarmena.github.io/2018/12/12/how-to-teach-git.html
- https://ohmygit.org/