GIT (20h)

- Cliente: Indra
- Fechas: 1 – 4 octubre
- Horario: 9-14h
- Duración: 20 horas
- Modalidad: Virtual - Cisco WebEx

Índice

Formador

Alejandro Cerezo Lasne alce65@hotmail.es

https://www.linkedin.com/in/alejandrocerezo/

https://github.com/alce65

Formador / Desarrollador Web FullStack

- JavaScript - Typescript - Angular - React
- NodeJS - Express - MongoDB - MySQL

Contenidos

- Objetos: blobs, trees, commits, tags

4. HERRAMIENTAS PARA PREPARAR UN BUEN COMMIT EN CUALQUIER SITUACIÓN

- git add p
- git rm, git mv
- git diff
- git blame | git log string
- .gitignore

5. REESCRIBIENDO LA HISTORIA

- amend
- checkout
- reset
- stash
- git clean n | git clean f
- revert
- rebase
- git bisect

6. TRABAJANDO EN PARALELO

- branches
  - Crear, borrar, intercambiar
  - Crear desde ref (git checkout b mybranch master~1)
- tags
  - Crear, usar
- patches
  - Crear, aplicar
- remotes:
  - remote v
  - push/pull
  - clones
  - repos bare
  - push branch, push tag
- Resolución de conflictos
- merge VS rebase VS cherrypick
- Pull Request
  - Creación
  - Uso
  - Merging
  - Cierre

7. UTILIDADES

- GitK, GitG y git gui | git log graph | formato git log
- IntelliJ

8. CONFIGURACIÓN DE GIT. Hooks

    - .alias
    - gitconfig
        - Editor
        - Coloreado comandos
        - Formato salida comandos
        - Otras opciones
    - Hooks
        - Cómo crear
        - hooks de lado cliente: commits, emails, rebase, ...
        - hooks de lado servidor: prereceive, postreceive, update

9. SUB-PROYECTOS

    - Crear submodules
    - workflow de commits
    - git submodule status recursive
    - git submodule foreach ...

10. INTEGRACIÓN CON OTRAS HERRAMIENTAS Y ENTORNOS

    - SourceTree
    - Github
    - GitLab
    - Bitbucket

11. BUENAS PRÁCTICAS

    - Commits atómicos
    - Commits frecuentes
    - No commits de trabajo a medias
    - Test antes de commit
    - Buenos mensajes de commit
    - Usar branches, feature-branching
    - Workflows
        - Presentar las opciones más usadas
        - Fijar un workflow común

Día 1

INTRODUCCIÓN

Qué es un SCV

Los sistemas de control de versiones (SCV) son una herramienta esencial para manejar proyectos de software; lo que ha hecho de ellos herramientas de uso habitual en el desarrollo profesional de software desde hace décadas.

Proporcionan una serie de funcionalidades claves para el desarrollo de proyectos como es

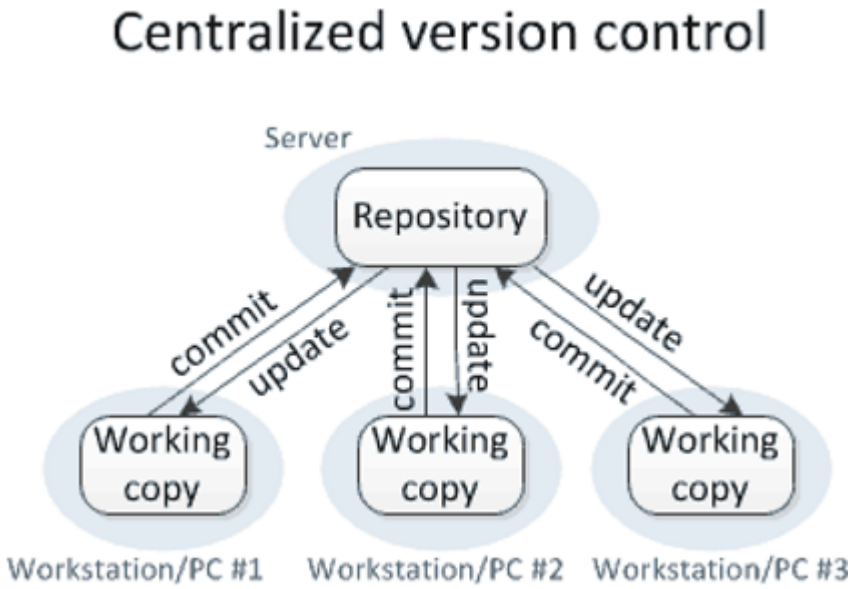
- el control de cambios en el código,

- la reversibilidad de dichos cambios,
- la posibilidad de colaborar en el desarrollo del código.

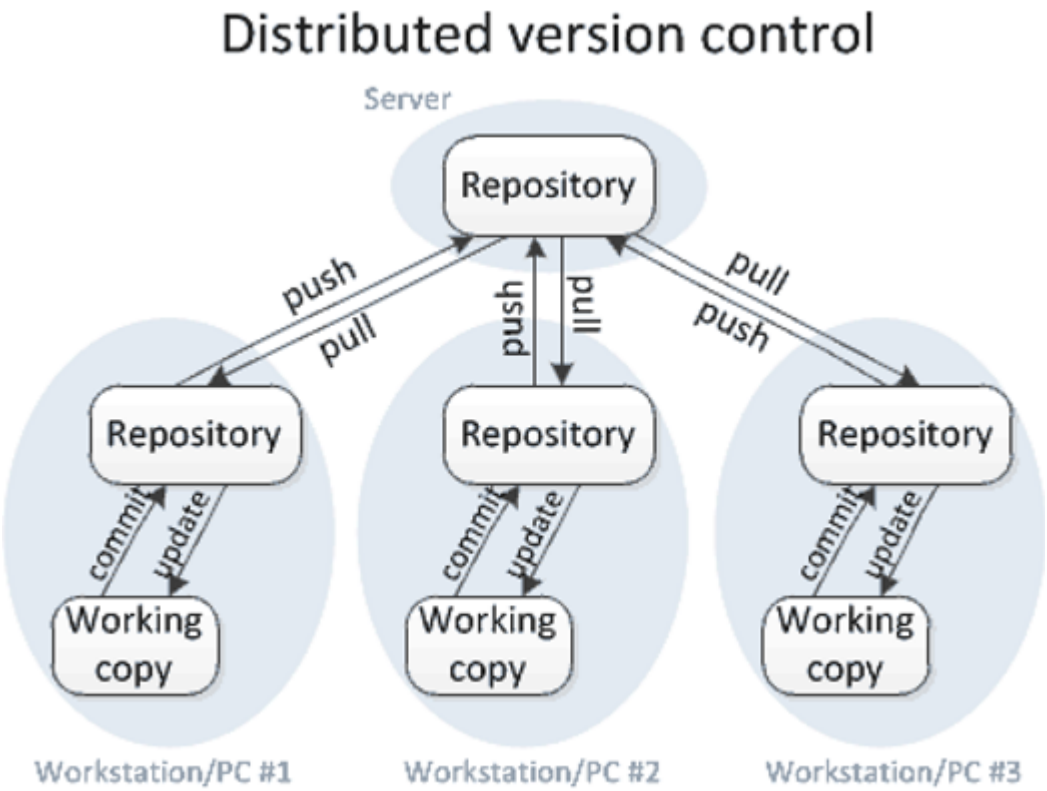
Además, los SCV permiten tener en paralelo varias versiones o ramas del proyecto. Las ramas se utilizan para desarrollar funcionalidades aisladas de los cambios en otras partes del proyecto que posteriormente pueden integrarse en la rama principal.

Tipos de SCV: centralizados y distribuidos

Los **SCV centralizados** como CVS y Subversion (SVN) son sistemas cliente-servidor donde hay un repositorio canónico en el servidor que contiene toda la información de los cambios mientras que los clientes solo tienen copias de trabajo.



En **sistemas distribuidos** como Mercurial y Git no existe el concepto de repositorio canónico por lo que cada cliente ha de tener una copia completa del repositorio. Desde 2010, la tendencia es utilizar cada vez más SCV distribuidos, en particular Git



Git: un SCV distribuido: Historia de GIT

Git es un sistema de control de versiones distribuido que fue creado por *Linus Torvalds* en 2005 para el desarrollo del kernel de Linux. Es un sistema de control de versiones escrito mayoritariamente en **C**, de **código abierto** y **gratuito** que es muy rápido y eficiente.

Git es un software diseñado pensando en la **eficiencia** y la **confiabilidad** del mantenimiento de versiones y aplicaciones cuando éstas tienen un **gran número de archivos** de código fuente.

En comparación con otros sistemas de control de versiones, Git es más rápido y tiene un tamaño de repositorio más pequeño. También tiene una **escalabilidad** muy buena, lo que significa que puede manejar proyectos muy grandes y muy pequeños.

En el contexto en el que fue diseñado, las dos principales opciones eran SVN y Mercurial. Git se diseñó para ser más rápido y eficiente que Mercurial, aunque conservando su estrategia de SVC distribuido, y más fácil de usar que SVN.

Características de Git como SCV distribuido

La experiencia del equipo de *Linus Torvalds* en la gestión de la integración de las diferentes aportaciones en un proyecto distribuido de la magnitud del kernel de Linux determinó los objetivos del proyecto

- Rapidez
- Simplicidad de uso
- Multiplicidad de versiones (Ramas)
- Distribuido: capaz de trabajar sin conexiones
- Preparado para grandes proyectos

Estos objetivos se reflejaron en las siguientes decisiones de implementación:

Versiones no incrementales. Git almacena cada cambio como una instantánea de todos los archivos del proyecto. Para ser eficiente, si el archivo no ha sido modificado, sólo se almacena un enlace al archivo idéntico previamente almacenado. Los SCV anteriores a Git habitualmente almacenaban solo una versión base y las modificaciones hechas en cada cambio por archivo.

Trabajo fuera de línea. Por ser un sistema distribuido, cada repositorio de Git es un repositorio completo capaz de funcionar sin acceso a la red o al resto de los repositorios distribuidos gracias a que contiene una copia local de la historia completa del desarrollo del proyecto. Los cambios en la historia pueden copiarse de un repositorio a otro como nuevas ramas de desarrollo y se pueden copiar de la misma manera que una rama de desarrollo local.

- Cada operación se realiza en el **repositorio local**.
- No necesita conexión con un servidor.
- Se puede trabajar normalmente sin conexión (**offline**)
- Es más **rápido** que los repositorios centralizados

Autenticación criptográfica de la historia. El identificador de un cambio se computa utilizando un algoritmo criptográfico que utiliza como entrada el cambio y la historia completa de cambios. Esto permite que cualquier cambio de la información durante la transmisión o en sistema de archivos sea detectado por Git.

- Todo está identificado por secuencias **hash**, como 24b9da6552252987aa493b52f8696cd6d3b00373
- ELos hashes son **inmutables**: imposible cambiar el contenido de cualquier archivo o directorio sin que Git lo sepa.
- No se puede perder información ni dañar el archivo sin que Git pueda detectarlo.

Instalación y configuración inicial

Disponible en la web para Windows, Mac y Linux. En este momento (octubre 2024) la versión disponible es la 2.46.2.

En la instalación guiada pueden indicarse algunos elementos de configuración como:

- los complementos incluidos (iconos, integración con el entorno)
- el editor que utilizará Git (e.g. Visual Studio Code o VSC).
- el ajuste del path para permitir su uso en diversos terminales
- el transporte HTTPS, generalmente openSSH
- la codificación de los saltos de línea (CRLF o LF)
- las opciones de terminal (minTTY, cmd, powershell)
- la gestión de las credenciales (credential helper)
- la cache de ficheros y la posibilidad de incluir enlaces simbólicos (desactivada por defecto)

Terminales

Git es una herramienta de línea de comandos desarrollada inicialmente para el Bash de Linux. En Windows, Git se instala con un terminal propio que es una versión de MinTTY, un emulador de terminal para Windows que permite el uso de Bash, conocida como **GitBash**.

Sin embargo, Git puede utilizarse en cualquier terminal de Windows, como el **cmd** o el **PowerShell**. Para ello, es necesario ajustar el path de Windows para que el ejecutable de Git esté disponible en todas las terminales, lo que se puede hacer en la instalación de Git.

PowerShell

El powershell de Windows es un terminal más potente que el cmd y que el propio GitBash, pero menos utilizado en el desarrollo de software. Puede configurarse para que funcione específicamente con Git añadiendo el plugin posh-git.

La web de Git proporciona una guía para la configuración de Git en Windows que incluye la configuración de las powershell.

El primer paso es habilitar la ejecución de scripts en PowerShell con el comando, adecuado, según se trate de hacerlo para el usuario actual o para todos los usuarios del sistema.

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned -Force
Set-ExecutionPolicy -Scope LocalMachine -ExecutionPolicy RemoteSigned -Force
```

El siguiente paso es instalar el módulo de posh-git con el comando

```
Install-Module posh-git -Scope CurrentUser -Force
```

Finalmente hay que añadir la importación del módulo en el fichero de configuración de PowerShell, que se encuentra en el directorio de usuario en el fichero `Microsoft.PowerShell_profile.ps1`. Para ello, se pueden utilizar los comandos

```
Import-Module <path-to-uncompress-folder>\src\posh-git.psd1
Add-PoshGitToProfile -AllHosts
```

El comando less

git muestra la información usando el comande less, que permite desplazarse por la información mostrada. Los comandos básicos son

:f -> scroll next page :b -> scroll previous page :q -> quit

Configuración

Git se configura a tres niveles

- **Sistema**. Archivo /etc/gitconfig: Contiene valores para todos los usuarios del sistema y todos sus repositorios. Si pasas la opción `--system` a git config, lee y escribe específicamente en este archivo.
- **Global o usuario**. Archivo ~/.gitconfig: Específico a tu usuario. Puedes hacer que Git lea y escriba específicamente en este archivo pasando la opción `--global`.
- **Repositorio**. Archivo config "local", en el directorio de Git (es decir, .git/config) del repositorio que estés utilizando actualmente: Específico a ese repositorio.

Cada nivel sobrescribe los valores del nivel anterior, por lo que los valores de .git/config tienen preferencia sobre los de /etc/gitconfig

Para conocer la configuración actual de Git, se puede utilizar el comando

```
git config --list
git config --list --show-origin
```

que mostrará la configuración actual de Git y en el segundo caso, el archivo donde se ha definido.

Para ver solo la configuración global, se puede utilizar

```
git config --global -l
```

Para editar la configuración global, se puede utilizar el comando `git config --global` añadiendo la clave y el valor que se quiere añadir.

Los valores imprescindibles son

```
git config --global user.name "Alejandro Cerezo"
git config --global user.email "alce65@hotmeil.es"
```

Si no se define el name y el email, Git no permitirá hacer commits.

Otros valores que se pueden definir son

```
git config --global core.editor "code --wait --new-window"
git config --global init.defaultBranch main
git config --global core.autocrlf false
git config --global core.ignorecase true
```

Los dos primeros, probablemente estarán definidos a nivel system durante la instalación de Git. El segundo es una opción que se ha añadido en la versión 2.28 de Git para cambiar el nombre de la rama principal de los repositorios de Git de master a main.

En caso de tener un repo cuya rama principal se llame master, se puede cambiar el nombre de la rama principal con el comando

```
git branch -m master main
```

El tercero y el cuarto son recomendables para evitar problemas con los saltos de línea en Windows y con la sensibilidad a mayúsculas y minúsculas.

Referencias

- Git Reference Documentación oficial de Git.
- Pro Git Free book on Git. *Scott Chacon* and *Ben Straub*. 2014.
- Gitting Things Done – A Visual and Practical Guide to Git [Full Book] *Omer Rosenbaum*. 2024
- git - the simple guide Cheat sheet de git. *Roger Dudler*. 2013.
- Think Like (a) Git

QUICK START

Comandos básicos de Git

- `git init`: Inicializa un repositorio local de Git
- `git status`: Muestra el estado de los archivos en el directorio de trabajo (workArea) y el área de preparación
- `git add`: Agrega un archivo al área de preparación (stagedArea)
- `git commit`: Guarda los cambios en el repositorio
- `git log`: Muestra el historial de commits
- `git show`: Muestra los cambios de un commit
- `git clone`: Clona un repositorio de Git

Primeros pasos

Primer repo (init)

Para crear un repositorio local de Git, se utiliza el comando `git init`. Este comando crea un directorio oculto `.git` en el directorio actual que contiene todos los metadatos necesarios para el control de versiones.

```
git init
```

Si no se indica nada, Git creará un repositorio en el directorio actual. Si se quiere crear un repositorio en un directorio específico, se puede indicar el directorio como argumento del comando.

```
git init nombre_del_repo
```

Por convención se suelen añadir dos ficheros en el directorio raíz del repositorio:

- `.gitignore`: Fichero que contiene los patrones de nombres de ficheros y directorios que Git ignorará.
- `README.md`: Fichero que contiene información sobre el repositorio.

El primero de ellos debe excluir las dependencias a terceros, los ficheros generados y los ficheros de configuración. Su contenido dependerá del lenguaje de programación y las herramientas que se utilicen en el proyecto.

Por ejemplo, para un proyecto en JS, el fichero `.gitignore` podría contener

```
node_modules
dist
```

El `README.md` debe contener información sobre el repositorio, como su propósito, su estructura y su uso. Si nuestro repo va a ser publico en Github, se considera una falta de cortesía no añadir un README.md. Además, es recomendable añadir una licencia y un fichero `LICENSE.md` con la licencia que se va a utilizar.

Anatomía de un repositorio git: staging area, index and cache

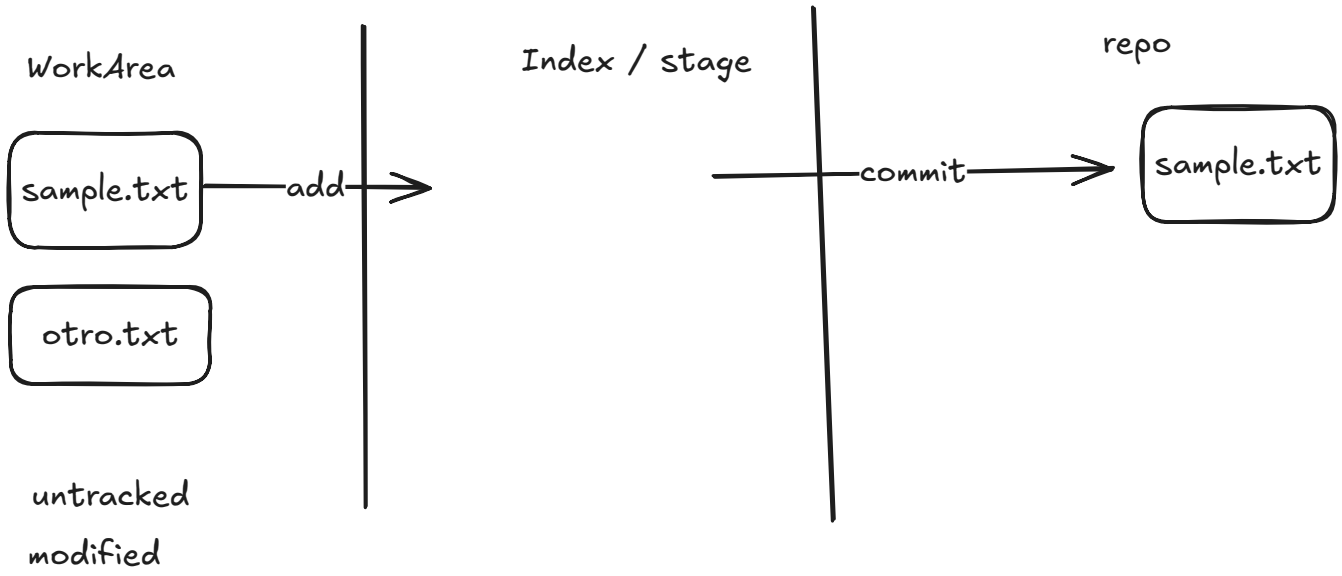
Para entender el funcionamiento de Git, nuestro modelo mental representara un repositorio de Git como tres áreas:

- `Working Area`: Directorio de trabajo
- `Staging Area`: Área de preparación
- `Repository`: Repositorio -> conjunto de commits y etiquetas

La working area es el directorio de trabajo donde se encuentran los ficheros del proyecto.

La staging area es un área intermedia donde se preparan los cambios antes de hacer un commit. Esto permite seleccionar los cambios que se quieren incluir en el commit ignorando por el momento los que no se quieren incluir.

El repositorio es el lugar donde se almacenan los commits y las etiquetas.



Los ficheros cambian su estado en función de las operaciones que se realicen con ellos y de su relación con estas áreas de Git. Los estados posibles son

- `Untracked` (sin seguimiento): Fichero no seguido por Git
- `Tracked`(con seguimiento): Fichero seguido por Git
  - `Modified`(modificado): Fichero modificado
  - `Staged` (preparado): Fichero añadido al área de preparación

    - `Committed` (confirmado): Fichero añadido al repositorio

Los comandos básicos de Git son los que permiten mover los ficheros entre estas áreas:

En el sentido habitual de trabajo, los comandos son

- `git add <pattern>`: Mueve los ficheros de la working area a la staging area
- `git commit`: Mueve los ficheros de la staging area al repositorio

En el sentido contrario, los comandos son, como luego veremos

- `git checkout <pattern>`: Mueve los ficheros de la staging area a la working area
- `git reset <pattern>`: Mueve los ficheros de la staging area al working area

Añadir contenidos al repositorio. Primer commit

Teniendo en cuenta el modelo mental de un repositorio de Git, nuestro trabajo inicial yy habitualmente seguirá los siguientes pasos:

1. Crear un repositorio local de Git con `git init`
2. Acceder al directorio de trabajo con `cd`
3. Crear los ficheros del proyecto
4. Comprobar el estado del repositorio con `git status`
5. Añadir los ficheros al repositorio con `git add`
6. De nuevo, comprobar el estado del repositorio con `git status`
7. Hacer el primer commit con `git commit`
8. Comprobar el historial de commits con `git log`

```
git init sample
cd sample
git status
echo 'Sample One' > sample1.txt
echo 'Sample Two' > sample2.txt
git add .
git status
git commit -m "Initial commit"
git log
git log --graph --decorate --oneline
```

Git add

Git add es el comando que se utiliza para añadir ficheros al área de preparación. Se puede utilizar de varias formas

`git add <fichero>` añade un solo fichero. `git add <patrón>` añade todos los ficheros que coincidan con el patrón, e.g. `git add *.txt` `git add .` añade todos los ficheros del directorio de trabajo al área de preparación.

Git add no añade los ficheros al repositorio, solo los añade al área de preparación. Es el paso previo a hacer un commit.

Git status

Git status nos muestra el estado de los ficheros en el directorio de trabajo y en el área de preparación. Nos indica qué ficheros han sido

- modificados (M)
- añadidos (U de untracked)
- eliminados (D)

Igualmente indica si están en el área de preparación o no.

Se puede utilizar `git status -s` para obtener una salida más compacta.

Además de `git status`, muestra información de alguna de las operaciones posibles en función de los ficheros que detecta en el directorio de trabajo y en el área de preparación.

Git commit. Mensajes de commit

Git commit es el comando que se utiliza para guardar en el repositorio los cambios que previamente se han añadido al área de preparación con `git add`.

```
git commit -m "Mensaje del commit"
```

Un commit es una **instantánea** (**snapshot**) de los cambios en el repositorio. En el se guardan los ficheros nuevos o modificados que se han añadido al área de preparación. Cada commit tiene una serie de elementos

- un SHA (Secure Hash Algorithm) que es un identificador único
- un autor
- un mensaje que describe los cambios realizados
- una fecha y hora de creación
- una referencia al commit padre (excepto el primer commit)
- un árbol cambios en los archivos y directorios

Más adelante veremos en que consisten exactamente los commits desde un punto de vista técnico, y como se combinan con otros elementos de git, **blobs** y **trees**, para construir el repositorio.

Los **mensajes de commit** son una parte muy importante de Git. Un buen mensaje de commit debe

- solo la primera palabra en mayúsculas
- comienza con un verbo en imperativo
- no se termina el mensaje con un punto
- no ser más largo de 50 (70) caracteres
- no ser un resumen de los cambios
- ser conciso y descriptivo
- contar su qué y sobre todo por qué, cómo

- reflejar la granularidad del commit

git log / git show

Git log es el comando que se utiliza para ver el historial de commits de un repositorio. Muestra los commits en orden cronológico inverso, es decir, el último commit en la parte superior.

```
git log
```

Es un comando con gran número de opciones que permiten personalizar la salida del comando. Por ejemplo, `git log --oneline` muestra los commits en una sola línea. Más adelante volveremos a hablar de este comando.

Git show es el comando que se utiliza para ver los cambios de un commit. Muestra los cambios realizados en un commit en relación con su padre.

```
git show
```

APRENDIENDO A REFERENCIAR REVISIONES Y PATHS

Anatomía de comandos típicos, referencias vs paths

Los comandos de Git siguen una estructura común que se compone de

- Comando y opciones
- Flags
- Paths
- Referencias

Por ejemplo

```
git log --graph --decorate --oneline
```

- `git`: Comando
- `log`: Opción
- `--graph`, `--decorate`, `--oneline`: Flags

```
git add *.js
```

- `git`: Comando
- `add`: Opción
- `*.js`: Paths

```
git show HEAD~1
```

- `git`: Comando
- `show`: Opción
- `HEAD~1`: Referencia

En algunos contexto puede ser válida tanto una referencia como un path. Por ejemplo, en el comando `git checkout`, el path puede ser un fichero o directorio o una referencia a un commit. El **doble guion** permite separar un nombre de archivo explícitamente identificado

- Checkout the tag named “main.c”

```
git checkout main.c
```

- Checkout the file named “main.c”

```
git checkout -- main.c
```

Paths

Los paths son los ficheros y directorios que se quieren incluir en una operación de Git. Su estructura sigue los criterios del sistema operativo y utiliza los siguientes caracteres especiales

- `.`: Directorio actual
- `..`: Directorio padre
- `/`: Separador de directorios
- `~`: Directorio home del usuario
- `*`: Comodín para cualquier cadena de caracteres
- `?`: Comodín para un solo carácter
- `[]`: Comodín para un rango de caracteres

Tipos de referencias

Las referencias son los commits y las ramas a los que se quiere hacer referencia.

- Absolutas: SHA1 que identifica un commit o su sub-cadena (5 primeros caracteres)
- Simbólicas: mediante etiquetas: HEAD, master (otras ramas) y otras referencias útiles (tags)
- Relativas: desde cualquiera de las anteriores, se puede acceder a commits anteriores o posteriores
- Por mensaje de commit (:/cadena)

Referencias simbólicas

Las referencias simbólicas son las etiquetas de Git:

- HEAD: Puntero a la rama actual

- Rama: puntero al commit actual (normalmente el ultimo)
- Etiquetas (Tags): Pueden ser anotadas o ligeras

Referencias relativas

Las referencias relativas permiten acceder a commits anteriores o posteriores a un commit dado.

Utilizan los siguientes operadores

- ~ (tilde): seguido de un número, indica el commit al que acceder
- ^ (circunflejo): el padre

Por ejemplo

- `HEAD~1`: El padre del commit actual
- `HEAD~n`: El n-ésimo padre del commit actual
- `HEAD^`: El padre del commit actual
- `HEAD^^`: El abuelo del commit actual

INTEGRACIÓN CON OTRAS HERRAMIENTAS Y ENTORNOS

Clientes gráficos

Cliente incluido en la instalación de Git ¿alguien lo usa?

Otros clientes gráficos según la web de Git

- GitHub Desktop Platforms: Mac, Windows Price: Free License: MIT
- SourceTree Platforms: Mac, Windows Price: Free License: Proprietary
- TortoiseGit Platforms: Windows Price: Free License: GNU GPL

Entornos de desarrollo

VSCode

VSC (Visual Studio Code) incluye soporte nativo para Git, permitiendo realizar las operaciones más comunes desde el propio editor.

Además existen numerosos plugins que amplían las funcionalidades de Git en VSC

- Git Graph
- GitLens
- Git History

Repositorios remotos (hosting de repositorios)

Con el fin de hacer disponible los repositorios de Git en la red, existen en Git 2 tipos de repositorios

- **Bare** (desnudo, simple): no tiene directorio de trabajo. No se usa para el directamente en el desarrollo de aplicaciones ya que en él no se pueden realizar directamente commits. De este tipo son los

repositorios que se publican, en servidores estándar o especializados en el alojamiento (hosting) de repositorios
- **Desarrollo**: repositorio típico. Mantiene la rama en la que se trabaja, proporciona una copia extraída de dicha rama en el **directorio de trabajo**

El primero de estos formatos se utiliza para publicar los repositorios en servidores de Git públicos, como **Github**, **GitLab** o **Bitbucket** o pertenecientes a una determinada empresa.

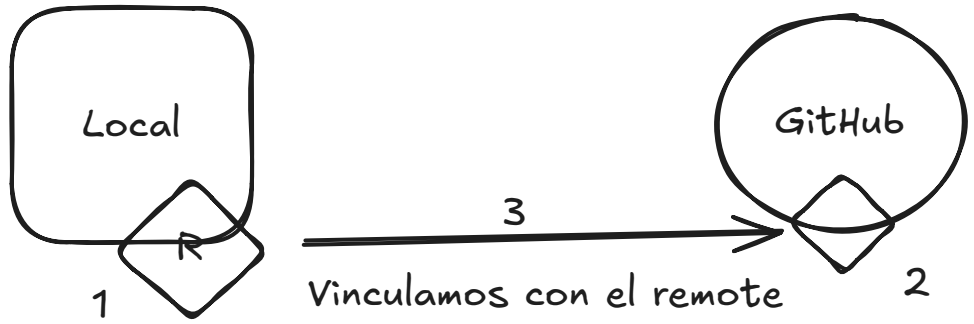
El establecimiento de la relación entre el repositorio local y el remoto se puede realizar en cualquiera de las dos direcciones:

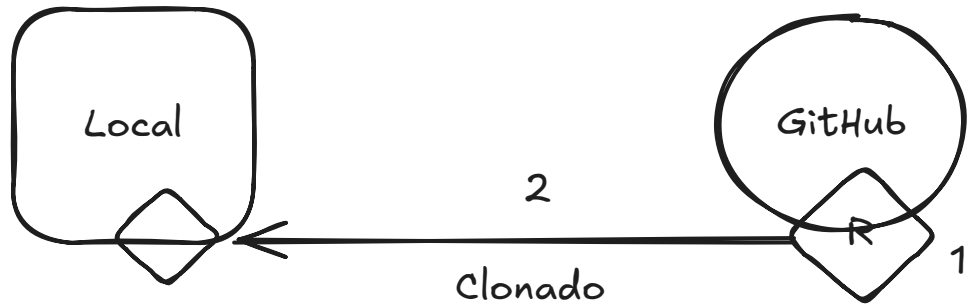
- Desde el repositorio local al remoto, mediante el comando `git remote add`

```
git remote add origin <url>
git push -u origin main
```

- Desde el repositorio remoto al local, mediante el comando `git clone`

```
git clone <url>
```





Cualquiera que sea la forma en que se han creado, una vez que existe un remote, los dos operaciones principales en relación con él son

- `git push`: Sube todos los cambios locales al repositorio remoto
- `git pull`: Descarga los cambios del repositorio remoto al repositorio local

Comandos de conexión y uso del repositorio remoto (resumen)

- `git clone <url>`
- `git remote add origin https://github.com/alce65/nombre_del_repo.git`
- `git push -u origin main`
- `git push`
- `git pull`

Dia 2

Git internals: Plumbing commands

Para conocer/manipular la estructura interna de Git, se utilizan los **plumbing commands** (comandos de fontanería). Estos comandos son los que utiliza el propio Git internamente para construir los comandos de más alto nivel conocidos como **porcelain commands**.

La carpeta .git

En términos de sistema de ficheros, un repositorio Git es una carpeta que incluye un directorio oculto `.git`. Éste contiene toda la información necesaria para el control de versiones:

- **Objetos**: blobs, trees, commits, tags anotados
- **Referencias**: HEAD, master branch, tags ligeros
- Hooks: scripts que se ejecutan en determinados eventos
- Info y Logs: información extra sobre el repo y los commits
- fichero Index: área de preparación
- fichero de configuración local

```
tree .git
tree .git/objects /f
```

Se puede ver que el resultado muestra como la carpeta objects consta de una serie de cartetas/archivos que contienen los objetos de Git, por el momento.

- blobs (los archivo)
- trees (los directorios)
- commits (los snapshots)

Más adelante veremos un cuarto tipo de objeto, los tags (anotadas).

Hashes: creación y lectura

La siguiente descripción refleja la forma de explicar estos concepto propuesta por *Paolo Perrotta* en su conferencia *Understanding Git*, 3 diciembre 2016, disponible en YouTube.

En su capa más profunda Git es un **mapa de objetos** de acuerdo con un patrón clave/valor. Cada objeto es un cierto contenido (valor) que tiene un hash que lo identifica (clave).

Existe un comando de git que permite obtener el hash de un objeto, `git hash-object`. Este comando toma un archivo o cualquier contenido y devuelve el hash del objeto creado.

```
echo "Hello, World" | git hash-object --stdin
// 110fdc0ce1c7582e08f31e17bbcfdec1b50a478c
```

En este comando, el hash para un mismo contenido y en el mismo shell siempre será el mismo. El shell influye porque en cada uno de ellos el stdin es distinto.

El mismo contenido de un archivo siempre tendrá el mismo hash de 40 bytes, sin importar el nombre del archivo o la ubicación en el sistema de archivos. Este hash es único para el contenido con una probabilidad tan alta que se puede considerar único.

> Un hash SHA-1 es un número hexadecimal de 40 caracteres que se utiliza para identificar de forma única los objetos de Git. Se calcula a partir del contenido del objeto y de su tipo.

Si añadimos al comando la opción `-w`, para guardar el objeto obtendremos un mensaje de error porque no nos encontramos en un repositorio de Git.

```
echo "Hello, World" | git hash-object -w --stdin
// fatal: not a git repository (or any of the parent directories): .git
```

El mapa de hashes creado por Git se convierte en **persistente** gracias a la existencia de un **repositorio**, que es básicamente un directorio `.git/objects` que contiene todos los objetos de git. Cada objeto es guardado en un archivo con el nombre del hash del objeto.

Creamos un repositorio vacío y guardamos el objeto en el repositorio.

```
git init sample-repo
cd sample-repo
dir /a:hd // [= ls -a]
echo "Hello, World" | git hash-object -w --stdin
//110fdc0ce1c7582e08f31e17bbcfdec1b50a478c
cd git
dir
cd objects
dir
cd 11
dir // 0fdc0ce1c7582e08f31e17bbcfdec1b50a478c
```

> El repositorio es una **base de datos de objetos** que se almacena en la carpeta oculta `.git`, que contiene los objetos de git en la carpeta objects. Por **cada hash** se crea una carpeta con los primeros dos caracteres del hash y dentro de esta carpeta se crea un fichero cuyo nombre es el resto del hash, que contiene la información correspondiente a ese hash en un formato comprimido.

De cada objeto guardado en un archivo se puede recuperar la información que contiene a partir del hash del objeto.

```
git cat-file -t 110fdc0 // blob
git cat-file -p 110fdc0 // Hello, World
```

Cada objeto es accesible por los 5 primeros caracteres de su hash: la carpeta (2 caracteres) + inicio del fichero (3 caracteres)

Elementos de un repositorio- Primer commit

Creamos un proyecto muy simple.

```
cd cook.book
```

C:. │ menu.txt // Apple Pie └───recipes apple_pie.txt readme.txt

```
cd cook.book
type menu.txt
// Apple Pie
type .\recipes\readme.txt
// Put your recipes in this folder, one for file.
type .\recipes\apple_pie.txt
//Apple Pie
```

Creamos un **repositorio** en el proyecto con un **commit inicial**.

```
cd cook.book
git init
git status
git add .
git commit -m "Initial commit"
```

Vemos el resultado en la carpeta objects del repositorio.

```
cd .git
cd dir /w // [25]   [5d]   [77]   [8d]   [9e]
cd 1f
```

Comprobamos los **logs** del repositorio

```
git log

commit 5da6f7b4682638317b18a5fe0f9edca98aeb1f7c (HEAD -> main)
Author: Alejandro Cerezo <alce65@hotmail.es>
Date:   Sat Sep 28 13:14:18 2024 +0200

    Initial commit
```

El **commit** corresponde al objeto 5d-a6f7b4682638317b18a5fe0f9edca98aeb1f7c que podemos leer con `git cat-file`.

```
git cat-file -p 5da6f7b

tree 8de329e56d2bf59ad7ce6df33df79e91a2a4a5a8
author Alejandro Cerezo <alce65@hotmail.es> 1727522058 +0200
committer Alejandro Cerezo <alce65@hotmail.es> 1727522058 +0200

Initial commit
```

Un resultado similar se consigue con el comando `git show --pretty=raw`, aunque en este caso se añade la información del diff.

```
git show 5da6f7b

commit 5da6f7b4682638317b18a5fe0f9edca98aeb1f7c
tree 8de329e56d2bf59ad7ce6df33df79e91a2a4a5a8
author Alejandro Cerezo <alce65@hotmail.es> 1727522058 +0200
committer Alejandro Cerezo <alce65@hotmail.es> 1727522058 +0200

    Initial commit

diff --git a/menu.txt b/menu.txt
new file mode 100644
```

Un commit no es mas que un texto, que como cualquier otro objeto de git, tiene un hash que lo identifica. En este texto se incluye la metadata del commit, como el autor, el committer, el mensaje al crearlo y la referencia uno o varios hashes.

En este caso, el commit apunta a un objeto de tipo tree que podemos leer con `git cat-file`.

```
git cat-file -p 8de329e

100644 blob 9eed377bbdeb4aa5d14f8df9cd50fed042f41023    menu.txt
040000 tree 25036a158dfdf583f672c11ef79f45c6b0e6347a    recipes
```

El **tree** guarda los nombres de los archivos y directorios (otros tree) que incluye, junto con sus hashes. En este caso, el tree apunta a dos objetos, uno de tipo blob y otro de tipo tree, que nuevamente podemos leer con `git cat-file`.

```
git cat-file -p 9eed377

Apple Pie
```

```
git cat-file -p 25036a1

100644 blob 9eed377bbdeb4aa5d14f8df9cd50fed042f41023    apple_pie.txt
100644 blob 7708256b70bf5e956ea609a785911a31fc14929f    readme.txt
```

En el caso del tree, la misma información se puede obtener con `git ls-tree`.

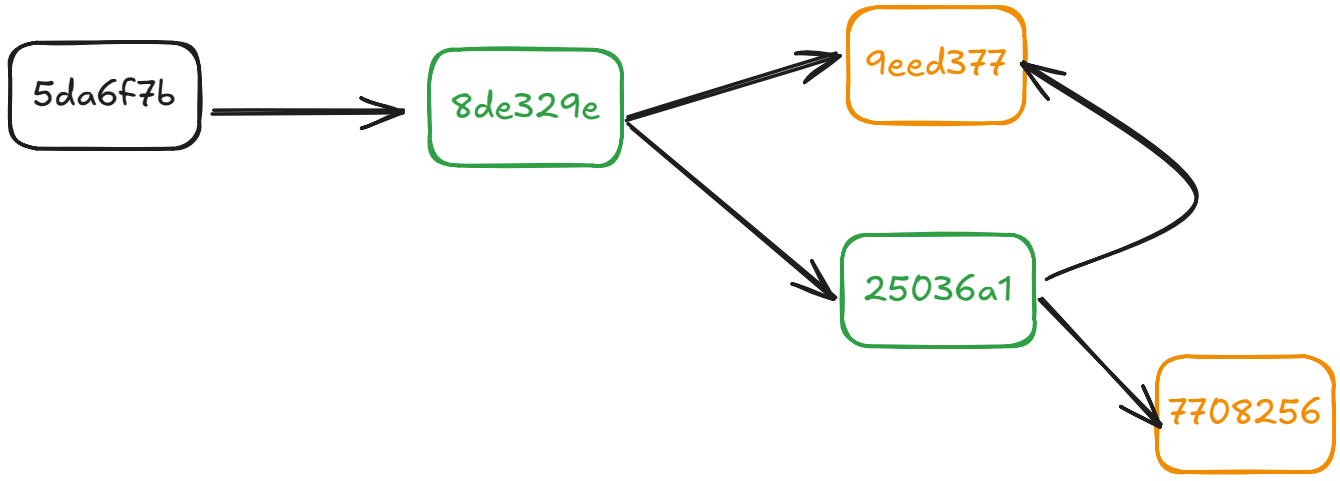
```
git ls-tree 8de329e

100644 blob 9eed377bbdeb4aa5d14f8df9cd50fed042f41023    apple_pie.txt
100644 blob 7708256b70bf5e956ea609a785911a31fc14929f    readme.txt
```

El objeto tree `25036a1` apunta a dos objetos de tipo blob que . Uno de ellos es nuevamente 9eed377, porque el contenido de `apple_pie.txt` es el mismo que el de `menu.txt`. El otro objeto blob lo podemos leer con `git cat-file`.

```
git cat-file -p 7708256

Put your recipes in this folder, one for file.
```



Modificación de un archivo

Añadimos otra línea en el archivo `menu.txt` y comprobamos los cambios.

```
cd ..\..\..\cook.book
echo Cheesecake>> menu.txt
type menu.txt
```

Creamos un nuevo commit para incorporar los cambios.

```
git add .
git commit -m "Add Cheesecake to menu"
git log

commit f1dff43f97543e83cab3f52f054cdcf9b26e8d55 (HEAD -> main)
Author: Alejandro Cerezo <alce65@hotmail.es>
Date:   Sat Sep 28 14:03:08 2024 +0200

    Add Cheesecake to menu
```

De nuevo, el commit es un objeto de tipo que podemos leer con `git cat-file`.

```
git cat-file -p f1dff4

tree 972e64a00a72e413bd158352ab3e6e98461bfbea
parent 5da6f7b4682638317b18a5fe0f9edca98aeb1f7c
author Alejandro Cerezo <alce65@hotmail.es> 1727524988 +0200
committer Alejandro Cerezo <alce65@hotmail.es> 1727524988 +0200

Add Cheesecake to menu
```

El commit apunta a un objeto de tipo commit, el commit anterior y a un nuevo objeto de tipo tree que podemos leer con `git cat-file`.

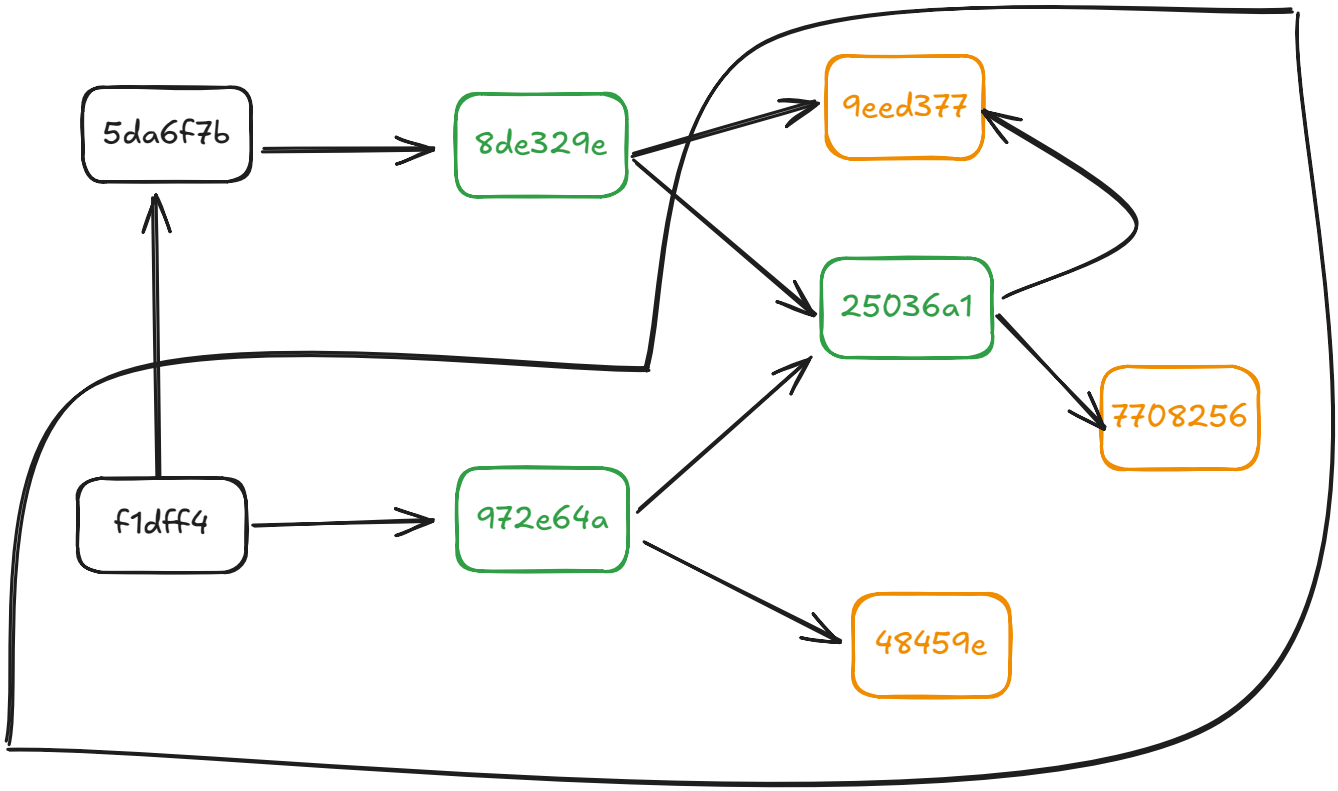
```
git cat-file -p 972e64a

100644 blob 48459e5685c4561d0fa6e26a7371041e982c0ff4    menu.txt
040000 tree 25036a158dfdf583f672c11ef79f45c6b0e6347a    recipes
```

El tree sigue siendo el mismo, pero el objeto blob `48459e5` es diferente al anterior. Si lo leemos con `git cat-file` vemos que contiene la nueva línea.

```
git cat-file -p 48459e5

Apple Pie
Cheesecake
```



Utilizando un commit como punto de entrada es posible recorrer todos los objetos de un repositorio de git correspondientes al momento en que se creó el commit, reconstruyendo a partir de ellos el estado del proyecto en ese momento, lo que denominamos una **snapshot del proyecto**.

Cuando un fichero no cambia, git no guarda una nueva versión del fichero, sino que guarda una referencia al fichero anterior. De esta forma, git ahorra espacio en disco y tiempo de ejecución.

En resumen, un commit es un objeto que apunta a un objeto de tipo tree que apunta a uno o varios objetos de tipo blob. Un objeto de tipo blob es un archivo, un objeto de tipo tree es un directorio y un objeto de tipo commit es un snapshot del proyecto en un momento dado.

> Técnicamente, un repositorio es un **grafo dirigido acíclico** donde los nodos son objetos de git (de tipo commit, tree o blob) y las aristas son referencias entre ellos. Con ello se consigue un **filesystem** de alto nivel, inmutable, eficiente y seguro, por encima del filesystem del sistema operativo. Además, gracias a los commits, se convierte en un **version filesystem**, con **control de versiones**.

References: Branches and HEAD

Las **ramas** en git son simplemente **punteros a commits**. En detalle son archivos de texto plano almacenados en la carpeta `.git/refs/heads` que contienen el hash del commit al que apuntan.

```
dir .git\refs\heads /b

main
```

```
type .git\refs\heads\main

f1dff43f97543e83cab3f52f054cdcf9b26e8d55
```

Crear una nueva rama se reduce a crear un nuevo archivo en la carpeta `.git/refs/heads` con el nombre de la rama y como contenido el hash del commit un commit, que de momento será el mismo que en la rama anterior.

```
git branch feature
dir .git\refs\heads /b

feature
main
```

```
type .git\refs\heads\feature

f1dff43f97543e83cab3f52f054cdcf9b26e8d55
```

Si en el SO creamos un fichero con el nombre de la rama y el contenido del hash del commit, git reconocerá la rama.

```
echo f1dff43f97543e83cab3f52f054cdcf9b26e8d55 > .git\refs\heads\bad-way
git branch

bad-way
feature
* main
```

```
del .git\refs\heads\bad-way

git branch

feature
* main
```

La rama main aparece indicada con un asterisco porque es la **rama actual** (current branch). Eso es lo que indica el **puntero** `HEAD`

El puntero `HEAD` es un archivo de texto plano almacenado en la carpeta `.git` que contiene el nombre de la rama actual. Como se ve en el contenido del archivo, es una referencia a una referencia

```

```

```
type .git\HEAD
ref: refs/heads/main
```

En realidad por cada rama se crea una referencia simbólica en la carpeta `.git/refs/heads` que apunta al hash del commit al que apunta la rama.

```
dir .git\refs\heads /b

feature
main
```

```
type .git\refs\heads\main

f1dff43f97543e83cab3f52f054cdcf9b26e8d55
```

Como veremos más adelante, algunos de los comandos más importantes de Git son aquellos que permiten **mover el puntero** `HEAD` y **cambiar de rama**.

Taller: Crear un nueva repositorio desde cero

En su libro *Gitting Things Done*, 2021, disponible en freecodecamp, *Omer Rosenbaum* propone un interesante ejercicio destinado a comprender a fondo en que consiste un repositorio deGit, consistente en crearlo desde cero y añadirle un primer commit.

Creamos una nueva carpeta y accedemos a ella.

```
mkdir scratch-repo
cd scratch-repo
```

Como es previsible git status nos indica que no estamos en un repositorio de git.

```
git status
// fatal: not a git repository (or any of the parent directories): .git
```

Creamos la estructura de carpetas básica de un repositorio.

```
mkdir -p .git
cd .git
mkdir objects
mkdir refs
cd refs
```

```
mkdir heads
cd..
cd..
tree
```

Obtenemos la siguiente estructura y volvemos a comprobar el estado del repositorio.

```
C:.
└───.git
    ├───objects
    └───refs
        └───heads
```

```
git status
fatal: not a git repository (or any of the parent directories): .git
```

El paso que nos falta es crear el fichero `HEAD` en la carpeta `.git` que apunte a la rama `main`.

```
echo ref: refs/heads/main > .git\HEAD
```

En este punto, git ya reconoce la carpeta como un repositorio.

```
git status
// On branch main
// No commits yet
// nothing to commit (create/copy files and use "git add" to track)
```

A pesar de que no existe la rama main en la carpeta `refs/heads/main`, git reconoce la rama `main` como la rama actual porque la tiene referenciada en el fichero HEAD.

Gestionando el repositorio con comandos plumbing

Hemos creado el repositorio simplemente con los comandos del SO, sin utilizar `git init`. Vamos a añadir un fichero al área de preparación y hacer el primer commit, pero sin utilizar los comandos porcelain, `git add` o `git commit`. En su lugar utilizaremos los comandos plumbing que subyacen a los de más alto nivel que utilizamos habitualmente.

Para crear un objeto blob, utilizamos el comando `git hash-object`, tal y como ya hemos visto, pasándole información desde el stdin.

```
echo Aprendiendo Git | git hash-object --stdin
// 7b31213ab333bd7eab40ce5de1185bd6565f120e
```

También sabemos que el modificados -w guarda el objeto en el repositorio.

```
echo Aprendiendo Git | git hash-object -w --stdin
tree .git
dir .git\objects\7b
```

```
C:.
└───.git
    ├───objects
    │   └───7b // 31213ab333bd7eab40ce5de1185bd6565f120e
    └───refs
        └───heads
```

Se ha creado un objeto blob en la carpeta `objects` del repositorio, con una sub-carpeta `7b` y un fichero con el nombre del resto del hash del objeto. El comando `git cat-file` nos permite leer el tipo y el contenido del objeto.

```
git cat-file -t 7b31213 // blob
git cat-file -p 7b31213 // Aprendiendo Git
```

A pesar de ello, vemos que git status no refleja esta cambio

```
git status
```

Para añadir el objeto al área de preparación, necesitamos crear el indice, el fichero que contiene información sobre los hashes de los objetos que se han añadido al área de preparación. Para ello utilizaremos el comando `git update-index`.

```
git update-index --add --cacheinfo 100644 7b31213ab333bd7eab40ce5de1185bd6565f120e
README.md
```

Indicamos la mascara del fichero a nivel del SO (100644), el hash del objeto y el nombre del fichero, que aun no existe.

EL resultado a nivel de git, es

```
git status
```

```
// Changes to be committed: new file:   README.md
// Changes not staged for commit: deleted:    README.md
```

El fichero está en el area de preparación, pero no en el directorio de trabajo. Para añadirlo al directorio de trabajo, crearemos el fichero con el mismo contenido que el objeto blob.

```
echo Aprendiendo Git > README.md
```

Y comprobamos que el fichero ha sido añadido al directorio de trabajo y aparece como staged en el área de preparación, seg´´un nos indica `git status`.

```
git status

// Changes to be committed: new file:   README.md
```

Creando un commit con comandos plumbing

El paso de la información desde la staged area al repositorio exige que creemos un objeto tree que contenga la información de los objetos que se han añadido al área de preparación. Para ello utilizamos el comando `git write-tree`. Para comprobar el contenido del objeto tree, utilizamos el comando `git cat-file`.

```
git write-tree // 3976f2ec82f4d61250cac1b1b26fa053439dbcae
git cat-file -t 3976f2e // tree
git cat-file -p 3976f2e // 100644 blob 7b31213ab333bd7eab40ce5de1185bd6565f120e
README.md
```

A nivel de carpetas, veremos que se a creado un objeto tree en la carpeta `objects` del repositorio.

```
tree .git
dir .git\objects\39
```

```
C:.
└───.git
    ├───objects
    │   └───39 // 76f2ec82f4d61250cac1b1b26fa053439dbcae
    │   └───7b // 31213ab333bd7eab40ce5de1185bd6565f120e
    └───refs
        └───heads
```

A partir del objeto tree, creamos un commit con el comando `git commit-tree`. Para ello necesitamos el hash del objeto tree, que ya conocemos.

```
git commit-tree 3976f2ec82f4d61250cac1b1b26fa053439dbcae -m "Initial commit" //
79aa70c84454bd9e928f0224139170837c8563c8
```

Nuestro objeto commit puede comprobarse con `git cat-file`.

```
git cat-file -t 79aa70c // commit
git cat-file -p 79aa70c // tree 3976f2ec82f4d61250cac1b1b26fa053439dbcae
```

Y nuestro repositorio refleja su existencia a nivel de carpetas pero no en el resultado de `git status`.

```
tree .git
dir .git\objects\79
```

```
C:.
└───.git
    ├───objects
    │   └───39 // 76f2ec82f4d61250cac1b1b26fa053439dbcae
    │   └───79 // aa70c84454bd9e928f0224139170837c8563c8
    │   └───7b // 31213ab333bd7eab40ce5de1185bd6565f120e
    └───refs
        └───heads
```

```
git status
// Changes to be committed:  new file:   README.md
```

El problema es simplemente que no se ha actualizado el puntero de la rama `main` al nuevo commit. En realidad, el fichero refs/heads/main al que hace referencia HEAD ni siquiera existe. Debemos crearlo con el hash del nuevo commit.

```
echo 79aa70c84454bd9e928f0224139170837c8563c8 > .git\refs\heads\main
```

De esta forma tanto `git status` como `git log` reflejarán la existencia del nuevo commit.

```
git status
// nothing to commit, working tree clean
```

```
git log
// commit 79aa70c84454bd9e928f0224139170837c8563c8 (HEAD -> main)
```

En resumen del Taller

Crear un nueva repositorio desde cero ha sido posible gracias a los comandos del SO y a los comandos de bajo nivel de Git, los **plumbing commands**.

Hemos creado un objeto blob, un objeto tree y un objeto commit, y hemos actualizado los punteros de la rama y HEAD para reflejar la existencia del nuevo commit. Pare ello hemos usado los siguientes comandos:

- `git hash-object`: para crear un objeto blob
- `git update-index`: para añadir el objeto al área de preparación
- `git write-tree`: para crear un objeto tree
- `git commit-tree`: para crear un objeto commit
- `git cat-file`: para leer el contenido de los objetos

HERRAMIENTAS PARA PREPARAR UN BUEN COMMIT EN CUALQUIER SITUACIÓN

Comprobar el repositorio. Git log

```
git log
git log --graph --decorate --oneline
```

Log es uno de los comandos de git con más opciones. Algunas de las más útiles son

- `--graph`: Muestra el historial de commits en forma de grafo
- `--decorate`: Muestra las referencias de los commits (HEAD, master, ...)
- `--oneline`: Muestra los commits en una sola línea
- `--all`: Muestra todos los commits, no solo los de la rama actual
- `--author`: Filtra los commits por autor
- `--since`: Filtra los commits por fecha
- `--until`: Filtra los commits por fecha
- `--grep`: Filtra los commits por mensaje
- `--no-merges`: Muestra solo los commits que no son merges
- `--stat`: Muestra estadísticas de los cambios en los commits
- `--patch`: Muestra los cambios en los commits

Se suelen combinar varias opciones para obtener la información deseada

- `--graph --oneline --decorate --all`: Muestra el historial de commits en forma de grafo, en una sola línea, con las referencias y todos los commits

Si estas combinaciones se utilizan con frecuencia, se pueden añadir a la configuración de git como alias

```
git config --global alias.lol "log --graph --decorate --oneline"
git lol
```

Se puede indicar a partir de que commit debe de empezar la serie git log

```
git log master~2
```

Se puede especificar un rango (desde .. hasta) git log ..

```
git log master~12..master~10
```

Se pueden mostrar los commits que afectan a un determinado path (carpeta, fichero...) git log --

```
git log -- README3.txt
```

Se pueden mostrar los commits que coincidan con una expresión regular git log --grep='reg-exp'

Se pueden excluir del listado los commits resultantes de un merge

```
git log --no-merges
```

Se pueden mostrar los cambios producidos durante un tiempo

```
git log --since={2010-04-18}
git log --before={2010-04-18}
git log --after={2010-04-18}
```

Dado lo complejo de la sintaxis de git log, y las limitaciones gráficas de la consola, se pueden utilizar herramientas gráficas para visualizar el historial de commits.

Por ejemplo, **gitk**, que se instala con Git, o **gitg**, que es una herramienta gráfica de Git para Gnome.

En **VSC** se puede utilizar la extensión **Git Graph**

Alias

Los alias fueron añadidos en Git 1.4.0

Evitan tener que teclear repetidas veces el mismo comando completo

Se pueden configurar en cualquiera de los ámbitos de configuración mencionados (system, global, local) tanto por línea de comandos como directamente en el fichero de configuración correspondiente

```
git config --global alias.lol "log --graph --decorate --oneline"
git lol
```

Un ejemplo de alias más complejo

```
git config --global alias.hist git log --pretty=format:"%h %ad | %s%d [%an]" --
graph --date=short
git hist
```

Los alias de git admiten los mismos parámetros y modificadores que los comandos originales

```
git config --global alias.ch git checkout
git ch -b feature/branch
```

Operaciones en la Staging Area (Index)

Añadir ficheros

Como ya hemos visto, el comando `git add` añade ficheros al área de preparación (staging area). Se puede añadir un solo fichero, todos los ficheros de un directorio o todos los ficheros del directorio de trabajo.

```
md samples
echo 'Sample One' > samples\sample1.txt
echo 'Sample Two' > samples\sample2.txt
tree samples /f
```

Se puede añadir al index

- un conjunto de ficheros
- todos los disponibles

```
git add <file>
git add .
git status
```

Eliminar de la Staging Area (Index)

Eliminar elementos de la zona de preparación (staging area), i.e. revertir add, se puede hacer de varias formas

Cuando aun no se ha hecho commit, se utiliza un formato especial, ya que aún no existe el HEAD

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

Si ya se ha hecho algún commit, el comando anterior, ademas de eliminar el fichero de la staging area, lo elimina del directorio de trabajo.

En estas circunstancias, se puede utilizar

```
git reset <file>
```

O la opción moderna, recomendada en el git status

```
git restore --staged <file>
```

git restore es un nuevo comando que reproduce una parte de la funcionalidad de git checkout: recover an earlier commit.

Una variación de git reset permite eliminar un archivo de la staging area y enviando al directorio de trabajo su estado en el último commit

```
git reset HEAD <file>
```

Por otra parte, el comportamiento de git checkout con ficheros trazados por git (tracked) es un poco diferente

- git checkout -- path: el path se toma como un fichero o directorio, si es un directorio, significa todos los ficheros dentro de ese directorio, recursivamente, y Git copia la copia actual del fichero tal como se encuentra en el índice a su árbol de trabajo.
- git checkout HEAD -- path: el path se toma en la misma forma que antes, y Git copia la copia actual del fichero tal como se encuentra en el commit HEAD al índice y al árbol de trabajo.

```
git checkout -- <file>
git checkout HEAD -- <file>
```

Eliminar ficheros

Puede hacerse en dos fases

- eliminar de la workArea (mediante el SO)
- subir el cambio a la Staging Area

```
del samples\sample1.txt
git status
```

```
git add samples\sample1.txt
git status
```

Es preferible hacerlo en un solo paso Al ser un borrado definitivo, es necesario el modificador f

```
git status
git rm samples\sample2.txt
git status
git rm -f samples\sample2.txt
git status
```

Problemas con .gitignore

En algún caso puede que añadamos a git ignore nuevos elementos que ya estaban en el repositorio. En este caso, git no los ignorará, ya que ya los conoce. Para que los ignore, hay que eliminarlos del repositorio

Con el comando rm podremos borrar los archivos del repositorio, pero si lo ejecutamos tal cual nos eliminará también el archivo de nuestro directorio de trabajo.

Si queremos conservarlo tendríamos que poner lo siguiente:

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

Si lo que queremos eliminar es un directorio con todo su contenido el comando sería el siguiente:

```
git rm -r --cached <directory>
```

El modificador -r indica que se trata de un directorio procesado de forma recursiva, es decir con todos sus ficheros y directorios, con todos los niveles de anidamiento que pueda tener.

En este punto tendremos pendiente de commit la eliminación del archivo o carpeta del repositorio, por lo que tendremos que hacer un commit para que se aplique el cambio.

```
git commit -m "Eliminado archivo de referido en .gitignore"
```

Finalmente actualizaremos con nuestros cambios el remoto

```
git push
```

Cambiar nombre de ficheros

```
md samples
echo 'Sample One' > samples\sample_bad1.txt
echo 'Sample Two' > samples\sample_bad2.txt
tree samples /f
git add .
```

Igual que en el caso anterior, puede hacerse en un paso o en dos

En dos fases

- cambiar el nombre en la workArea (mediante el SO)
- subir el cambio a la Staging Area

```
ren samples\sample_bad1.txt sample1.txt
git status
git add samples\sample_bad1.txt
git add samples\sample1.txt
git status
```

En una sola fase

```
git mv samples\sample_bad2.txt samples\sample2.txt
git status
```

git diff

Permite ver los cambios entre dos commits, dos ramas, dos ficheros, etc.

Por defecto, compara el directorio de trabajo con el index (staging area)

```
git diff
```

Para comparar el directorio de trabajo con el último commit

```
git diff HEAD
```

Para comparar el directorio de trabajo con un commit concreto

```
git diff <commit>
```

Para comparar dos commits

```
git diff <commit1> <commit2>
```

Para comparar dos ramas

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

Para comparar dos ficheros

```
git diff --no-index <file1> <file2>
```

En todos los casos, git tiene que calcular las diferencias, por lo que el resultado no es inmediato. Hay que recordar que en git no se almacenan las diferencias, sino los estados de los ficheros en cada commit.

git blame

Permite conocer el autor de la última modificación de cada línea de un fichero y en que commit se incluyó el cambio.

```
git blame <file>
```

Es una herramienta muy útil para la revisión de código, que permite quién ha hecho un cambio en un fichero y para saber por qué se ha hecho un cambio.

Esta también disponible cuando se accede al repositorio en GitHub

Recapitulando: Git básico

- `git config`: Configura Git; permite añadir alias
- `git init`: Inicializa un repositorio local de Git
- `git clone`: Clona un repositorio de Git
- `git status`: Muestra el estado de los archivos en el directorio de trabajo (workArea) y el área de preparación
- `git add`: Agrega un archivo al área de preparación (stagedArea)
- `git commit`: Guarda los cambios en el repositorio
- `git log`: Muestra el historial de commits
- `git show`: Muestra los cambios de un commit
- `git rm`: Elimina un archivo del repositorio
- `git mv`: Cambia el nombre de un archivo
- `git reset`: Mueve el puntero de la rama; en consecuencia restablece el estado de los archivos al commit indicado

    - `git reset <file>`: Elimina el fichero de la staging area
- `git checkout <file>` mueve los ficheros entre el repositorio, el área de preparación y el directorio de trabajo
    - `git restore --staged <file>`: Elimina el fichero de la staging area
- `git diff`: Muestra las diferencias entre dos commits, dos ramas, dos ficheros, etc.
- `git blame`: Muestra el autor de la última modificación de cada línea de un fichero y en que commit se incluyó el cambio
- `git remote`: Conecta el repositorio local con un repositorio remoto
- `git push`: Sube todos los cambios locales al repositorio remoto
- `git pull`: Descarga los cambios del repositorio remoto al repositorio local
-

REESCRIBIENDO LA HISTORIA

Advertencia

La *ley de oro de Git* dice que **no se deben modificar commits que ya han sido compartidos**

Veremos que pasa en caso de hacerlo, cuando estemos hablando de los repositorios remotos.

git command --amend

El comando `git commit --amend` permite modificar el último commit. Se puede utilizar para

- cambiar el mensaje del commit
- añadir ficheros al commit
- modificar ficheros del commit

```
git commit --amend
```

Si no se añade ningún fichero, se abrirá el editor de texto para modificar el mensaje del commit. Si han añadido ficheros en el stage, se añadirán al commit.

```
git add <file>
git commit --amend
```

En realidad, `git commit --amend` crea un nuevo commit con los cambios del commit anterior y los nuevos cambios. El commit anterior dejara de estar vinculado a ninguna rama y se eliminará en la proxima operación del garbage collector.

El uso de amend debe limitarse a "cambios en el commit anterior", como corregir errores en el mensaje del commit, modificar los ficheros o añadir puntualmente otros nuevos, claramente vinculados a los anteriores. Para cambios de mayor envergadura es mejor práctica crear un nuevo commit.

git checkout

El comando `git checkout` mueve el puntero de referencia HEAD a un commit específico.

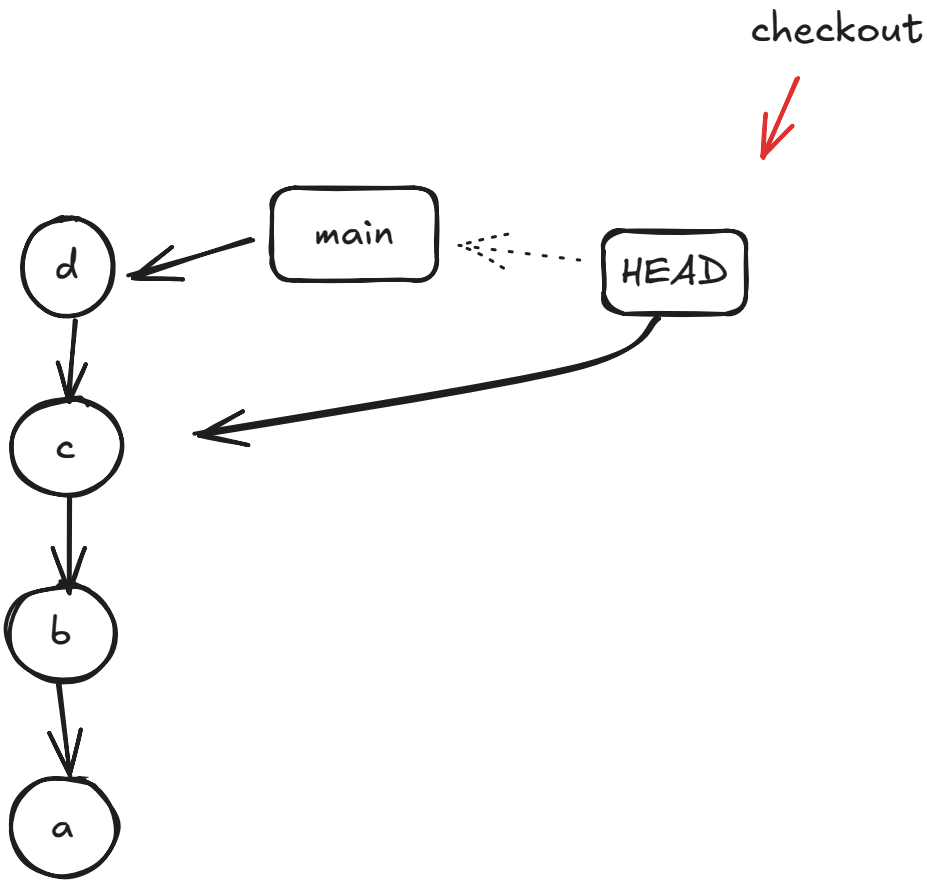
- en otra rama (lo habitual)
- en la misma rama. Veamos este segundo caso

```
git checkout HEAD~1

You are in 'detached HEAD' state. You can look around, make experimental changes
and commit them, and you can discard any commits you make in this state without
impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may do so
(now or later) by using -c with the switch command. Example: git switch -c [new-
branch-name]

Or undo this operation with: git switch -
```



En este caso, el puntero HEAD se ha movido al commit anterior al último commit de la rama actual. El mensaje indica que estamos en un estado de HEAD desacoplado (detached HEAD), lo que significa que no estamos en una rama y que cualquier commit que hagamos no estará vinculado a ninguna rama.

En este estado, podemos consultar el estado del proyecto en ese commit, que se reflejara en la working area. Igualmente podemos hacer cambios experimentales, pero si queremos conservarlos, debemos crear una nueva rama desde el último commit desacoplado.

```
git checkout -b new-branch
```

Si no queremos conservar los cambios, podemos volver a la rama más adelantada con alguno de los comandos

```
git switch -
git checkout -
```

Igualmente podemos volver a cualquier rama con alguno de los comandos

```
git switch <branch>
git checkout <branch>
```

git checkout a nivel de archivo

En lugar de mover el HEAD del repositorio, lo que hace es llevar al directorio de trabajo el fichero al que hemos hecho checkout con el contenido que tenía en el commit especificado

```
git checkout HEAD~1 README.md
```

El resultado es que el fichero README.md en el area de trabajo vuelve a tener el contenido que tenía en el commit anterior al HEAD.

```
git checkout HEAD~1 README.md --stage
```

En este caso, el fichero README.md en el área de preparación vuelve a tener el contenido que tenía en el commit anterior al HEAD.

Día 3

REESCRIBIENDO LA HISTORIA (2)

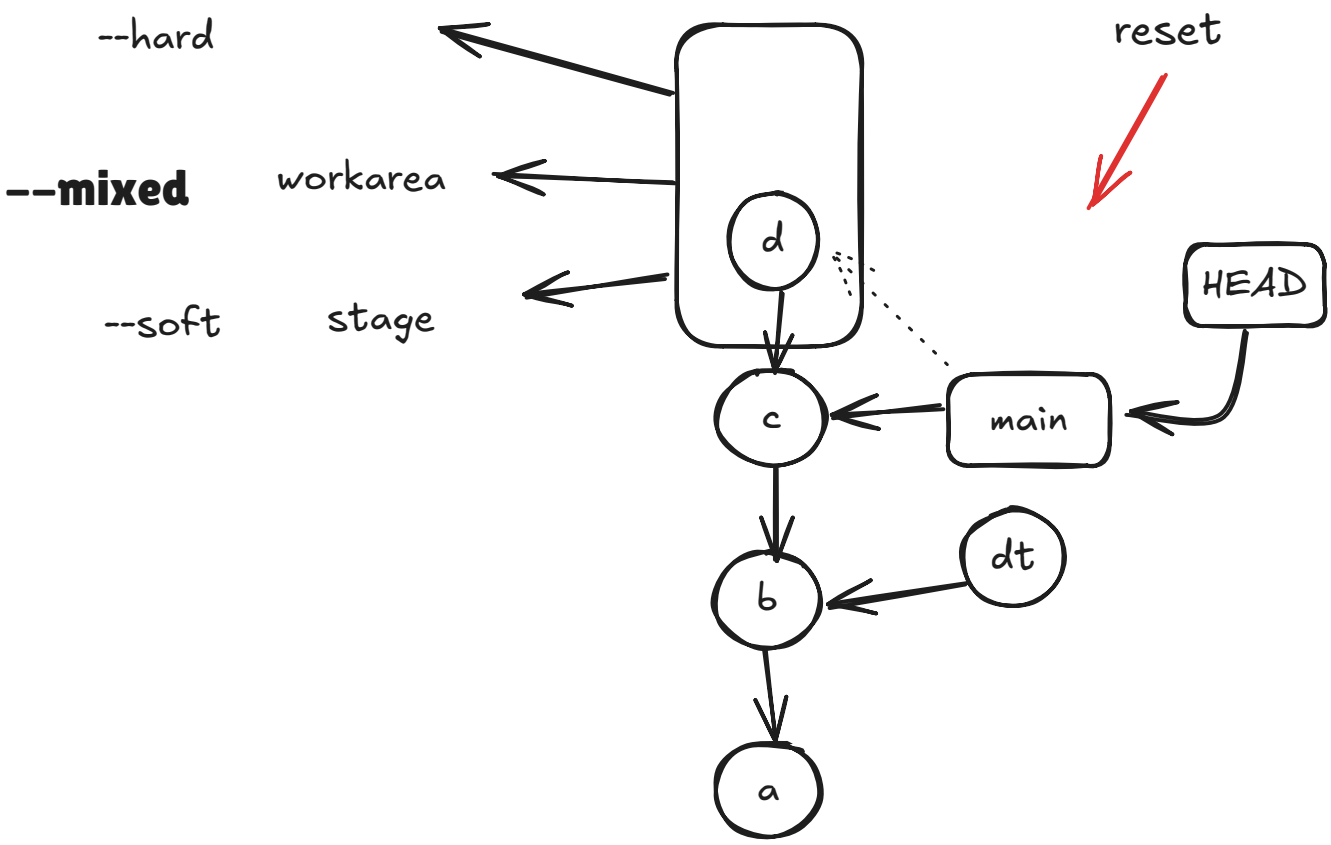
git reset

El comando `git reset` mueve el puntero de referencia de una rama (acompañado por el HEAD), a un commit específico, normalmente un commit anterior de la misma rama. Estaremos 'deshaciendo' los commits posteriores que quedarán huérfanos y se eliminarán la próxima vez que Git haga limpieza.

Sus efectos sobre la working y staging areas dependen de la opción seleccionada:

- hard: el contenido del commit apuntado por la rama se refleja en la working y staging areas

- mixed: (valor por defecto) el contenido del commit apuntado por la rama se refleja en la working area
- soft: no se modifican la working y staging areas. Cambia todos los archivos a "Cambios a ser committed".



git reset a nivel de archivo

git reset HEAD [file]

En este caso no mueve el HEAD del repositorio, lo que hace es llevar al directorio de staged el fichero al que hemos hecho reset con el contenido que tenía en el último commit. Como se una el parámetro por defecto, mixed, en el directorio de trabajo estará la versión última del contenido pendiente de commit y en staged la versión del contenido a la que hemos vuelto.

rebase interactivo

Uno de los comando más potentes de Git es el rebase interactivo. Permite reescribir la historia de un repositorio, cambiando el orden de los commits, modificando los mensajes de los commits, eliminando commits, fusionando commits, etc.

```
git rebase -i <primer commit no incluido en el rebase >
git rebase -i HEAD~4
```

Como resultado se abre un editor de texto con una lista de commits que se van a reescribir. Cada línea tiene un comando y un commit.

```
pick 1ecb721 Mensaje del commit HEAD~3
pick 07ad6a9 Mensaje del commit HEAD~2
pick 7515002 Mensaje del commit HEAD~1
pick 7c4668d Mensaje del último commit

# Rebase 9457f43..7c4668d onto 9457f43 (4 commands)
#
# Commands:

# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
# commit's log message, unless -C is used, in which case
# keep only this commit's message; -c is same as -C but
# opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# create a merge commit using the original merge commit's
# message (or the oneline, if no original merge commit was
# specified); use -c <commit> to reword the commit message
# u, update-ref <ref> = track a placeholder for the <ref> to be updated
# to this position in the new commits. The <ref> is
# updated at the end of the rebase
#

# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
```

Los comandos más comunes son

- pick: utiliza el commit
- reword: utiliza el commit, pero permite cambiar el mensaje
- edit: utiliza el commit, pero para en él para hacer cambios
- squash: fusiona el commit con el anterior
- fixup: fusiona el commit con el anterior, pero mantiene el mensaje del anterior
- drop: elimina el commit

También es posible reordenar los commits simplemente cambiando el orden de las líneas.

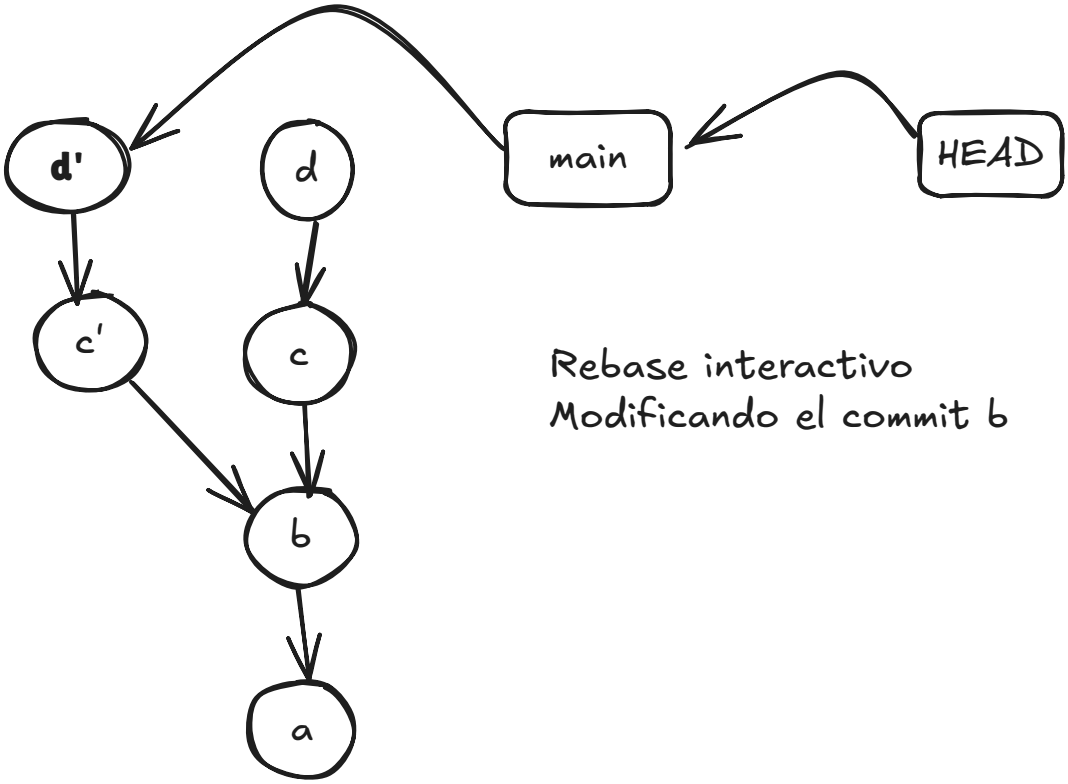
A partir del primer commit que cambia, todos los commits se deben reescribir, dado que su inmutabilidad impide ningún cambio en ellos.

Por ejemplo si se quiere cambiar el mensaje de un commit, se cambia el comando pick por reword. Al cerrar el editor se abrirá otro editor con el mensaje del commit, que se puede modificar.

```
git rebase -i HEAD~4
```

```
pick 1ecb721 Mensaje del commit HEAD~3
reword 07ad6a9 Mensaje del commit HEAD~2
pick 7515002 Mensaje del commit HEAD~1
pick 7c4668d Mensaje del último commit
```

```
Mensaje del commit HEAD~2
## Información de que estamos en el editor de reword
```



edit: modificando un commit

Edit utiliza el commit indicado para hacer cambios, incluyendo eliminar parte del contenido, añadir más contenido o separar el contenido en diversos commits. Al cerrar el editor, se abrirá una consola de Git en la que se pueden hacer los cambios deseados.

```
git rebase -i HEAD~4
```

```
pick 1ecb721 Mensaje del commit HEAD~3
pick 07ad6a9 Mensaje del commit HEAD~2
edit 7515002 Mensaje del commit HEAD~1
pick 7c4668d Mensaje del último commit
```

```
Stopped at 7515002...  Mensaje del commit HEAD~1
You can amend the commit now, with

  git commit --amend

Once you are satisfied with your changes, run

  git rebase --continue
```

La salida de git status nos indicara en que commit estamos y que podemos hacer cambios en él.

```
interactive rebase in progress; onto 9457f43
Last command done (1 command done):
   edit 1ecb721 Add new Huskies
Next commands to do (3 remaining commands):
   pick 07ad6a9 Add and configure github action
   pick 7515002 Simulate lint fail
  (use "git rebase --edit-todo" to view and edit)
You are currently editing a commit while rebasing branch 'feature/actions' on
'9457f43'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

nothing to commit, working tree clean
```

Como estamos editando el commit HEAD~1, empezaremos por un reset con su valor por defecto --mixed enviando todo su contenido al area de trabajo.

```
git reset HEAD~1
```

A partir de aquí podemos hacer los cambios que deseemos, añadir ficheros, eliminar ficheros, modificar ficheros, etc. y almacenarlos en tantos commits como necesitemos.

```
git add <files>
git commit -m "Mensaje del primer commit modificado"
git add <other files>
git commit -m "Mensaje del segundo commit modificado"
```

Una vez que hemos terminado de hacer los cambios, podemos continuar con el rebase

```
git rebase --continue
```

squash y fixup: fusionando commits

Si se quiere fusionar dos o más commits, se puede utilizar squash o fixup. La diferencia entre ambos es que squash mantiene el mensaje del commit que se va a fusionar, mientras que fixup lo elimina.

```
git rebase -i HEAD~4
```

```
pick 1ecb721 Mensaje del commit HEAD~3
pick 07ad6a9 Mensaje del commit HEAD~2
squash 7515002 Mensaje del commit HEAD~1
pick 7c4668d Mensaje del último commit
```

El comando **squash** fusiona el commit HEAD~1 con el commit anterior, HEAD~2. Al cerrar el editor, se abrirá otro editor con el mensaje de los dos commits, que se puede modificar, Seleccionando alguno de los mensajes o creando uno nuevo.

Por su parte el comando **fixup** fusiona el commit HEAD~1 con el commit anterior, HEAD~2, pero mantiene el mensaje del commit anterior, sin darnos opción a cambiarlo.

drop: eliminando un commit

Al indicar drop, el commit desaparece de la lista de commits a reescribir.

```
git rebase -i HEAD~4
```

```
pick 1ecb721 Mensaje del commit HEAD~3
pick 07ad6a9 Mensaje del commit HEAD~2
drop 7515002 Mensaje del commit HEAD~1
pick 7c4668d Mensaje del último commit
```

El resultado es que el commit HEAD~1 desaparece de la lista de commits a reescribir. Como consecuencia, la información de ese commit se pierde completamente.

Un posible caso sería eliminar información sensible que por error se ha incluido en un commit.

Ref logs

El reflog es un registro de los cambios en los punteros de referencia (HEAD, ramas, etc.) que se han producido en el repositorio. Se puede consultar con el comando `git reflog`

```
git reflog
```

También se puede obtener la lista de todos los logs en los que se guarda la información de los cambios en los punteros de referencia, dentro de la carpeta logs del repositorio.

```
git reflog list
```

Esta información es muy util para conocer los commits "eliminados", es decir aquellos a los que ya no se puede acceder directamente desde las ramas, pero que siguen existiendo en el repositorio.

Utilidades gráficas como `gitk` o la extensión `Git Graph` de Visual Studio Code permiten visualizar la información de los reflogs de una manera más intuitiva para el usuario.

Otros comandos

- git stash: guarda los cambios en un commit temporal, que se almacena en una pila de cambios. Se puede recuperar en cualquier momento.
- git clean: elimina los ficheros no rastreados por Git
  - git clean -n: muestra los ficheros que se eliminarán
  - git clean -f: elimina los ficheros
- git revert: crea un nuevo commit que deshace los cambios de un commit anterior
- git bisect: busca un commit que introdujo un error

Git stash

El comando `git stash` permite guardar los cambios en un commit temporal, que se almacena en una pila de cambios. Se puede recuperar en cualquier momento.

```
git stash
git stash list
```

Suben al stash todos los cambios no confirmados, pero no los archivos no rastreados (untracked files). Para incluirlos, se utiliza el modificador -u

```
git stash -u
```

Para recuperar los cambios guardados, se utiliza el comando `git stash apply` o `git stash pop`. La diferencia entre ambos es que `git stash apply` deja el cambio en la pila de cambios, mientras que `git stash pop` lo elimina de la pila.

```
git stash apply
git stash pop
```

Una vez que se han recuperado los cambios, se puede eliminar el commit temporal con el comando `git stash drop` o `git stash clear`. La diferencia entre ambos es que `git stash drop` elimina el commit temporal más reciente, mientras que `git stash clear` elimina todos los commits temporales.

```
git stash drop
git stash clear
```

Git clean

El comando `git clean` permite eliminar los ficheros no rastreados por Git. Es decir, los ficheros que no están en el área de preparación ni en el último commit.

```
git clean -n
```

Muestra los ficheros que se eliminarán, sin eliminarlos realmente.

```
git clean -f
```

Elimina los ficheros no rastreados por Git.

El modificador -d indica que se eliminarán también los directorios no rastreados por Git.

```
git clean -fd
```

Git revert

El comando `git revert` permite crear un nuevo commit que deshace los cambios de un commit anterior. A diferencia de `git reset`, que mueve el puntero de la rama a un commit anterior, `git revert` crea un nuevo commit que deshace los cambios del commit especificado.

```
git revert <commit>
```

Crea un nuevo commit que deshace los cambios del commit especificado.

```
git revert HEAD~1
```

Por ejemplo, el comando anterior, crea un nuevo commit que deshace los cambios del commit anterior al último commit.

Git bisect

El comando `git bisect` permite buscar un commit que introdujo un error en el código. Utiliza una búsqueda binaria para encontrar el commit problemático de manera eficiente.

```
git bisect start
git bisect bad # Marca el commit actual como malo
git bisect good <commit> # Marca un commit anterior como bueno
```

A partir de aquí, Git hará un checkout de un commit intermedio y te permitirá probar si el error está presente o no. Debes usar `git bisect good` o `git bisect bad` para marcar cada commit como bueno o malo hasta que encuentres el commit problemático.

Cuando hayas encontrado el commit problemático, puedes usar el comando `git bisect reset` para volver al estado original del repositorio.

```
git bisect reset
```

TRABAJANDO EN PARALELO

Ramas (branches)

Crear y seleccionar ramas

- `git branch`: Lista las ramas locales
- `git branch nombre_rama`: Crea una nueva rama
- `git checkout nombre_rama`: Cambia a la rama especificada
- `git checkout -b nombre_rama`: Crea una nueva rama y cambia a ella

Se puede crear una rama a partir de un commit específico

- `git branch nombre_rama <commit>`: Crea una nueva rama a partir del commit especificado
- `git checkout -b nombre_rama <commit>`: Crea una nueva rama a partir del commit especificado y cambia a ella

Ejemplos con checkout

- Crear desde HEAD (`git checkout -b mybranch`)
- Crear desde rama (`git checkout -b mybranch master`)
- Crear desde commit (`git checkout -b mybranch <commit>` (e.g. `git checkout -b mybranch master~1`))

Ver las ramas

- `git branch`: Lista las ramas locales
- `git branch -a`: Lista todas las ramas, locales y remotas
- `git branch -r`: Lista las ramas remotas
- `git branch --merged`: Muestra las ramas que han sido fusionadas en la rama actual
- `git branch --no-merged`: Muestra las ramas que no han sido fusionadas en la rama actual
- `git branch --contains <commit>`: Muestra las ramas que contienen el commit especificado

Borrar ramas

- `git branch -d nombre_rama`: Borra la rama especificada (solo si ha sido fusionada)
- `git branch -D nombre_rama`: Borra la rama especificada (incluso si no ha sido fusionada)

Mover y renombrar ramas

- `git branch -m nombre_rama`: Cambia el nombre de la rama actual
- `git branch -m nombre_rama_antiguo nombre_rama_nuevo`: Cambia el nombre de la rama especificada

Para mover una rama a otro commit puede forzarse que se vuelva a crear en la posición deseada

```
git branch -f nombre_rama_existente <nuevo commit>
```

Otra posibilidad es situarse en la rama y hacer un reset al commit deseado

```
git checkout nombre_rama_existente
git reset --hard <nuevo commit>
```

Otras operaciones con ramas

- `git branch --set-upstream-to=origin/nombre_rama`: Establece la rama remota de seguimiento para la rama actual
- `git branch --unset-upstream`: Elimina la rama remota de seguimiento para la rama actual

Combinación de ramas: Merge y Rebase

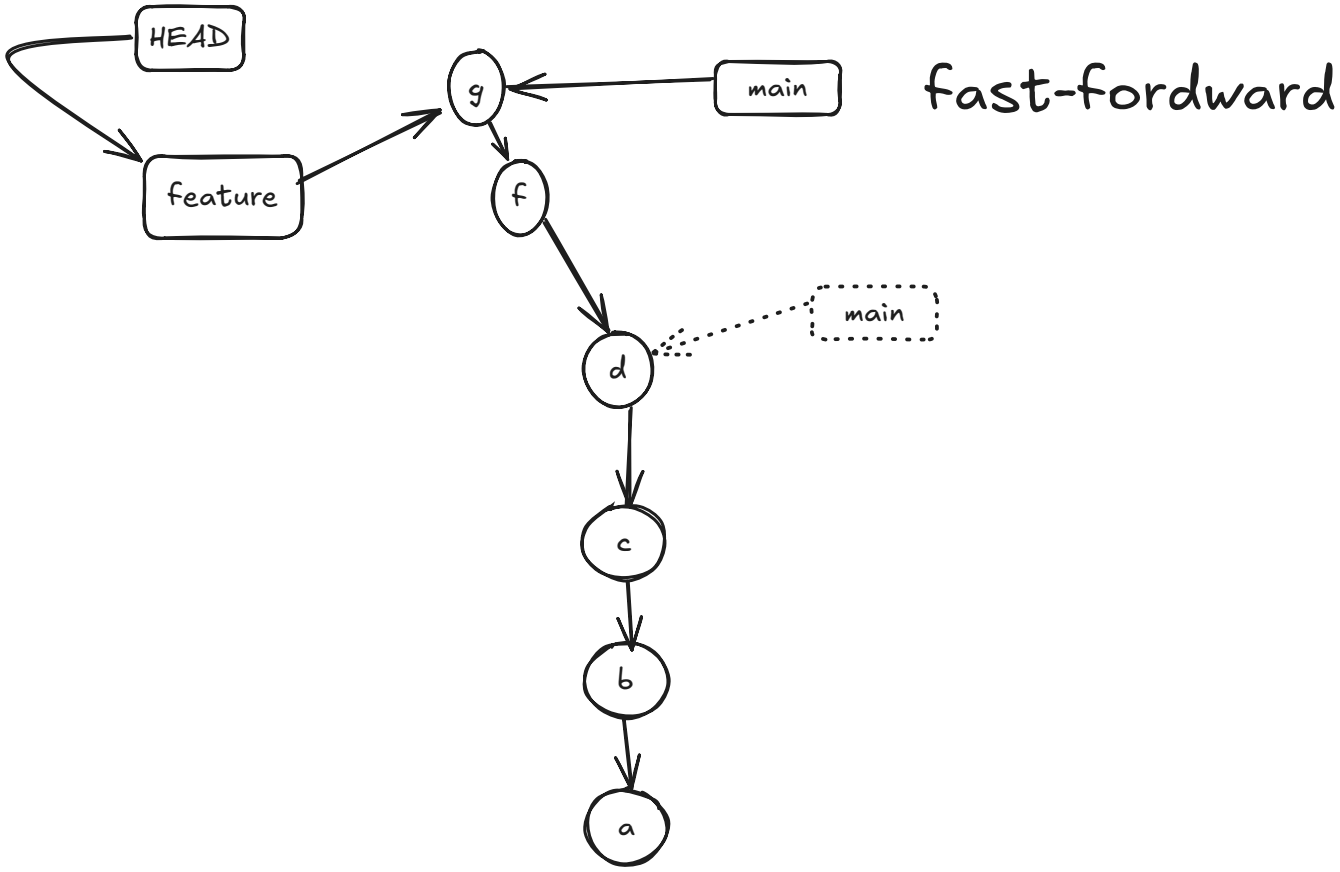
El destino final de una rama suele ser la fusión con la rama principal. Para ello se utilizan dos estrategias, basadas en distintos comandos

- merge con el comando `git merge`
- rebase con el comando `git rebase`

Merge fast-forward

El merge fast-forward es una estrategia de merge que se utiliza cuando la rama que se va a fusionar (e,g. main) no tiene commits que no estén en la rama de destino. En este caso, Git no crea un commit de merge, sino que mueve el puntero de la rama de destino al último commit de la rama que se va a fusionar.

```
git checkout -b feature/branch
echo 'Feature One' > feature1.txt
git add feature1.txt
git commit -m "Add feature one"
git checkout main
git merge feature/branch
```

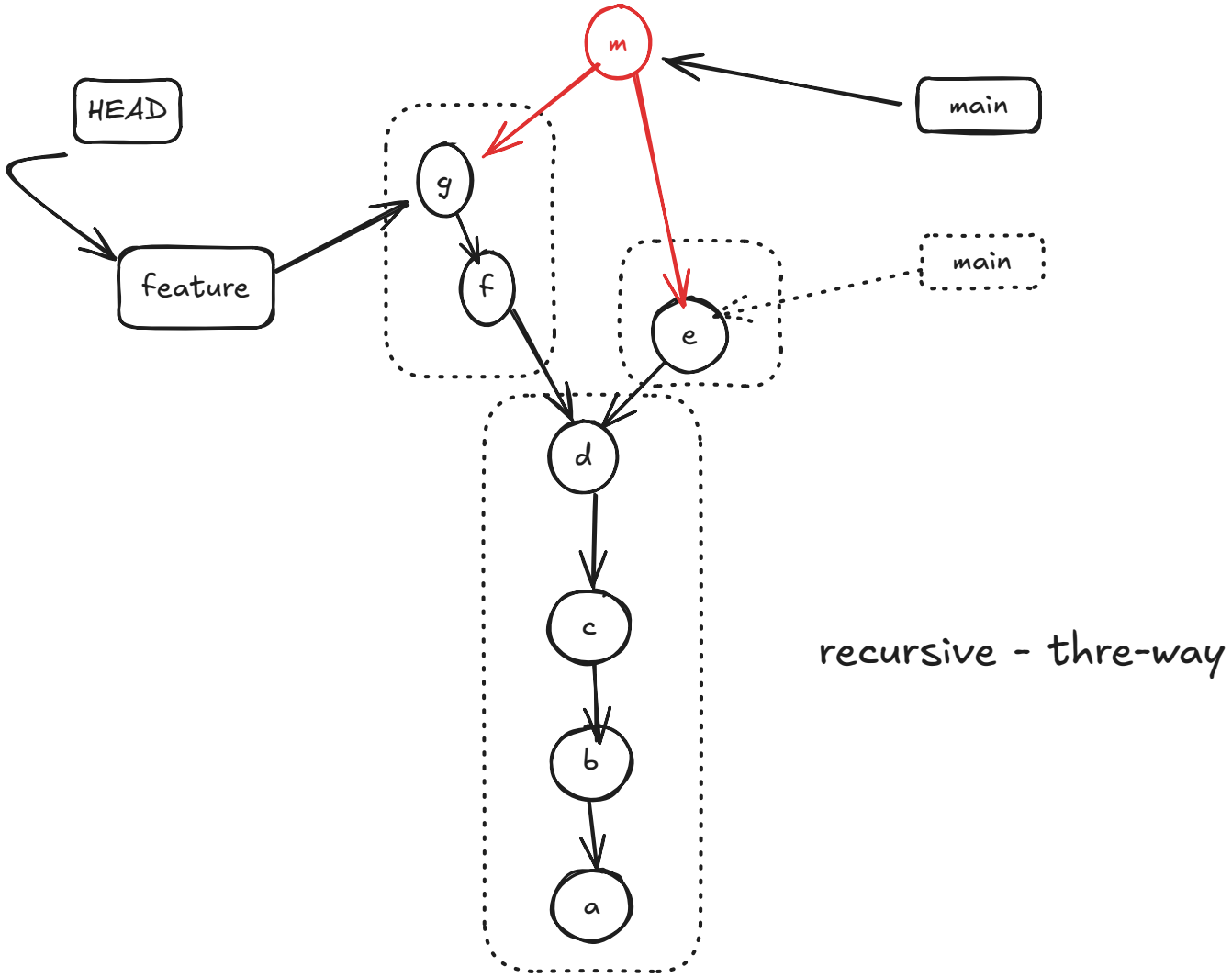


Merge recursive

El merge recursive es una estrategia de merge que se utiliza cuando la rama que se va a fusionar tiene commits que no están en la rama de destino. En este caso, Git crea un commit de merge, que tiene dos padres, uno de la rama de destino y otro de la rama que se va a fusionar.

```
git checkout -b feature/branch
echo 'Feature Two' > feature2.txt
git add feature2.txt
git commit -m "Add feature two"
git checkout main
echo 'Feature Three' > feature3.txt
git add feature3.txt
```

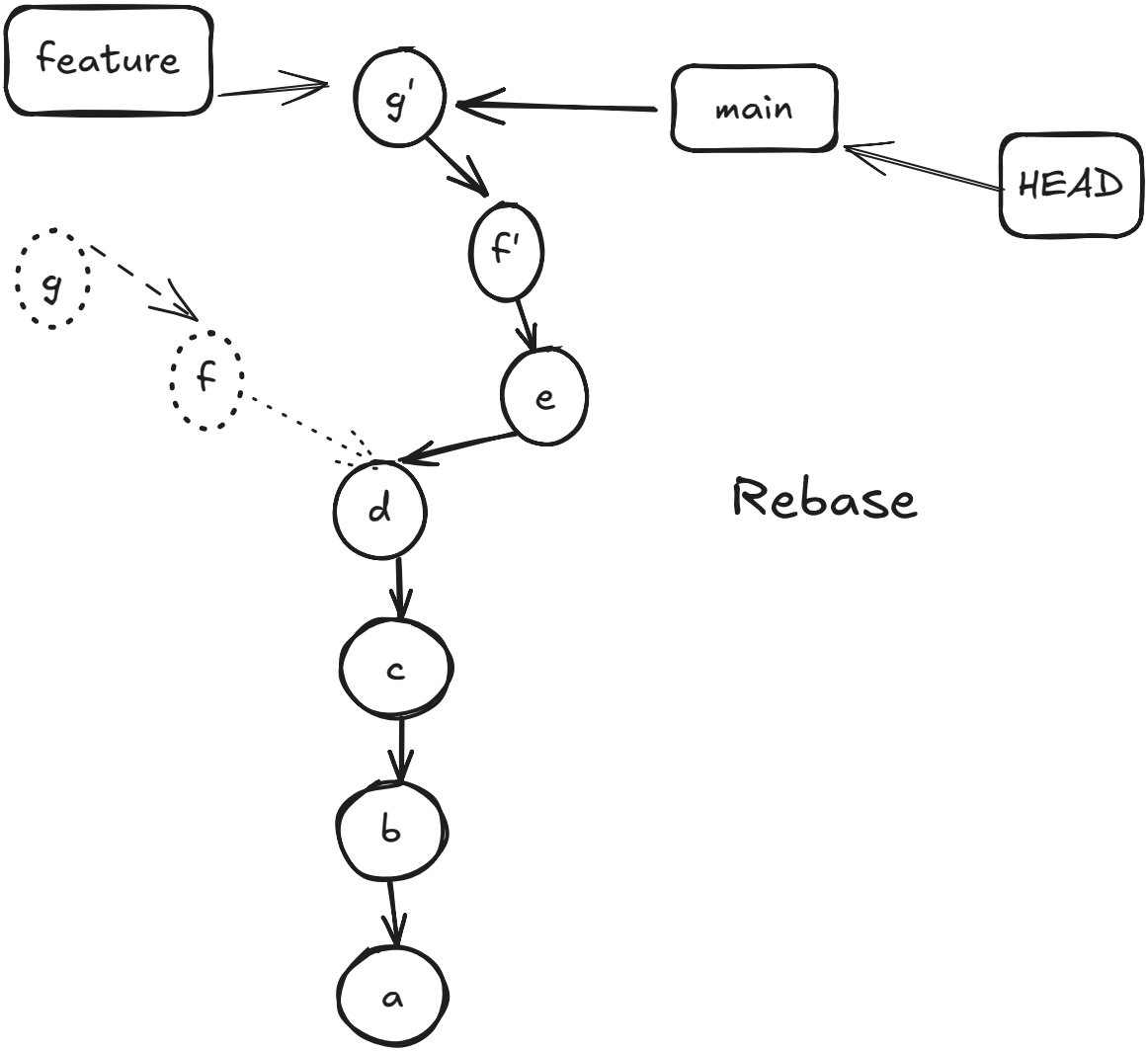
```
git commit -m "Add feature three"
git merge feature/branch
```



Rebase

El rebase es una estrategia de merge que se utiliza para reescribir la historia de una rama, cambiando el orden de los commits, modificando los mensajes de los commits, eliminando commits, fusionando commits, etc.

```
git checkout -b feature/branch
echo 'Feature Four' > feature4.txt
git add feature4.txt
git commit -m "Add feature four"
git checkout main
echo 'Feature Five' > feature5.txt
git add feature5.txt
git commit -m "Add feature five"
git rebase feature/branch
```



Resolución de conflictos

Los conflictos surgen durante un merge recursivo o un rebase, cuando dos commits han modificado la misma parte de un fichero. Git no puede decidir cuál de los dos cambios es el correcto, por lo que deja la resolución del conflicto al usuario.

El editor de texto asociado a git muestra el fichero con los conflictos, que se indican con una estructura:

```
<<<<<<<
Código de la rama actual
=======
Código de la rama que se va a fusionar
>>>>>>>.
```

El usuario debe decidir qué cambios se deben mantener: los actuales, los entrantes o ambos y el editor eliminará los marcadores de conflicto.

En algún caso, si queda contenido fuera de stage que se quiere incluir, puede ser necesario hacer un add.

Finalmente, se debe hacer un commit para finalizar la resolución del conflicto. En este caso no se indicara mensaje y git abrirá de nuevo el editor con el mensaje por defecto de un commit de fusión, que se puede

modificar.

```
git add <file>
git commit
```

Cherrypick

El comando cherry-pick "copia" un commit, creando un nuevo commit en el branch actual con el mismo mensaje y patch que otro commit. Es un rebase de un solo commit.

Si en la rama actual se quiere añadir un commit de otra rama, se puede hacer con el comando `git cherry-pick`

```
git cherry-pick <commit>
```

La principal utilidad de cherry-pick es la de añadir a una o varias ramas un hotfix (solución de un problema urgente) que se ha hecho en otra rama, sin tener que hacer un merge.

El hotfix debe ser un commit atómico, es decir, que no dependa de otros commits. Esto sería una buena práctica en cualquier caso, pero en el caso de un hotfix es imprescindible.

Día 4

TRABAJANDO EN PARALELO (2)

¿Qué son los repositorios remotos?

Un repositorio remoto es una versión del proyecto que se encuentra alojada en un **servidor** (cualquier otro ordenador). Puede ser útil para

- colaborar con otras personas en un proyecto, ya que les permite enviar cambios al proyecto y recibir cambios del proyecto.
- mantener una copia del proyecto en un servidor remoto, para tener una copia de seguridad o para trabajar en diferentes ordenadores.

EL "servidor" puede ser

- un **hosting** de repositorios Git, como **GitHub**, **GitLab** o **Bitbucket**
- un servidor propio, en el que se instalara un servidor Git como **GitLab**, **Gitea** o **Gogs**

Los **repositorios remotos**, alojados en los servidores, cualquier que sea su tipo, son una versión de repositorio algo diferente, conocida como **repositorio bare**.

Se denominan bare porque no tienen working area, es decir, no tienen los ficheros del proyecto, solo los metadatos de Git.

Para crear un repositorio remoto en el servidor, se puede utilizar el comando `git init` con la opción `--bare`

```
git init --bare
```

Normalmente es el software del servidor el que se encarga de crear el repositorio remoto, por lo que no es necesario hacerlo manualmente.

Por ejemplo en github, se crea un repositorio completando un **formulario** en la web, indicando el nombre del repositorio, si es público o privado, si tiene un README.md, etc.

Clonado de repositorios

El proceso de obtener localmente una copia de un repositorio remoto se denomina **clonado**. Se realiza con el comando `git clone`

```
git clone <url>
```

Un proceso de clonado realiza varias operaciones

- Crea un directorio con el nombre del repositorio
- Inicializa un repositorio local de Git
- Añade un repositorio remoto con el nombre `origin` y la URL del repositorio remoto
- Descarga los ficheros del repositorio remoto en su rama main al directorio local
- Añade un puntero a la rama main del repositorio remoto con el nombre `origin/main`
- Crea una rama local `main` que apunta al mismo commit que `origin/main`
- Crea una working area con los ficheros del proyecto
- Crea un puntero HEAD que apunta a la rama local `main`

Es posible indicar al clonado que se realice en un directorio diferente al del repositorio remoto

```
git clone <url> <nombre_directorio>
```

Respecto a las ramas, el clonado crea una rama local `main` que apunta al mismo commit que `origin/main`. Si el repositorio remoto tiene más ramas, se pueden añadir al repositorio local con el comando `git fetch`

```
git fetch
```

Estas ramas se añaden al repositorio local con el nombre `origin/nombre_rama`.

Las ramas remotas o **tracking branch** son ramas que siguen a una rama remota. Se crean automáticamente al clonar un repositorio remoto o al añadir un repositorio remoto a un repositorio local.

Se pueden ver las ramas remotas con el comando `git branch -r`

```

```

```
git branch -r
```

Las ramas remotas funcionan como marcadores, para recordarte en qué estado se encontraban tus repositorios remotos la última vez que conectaste con ellos. Se mueven automáticamente cuando estableces comunicaciones en la red y no pueden ser movidas de otra manera.

No puedes trabajar directamente en una rama remota (no se pueden realizar merge o añadir commits), pero puedes hacerlo en una rama local que siga a una rama remota.

Para trabajar con ellas, se puede crear una rama local que apunte a la rama remota con el comando `git checkout`

```
git checkout -b nombre_rama origin/nombre_rama.
```

Git remote

Git utiliza remote y ramas remotas (tracking branch) como referencias que facilitan la conexión con otros repositorios (conocidos como remotos)

Si el repositorio ha sido **clonado** con `git clone`, se habrá creado automáticamente una referencia al repositorio remoto con el nombre `origin`.

Si se ha creado un repositorio local con `git init`, se puede añadir una referencia al repositorio remoto con el comando `git remote add`

```
git remote add origin <url>
```

Operaciones con git remote

- `git remote`: Lista los repositorios remotos
- `git remote -v`: Lista los repositorios remotos con la URL
- `git remote add <nombre> <url>`: Añade un repositorio remoto
- `git remote remove <nombre>`: Elimina un repositorio remoto
- `git remote show <nombre>`: Muestra información sobre un repositorio remoto
- `git remote rename <nombre> <nuevo_nombre>`: Cambia el nombre de un repositorio remoto
- `git fetch <nombre>`: Descarga los cambios del repositorio remoto

Operaciones con repositorios remotos

Para mantener sincronizados los repositorios local y remoto, se utilizan los comandos `git push` y `git pull` que permiten subir y bajar cambios entre ambos repositorios.

Git push

- `git push`: Sube los cambios locales al repositorio remoto. Si aún no existe la rama remota, da un mensaje de error que indica el comando necesario para poder crear la rama en el remoto: `git push -u origin <rama>`
    - `git push -u origin main`: Sube los cambios locales al repositorio remoto y establece la rama remota como rama de seguimiento
    - `git push origin :rama`: Elimina la rama remota
    - `git push origin --tags`: Sube todos los tags al repositorio remoto (no se suben por defecto)

Git push no puede subir cambios si hay cambios en el repositorio remoto que no están en el local. En este caso, se debe hacer un `git pull` para descargar los cambios del remoto al local y fusionarlos con los locales.

Sin embargo, si se quiere subir los cambios locales sin fusionar con los remotos y sobre escribiéndolos, se puede utilizar el modificador `--force` o `-f`

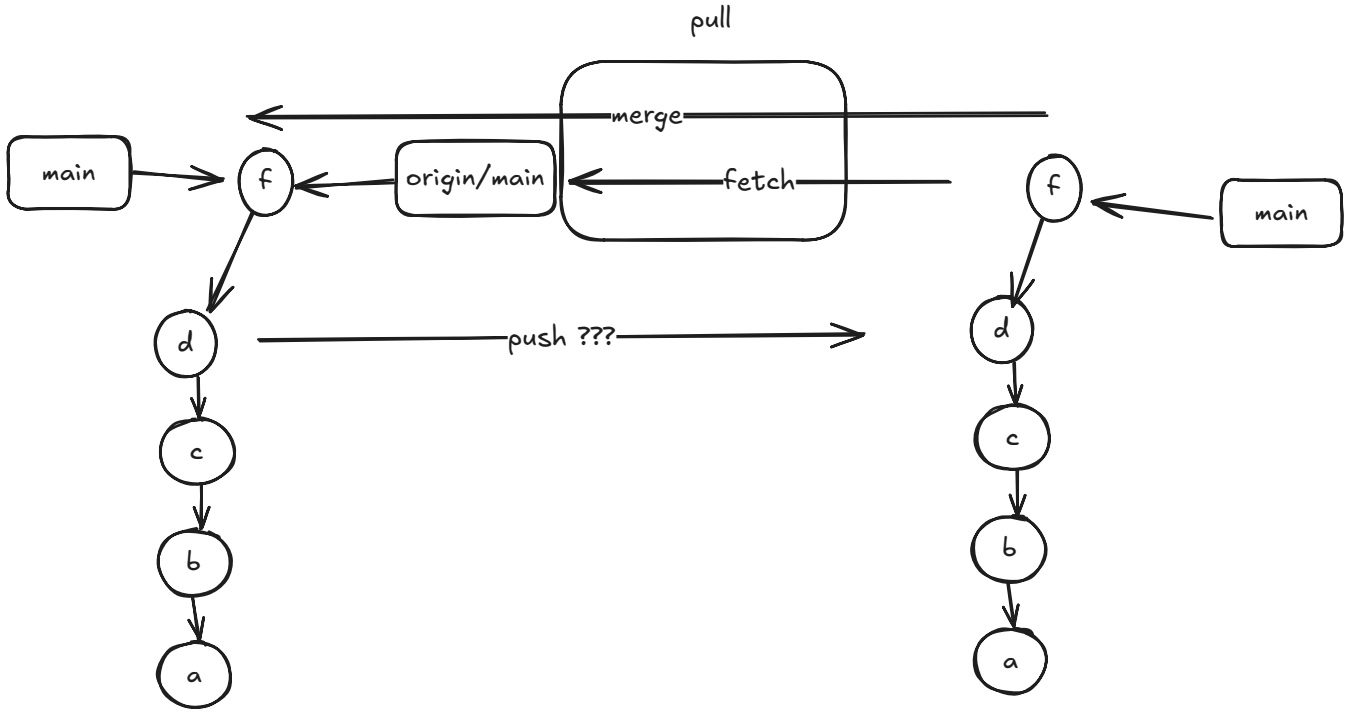
```
git push -f
```

Esto podría ser necesario si hemos **reescrito la historia localmente** y queremos subir los cambios al remoto. Es una operación peligrosa, ya que se pierden los cambios del remoto. Y en todo caso debe estar concertada con el resto del equipo.

Git pull

- `git pull`: Descarga los cambios del repositorio remoto al repositorio local. Es equivalente a `git fetch` seguido de `git merge`
    - `git fetch` descarga una determinada rama del repositorio remoto y la almacena en una rama remota, con su etiqueta `origin/rama`. No produce ningún cambio en las ramas locales
    - `git merge` fusiona la rama remota con la rama local, utilizando la estrategia ff (fast forward) si es posible

Con frecuencia el intento de hacer un push recibe el aviso de que hay cambios en el remoto que no están en el local, lo que impide hacer el push. En este caso, se debe hacer un pull para descargar los cambios del remoto al local y fusionarlos con los locales.



Comandos relacionados con estas operaciones son

- `git pull --rebase`: Descarga los cambios del repositorio remoto al repositorio local y los fusiona con los locales, utilizando la estrategia rebase
- `git pull --ff-only`: Descarga los cambios del repositorio remoto al repositorio local y los fusiona con los locales, utilizando la estrategia ff (fast forward) si es posible
- `git pull --no-ff`: Descarga los cambios del repositorio remoto al repositorio local y los fusiona con los locales, creando un commit de merge aunque sea posible hacer un ff (fast forward)
- `git checkout --track -b <local-branch> <remote-repo> / <remote-branch>`: Recupera una rama remota en una rama local y cambia a dicha rama
- `git branch <local-branch> <repo>/<remote-branch>`: Recupera una rama remota en una rama local

Pull Request

Un **Pull Request** es una petición que se hace a los colaboradores de un proyecto para que revisen y acepten los cambios que se han hecho en una rama y se fusionen con la rama principal.

No es una operación de Git, sino una funcionalidad de los servidores de repositorios Git, como GitHub, GitLab o Bitbucket.

Por tanto se crea en el servidor, no en el repositorio local. Se puede hacer desde la web del servidor o desde la línea de comandos, con el comando `git request-pull`

```
git request-pull <inicio> <final> <repositorio>
```

- Si la nueva rama se crea localmente, el primer paso es subirla al repositorio remoto con `git push`

```
git checkout -b <rama>
echo "Nuevo contenido" > README.md
git add .
git commit -m "Mensaje"
git push --set-upstream origin feature
# equivale a git push -u origin <rama>
```

Al acceder al repositorio en GitHub, se puede ver la nueva rama y generalmente aparecerá un aviso "feature had recent pushes less than a minute ago" y un botón "Compare & pull request"

El interface de GitHub muestra la información de la PR: base: la rama a la que se quiere fusionar (main) compare: la rama que se quiere fusionar (feature)

Además, permite añadir un título y una descripción al Pull Request, y seleccionar los revisores y diversas etiquetas (Labels, Projects, Milestone...) relacionadas con la gestión del proyecto.

Una vez creada, la PR realiza las comprobaciones que tenga definidas el proyecto, como tests, análisis de código, etc. y notifica a los revisores.

En una PR abierta tanto el autor como los revisores pueden realizar la siguientes operaciones

- Permite a los revisores revisar los cambios y hacer comentarios
- Permite a los revisores aprobar o rechazar los cambios
- Permite a los revisores fusionar los cambios
- Permite a los revisores cerrar la PR
- Permite al autor de la PR cerrar la PR
- Permite al autor de la PR borrar la rama
- Permite al autor de la PR reabrir la PR

El autor puede cambiar el estado de la PR

- draft, para indicar que no está lista para ser merge, aunque la política del equipo quizás defina que puede ir siendo revisada,
- ready, para indicar que está lista.

Cualquiera con acceso a la rama puede añadir nuevos commits, que se incorporaran automáticamente a la PR, desencadenando de nuevo los procesos de revisión y notificación.

Una PR abierta puede verse como un mecanismo de colaboración en el equipo, dando lugar a discusiones, mejoras, etc. más que como un simple mecanismo de aprobación.

En cualquier caso su uso depende mucho de la dinámica de trabajo de cada equipo. Es frecuente que se defina la necesidad de un mínimo de aprobaciones para poder fusionar una PR, que dependerá entre otras coss del tamaño del equipo. Se puede definir que el responsable de fusionar sea el autor, una vez recibidas las aprobaciones, el último de los revisores etc.

En cualquier caso el proceso de fusión de una PR es un proceso de merge / rebase, que puede ser automático o manual, dependiendo de la configuración del proyecto. En el segundo caso,basta usar el botón que proporciona el interfaz y elegir la estrategia de fusión, ff, rebase, squash, etc. de entre las que se hayan permitido en la configuración de la rama principal.

El cierre de la PR puede ser consecuencia de su fusión o del abandono de esta línea de trabajo, y si es necesario se puede reabrir.

Configuración de las ramas y PR

Cuando se trabaja con PR es habitual realizar una serie de ajustes en la configuración de la s ramas, especialmente en la rama principal, para facilitar el trabajo con PR.

- Proteger la rama principal para evitar cambios directos
- Configurar la rama principal como rama por defecto de las PR
- Configurar las opciones de fusión permitidas para las PR y la opción por defecto
- Configurar las opciones de revisión de las PR
- Configurar las opciones de notificación de las PR
- Configurar las opciones de eliminación de las ramas de las PR
- Configurar las opciones de eliminación de las PR

Todos estos ajustes se realizan en la configuración del repositorio en el servidor, no en el repositorio local, accediendo a los settings -> branches.

- Se crea un rulset
- Se aplica a la rama default
- Se selecciona Require a pull request before merging
- Se selecciona Require status checks to pass

Otra opción es utilizar "Add classic branch protection rule" que también permite configurar las opciones de protección de la rama principal de forma detallada.

Actualizaciones de las ramas feature

Con el tiempo, la rama principal puede avanzar y la rama feature puede quedarse desactualizada. Para mantenerla actualizada, se puede hacer un merge de la rama feature sobre la rama principal, después de haber actualizado esta.

```
git checkout main
git pull
git checkout feature
git merge main
```

Los posibles conflictos se resuelven como en cualquier merge, en el entrono local y una vez resueltos, se suben los cambios al repositorio remoto

```
git push
```

Tags

Los tags son referencias a un commit específico. Se utilizan para marcar versiones, releases, etc. Son otra forma de referenciar un commit, como una rama pero hay una diferencia entre ambas:

- las ramas son dinámicas, cambian con cada commit que se hace en ellas. Su objetivo es seguir el desarrollo del proyecto.
- los tags son estáticos, no cambian ni se mueven con el tiempo. Una vez aplicado, se debe dejar tal cual.

Existen dos tipos de tags:

- Ligeras: Se crean sin modificador y no tienen mensaje. Son simplemente un puntero a un commit, igual que una rama.

Para crear un tag se utiliza el comando `git tag`

```
git tag v1.0 <referencia al commit>
```

- Anotados: Se crean con el modificador -a y se les puede añadir un mensaje. Se almacenan como objetos completos en la base de datos de Git. Son pos tanto identificadas por su hash (checksum); contienen el nombre, email y fecha del tagger. Se pueden firmar y verificar. Son el formato recomendado Are stored as full objects in the Git database. They are checksummed; contain the tagger name, email and date. Can be signed and verified.

Para crear un tag se utiliza el comando `git tag` con el modificador -a

```
git tag -a v1.0 -m "Primera versión"
```

Operaciones con tags

Para listar los tags se utiliza el comando `git tag`

```
git tag
```

Pare mostrar un tag creado se utiliza el comando `git show`

```
git show v1.0
```

Para subir un tag al repositorio remoto se utiliza el comando `git push`

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

También es posible subir todos los tags al repositorio remoto:

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

Para usar un tag, se puede hacer checkout a un tag

```
git checkout v1.0
```

Patches

Los patches son ficheros que contienen los cambios entre dos commits. Se pueden generar con el comando `git format-patch`

```
git format-patch HEAD~2..HEAD
```

Para aplicar un patch se utiliza el comando `git apply`

```
git apply 0001-Add-new-feature.patch
```

Workflows

GitFlow

El GitFlow es un modelo de ramificación que se basa en dos ramas principales:

- master: rama principal, estable, que contiene el código en producción
- develop: rama de desarrollo, inestable, que contiene el código en desarrollo

Además, se utilizan otras ramas auxiliares:

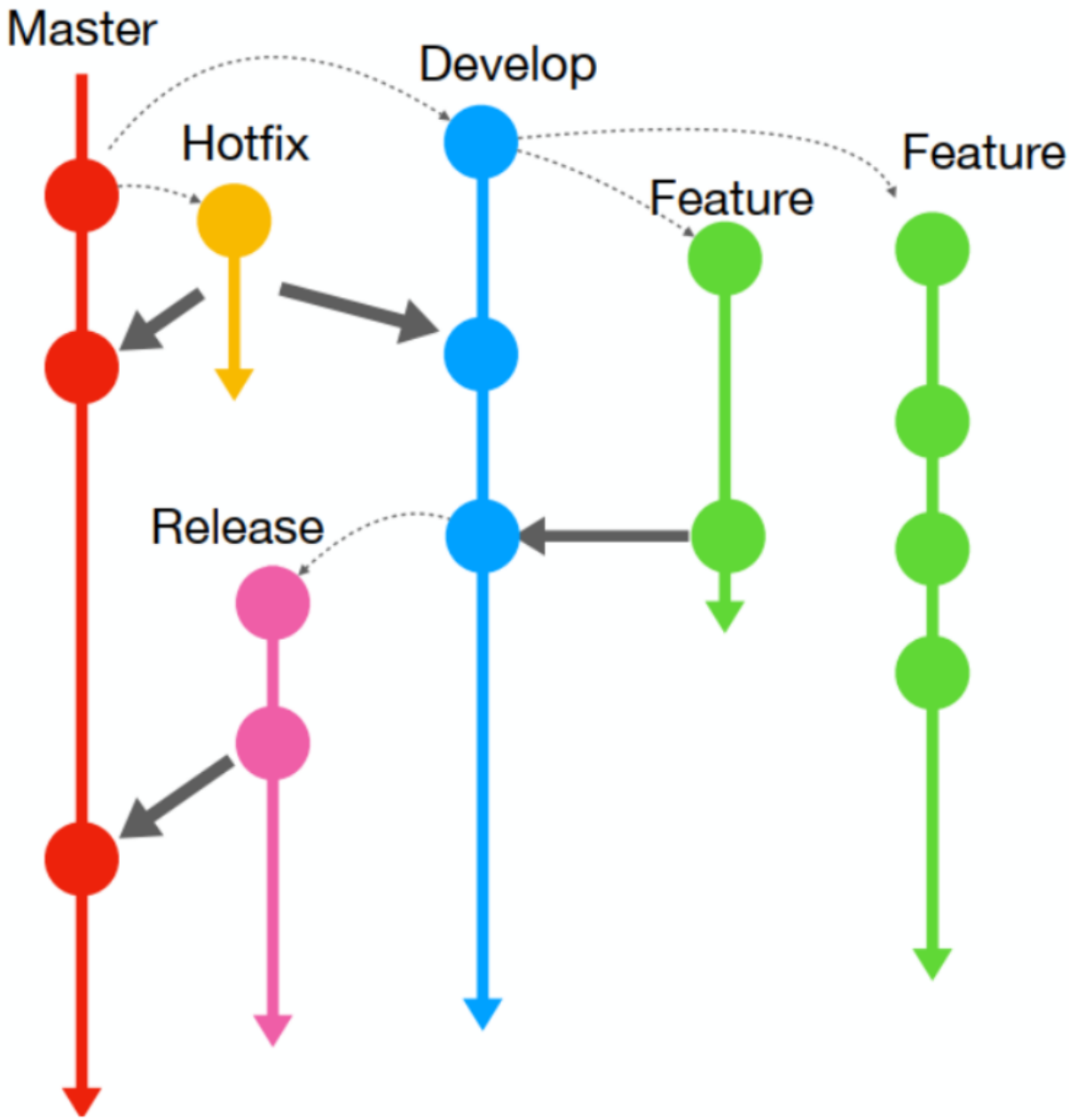
- feature: rama de desarrollo de una nueva funcionalidad
- release: rama de preparación de una nueva versión
- hotfix: rama de corrección de errores en producción

El flujo de trabajo es el siguiente:

1. Se crea una rama feature a partir de develop
2. Se desarrolla la funcionalidad en la rama feature
3. Se fusiona la rama feature en develop
4. Se crea una rama release a partir de develop
5. Se prepara la nueva versión en la rama release
6. Se fusiona la rama release en master
7. Se fusiona la rama release en develop

En caso de que aparezca la necesidad de corregir un error en producción,

1. se crea una rama hotfix a partir de master
2. se corrige el error
3. se fusiona la rama hotfix en master y develop.



Entre las ventajas del GitFlow se encuentran:

- Facilita la colaboración en equipos grandes
- Hay mucha documentación y herramientas que lo soportan
- Es un modelo muy extendido
- Las ramas están muy bien definidas y organizadas

Entre las desventajas del GitFlow se encuentran:

- Es un modelo muy complejo, especialmente para equipos pequeños
- Puede ser difícil de entender y de implementar

- Si hay un rollback, puede ser complicado de gestionar y se pierde todo el valor entregado
- Las ramas feature pueden ser muy largas y difíciles de gestionar
- Puede haber problemas de integración si no se hace un merge frecuente
- No esta muy bien adaptado a la filosofía de entrega continua (CI/CD)

Refetencias

- Git Flow
- Git Flow Cheatsheet

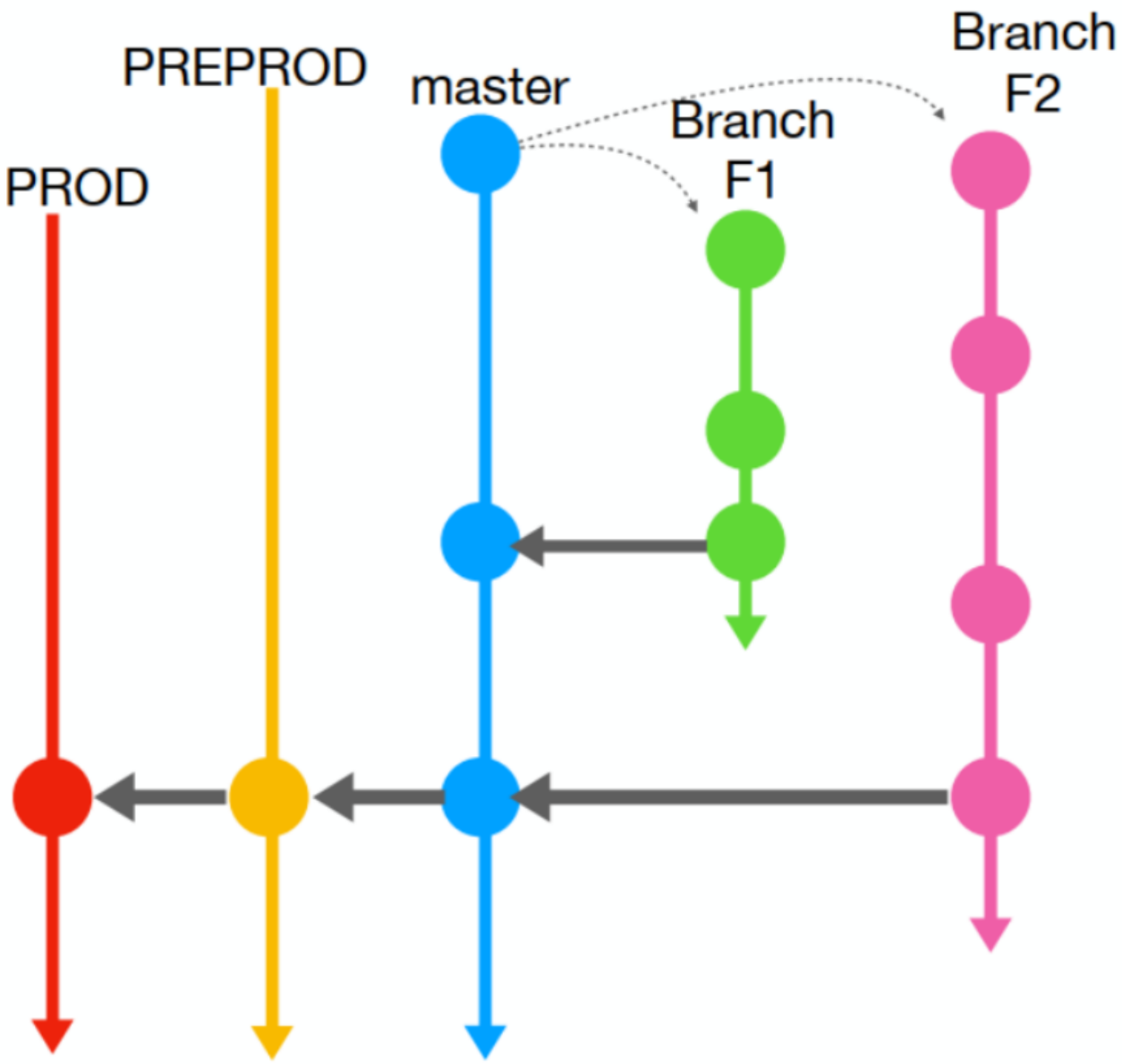
GitLab Flow (Environment Branching)

El GitLab Flow es un modelo de ramificación que se basa en las siguientes ramas:

- environment: ramas de entorno, que contienen el código en producción, preproducción, etc.
- master: rama principal, estable, que contiene el código en desarrollo
- feature: ramas de desarrollo de una nueva funcionalidad
- hotfix: ramas de corrección de errores en producción

El flujo de trabajo es el siguiente:

1. Se crea una rama feature a partir de master
2. Se desarrolla la funcionalidad en la rama feature
3. Se fusiona la rama feature en master
4. Se crea una rama environment a partir de master
5. Se despliega la rama environment en un entorno de pruebas
6. Se prueba la funcionalidad en el entorno de pruebas
7. Se fusiona la rama environment en producción
8. Se despliega la rama environment en producción



Entre las ventajas del GitLab Flow se encuentran:

- Facilita la colaboración en equipos grandes
- Master nunca esta roto, siempre está en producción
- Los environment branches permiten tener entorno para probar distintas configuraciones

Entre las desventajas del GitLab Flow se encuentran:

- Es un modelo muy complejo, especialmente para equipos pequeños
- Las ramas feature pueden ser muy largas y difíciles de gestionar
- Si hay un rollback, puede ser complicado de gestionar y se pierde todo el valor entregado
- No esta muy bien adaptado a la filosofía de entrega continua (CI/CD)

Referencias

- GitLab Flow- - Environment and other Branching Strategies

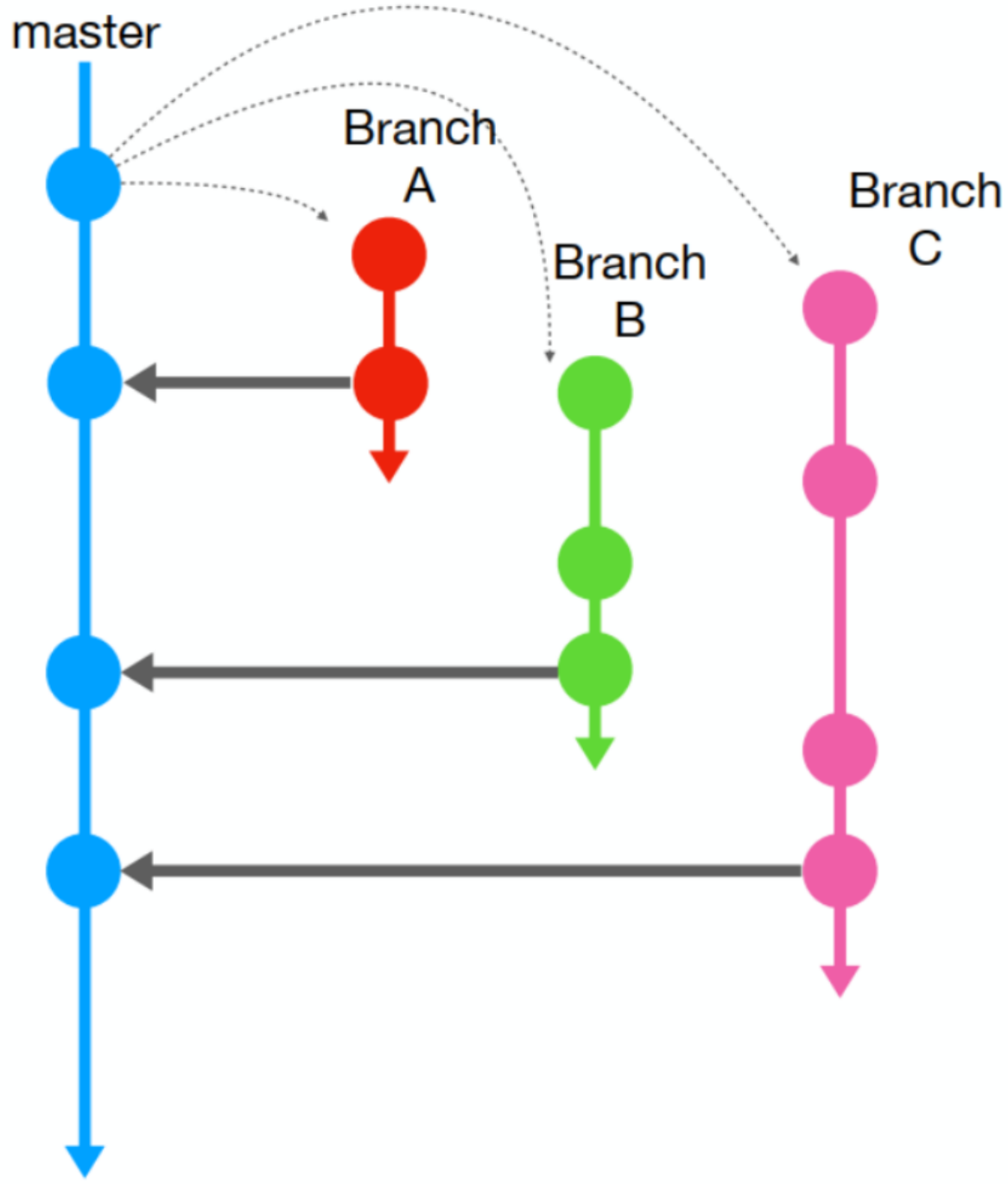
GitHub Flow, Feature Branching, Trunk Based Development

El GitHub Flow, con distintas variantes conocidas como Feature Branching o Trunk Based Development, es un modelo de ramificación que se basa en las siguientes ramas:

- master: rama principal, estable, que contiene el código en producción
- feature: ramas cortas de de desarrollo de una nueva funcionalidad

El flujo de trabajo es el siguiente:

1. Se crea una rama feature a partir de master
2. Se desarrolla la funcionalidad en la rama feature
3. Se fusiona la rama feature en master
4. Se despliega la rama master en producción
5. En algunas variantes del modelo, se aplican tags a determinados commits de master para marcar las versiones
6. Se repite el proceso con cada nueva rama feature



Entre las ventajas del GitHub Flow se encuentran:

- Es un modelo muy sencillo, fácil de entender y de implementar
- Facilita la colaboración en equipos pequeños
- Master nunca esta roto, siempre está en producción
- Las ramas feature son cortas y fáciles de gestionar
- Se adapta muy bien a la filosofía de entrega continua (CI/CD)

Entre las desventajas del GitHub Flow se encuentran:

- Puede no ser adecuado para equipos grandes
- Es muy importante que las ramas de feature sean cortas
- Al aplicarse la filosofía de CI/CD y desplegar master en producción, la integración de estar bien automatizada y los tests se vuelven críticos

Referencias

- GitHub Flow
- Understanding the GitHub Flow
- Trunk based

Ship-show-ask

Es un modelo de ramificación propuesto más recientemente que se basa en tres procedimientos diferentes

- Ship: se desarrolla el código directamente en master y por tanto se envía inmediatamente a producción
- Show: se desarrolla el código en una rama feature y se muestra a los interesados antes de fusionarla en master, pero sin someterla a un proceso de code review y aprobación
- Ask: se desarrolla el código en una rama feature y se somete a un proceso de code review y aprobación antes de fusionarla en master, como se hace en el GitHub Flow

El desarrollador cobra un mayor protagonismo al decidir cual de las tres estrategias aplicar en cada caso, en función de la criticidad del cambio, la complejidad del código, la urgencia del despliegue, etc.

Referencias

- ship-show-ask

CONFIGURACIÓN DE GIT. Hooks

Configuración. gitconfig

- .alias
- Editor
- Coloreado comandos
- Formato salida comandos
- Otras opciones

Hooks

Los hooks son scripts que se ejecutan automáticamente en determinados momentos del ciclo de vida de un repositorio Git. Permiten automatizar tareas, como la validación de código, la ejecución de tests, el envío de notificaciones, etc.

Podemos encontrar información sobre su funcionamiento

- en la documentación oficial de Git: https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks
- el el libro Pro Git: https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks

Al inicializar un nuevo repositorio con `git init`, Git llena el directorio de hooks `.git/hooks` con varios scripts de ejemplo, muchos de los cuales son útiles por sí mismos; además, documentan los valores de entrada de cada script. Todos los ejemplos están escritos como `scripts de shell`, con algo de `Perl`, pero cualquier script ejecutable con un nombre correcto funcionará correctamente; (`Ruby`, `Python`...). Si quieres usar los scripts de hook incluidos, tendrás que renombrarlos; todos sus nombres de archivo terminan en .sample.

Los hooks que se encuentran inicialmente en el directorio `.git/hooks` del repositorio son los siguientes:

- prepare-commit-msg.sample -> Preparar el mensaje de commit
- commit-msg.sample -> Validar el mensaje de commit
- pre-commit.sample -> Validar los cambios antes de hacer un commit
- push-to-checkout.sample -> Validar los cambios antes de hacer un checkout
- pre-merge-commit.sample -> Validar los cambios antes de hacer un merge
- pre-rebase.sample -> Validar los cambios antes de hacer un rebase
- pre-push.sample -> Validar los cambios antes de hacer un push
- pre-receive.sample -> Validar los cambios antes de recibir un push
- update.sample -> Validar los cambios antes de hacer un update
- post-update.sample -> Notificar a los usuarios sobre actualizaciones
- applypatch-msg.sample -> Validar los mensajes de los parches aplicados
- pre-applypatch.sample -> Validar los parches antes de aplicarlos
- fsmonitor-watchman.sample -> Integración con Watchman para mejorar el rendimiento de git status
- sendemail-validate.sample -> Validar los correos electrónicos enviados

Estos hooks se pueden clasificar en dos tipos:

- hooks de lado cliente: commits, emails, rebase, ...
- hooks de lado servidor: prereceive, postreceive, update

Los hooks de lado cliente se ejecutan en el equipo del desarrollador y permiten validar los cambios antes de hacer un commit, un push, un rebase, etc. Los hooks de lado servidor se ejecutan en el servidor y permiten validar los cambios antes de recibir un push, notificar a los usuarios sobre actualizaciones, etc.

Husky

Husky es una herramienta que facilita la gestión de hooks en proyectos de JavaScript que utilizan Git.

SUB-PROYECTOS

Existen varias formas de incluir un subproyecto en un proyecto Git

- Submodules
- Subtree
- Subrepo (opción extra, no incluida en Git)

Los submodules son la opción más utilizada y la que mejor se integra con Git. Permiten incluir un repositorio Git dentro de otro repositorio Git, manteniendo ambos repositorios independientes. Los subtree permiten incluir un repositorio Git dentro de otro repositorio Git, pero integrando ambos repositorios en uno solo. No mantienen la independencia entre ambos repositorios.

Submodules

Los submodules son una característica de Git que permite incluir un repositorio Git dentro de otro repositorio Git, manteniendo ambos repositorios independientes. Los submodules son útiles cuando se quiere incluir una librería, un framework, un plugin, etc. en un proyecto, pero se quiere mantener la independencia entre ambos proyectos.

Creación de un submodule

El primer paso es añadir un repositorio ya existente como submodule al repositorio principal con el comando `git submodule add`

```
git submodule add \<url\> [<path>]
```

La url es la dirección del repositorio que se quiere añadir como submodule. El path es el directorio donde se quiere clonar el submodule. Si no se indica, se clona en un directorio con el mismo nombre que el repositorio. Un valor habitual para librerías externas es `vendor/nombre_libreria`

El resultado es el equivalente a clonar el repositorio en el directorio indicado y añadir un fichero `.gitmodules` en el que se almacena los metadata con la información del submodule, como la url y el path.

Si despues de añadir el submodule se hace un `git status`, se verá que hay dos cambios pendientes de commit

- el fichero `.gitmodules`
- el directorio del submodule, que se muestra como un nuevo fichero, con el nombre del submodule y el hash del commit al que apunta

```
git status
On branch main
```

```
new file:   .gitmodules
new file:   nombre_submodule (new commits)
```

Si hacemos git add y git diff, veremos que no muestra los cambios en el submodule como una carpeta, sino como el hash del commit al que apunta el submodule.

```
git add .
git diff --cached
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..e69de29
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+ [submodule "nombre_submodule"]
+   path = nombre_submodule
+   url = <url>
diff --git a/nombre_submodule b/nombre_submodule
new file mode 160000
index 0000000..4b825dc
--- /dev/null
+++ b/nombre_submodule
@@ -0,0 +1 @@
+Subproject commit 4b825dc642cb6eb9a060e54bf8d69288fbee4904
```

Lo siguiente paso es hacer un commit para guardar los cambios en el repositorio principal.

```
git commit -m "Añadido submodule nombre_submodule"
```

Clonado de un repositorio con submodules: inicialización

Cuando se clona un repositorio que contiene submodules, los submodules no se clonan automáticamente. Se crea la estructura de directorios, pero no se descargan los ficheros del submodule. Es necesario inicializarlos y actualizarlos con los comandos `git submodule init` (para registrar el submodule) y `git submodule update` (para descargar los ficheros del submodule)

```
git submodule init
git submodule update
```

Si se quiere clonar el repositorio y los submodules en un solo paso, se puede utilizar el modificador `--recurse-submodules` del comando `git clone`

```
git clone --recurse-submodules <url>
```

Actualizaciones de un submodule

Los submodules son independientes del repositorio principal, por lo que se pueden actualizar de forma independiente. Para actualizar un submodule, existen dos mecanismos.

- Primera opción

se debe entrar en el directorio del submodule y hacer un `git pull` para descargar los cambios del repositorio remoto del submodule.

```
cd nombre_submodule
git pull
```

Como se habrá actualizado el submodule apuntando al nuevo commit final de repo al que apunta , el repositorio principal detectará que hay cambios pendientes de commit en el submodule. Se debe hacer un `git add` y un `git commit` para guardar los cambios en el repositorio principal.

```
cd ..
git add nombre_submodule
git commit -m "Actualizado submodule nombre_submodule"
```

- Segunda opción

Se puede actualizar el submodule desde el repositorio principal, sin entrar en el directorio del submodule, con el comando `git submodule update --remote`

```
git submodule update --remote --recursive
```

De esta forma se actualizarían todos los submódulos que haya en el repositorio, incluidos los submódulos de los submódulos (si los hubiera).

Como en el caso anterior, se debe hacer un `git add` y un `git commit` para guardar los cambios en el repositorio principal.

```
git add .
git commit -m "Actualizado submodule nombre_submodule"
```

Cuando otros usuarios han actualizado los submodulos y lo han reflejado en el repo compartido, al hacer un `git pull` en el repositorio principal, los submodules no se actualizan automáticamente. Se debe hacer un `git pull --recurse-submodules` para actualizar los submodules a la versión que indica el repositorio principal.

BUENAS PRÁCTICAS

- Commits atómicos
- Commits frecuentes
- No commits de trabajo a medias
- Test antes de commit
- Buenos mensajes de commit
- Usar branches, feature-branching
- Fijar un workflow común

Apéndice. UTILIDADES. INTEGRACIÓN CON OTRAS HERRAMIENTAS Y ENTORNOS

- GitK, GitG y git gui | git log graph | formato git log
- IntelliJ
- SourceTree
- Github
- GitLab
- Bitbucket