GUÍA PARA PRINCIPIANTES DE GIT

¿Por qué usar Git?

El propósito de trabajar con Git es tener un control de las diferentes versiones que vamos teniendo de nuestro código y poder compartirlo con otros. Además, Git es el estándar de facto del sistema de control de versiones. Es decir, aunque existen otros sistemas de control de versiones, Git es ampliamente utilizado por la mayoría de programadores.

En esta guía encontrarás, qué es Git y cómo funciona, los conceptos básicos de Git, cómo dar los primeros pasos en Git, el resumen de los diferentes objetos de Git y un listado con los diferentes comandos de Git con su sintaxis y la explicación de sus funciones.
Espero que te sea de ayuda. Para cualquier cosa, puedes ponerte en contacto con nosotros mediante el siguiente email: transparentia2025@gmail.com

INTRODUCCIÓN A GIT

¿Qué es Git?

Supongo que ya tienes idea de lo que es Git y de para qué se utiliza, si no, no estarías buscando una guía, pero te explico. Git es un sistema de control de versiones distribuido diseñado para rastrear cambios en archivos (versiones), principalmente código fuente de software. A diferencia de los sistemas centralizados, Git permite a cada desarrollador tener una copia completa del repositorio, lo que facilita la colaboración y el trabajo offline.
Git es uno de los tres software diseñados por Linus Torvalds, el creador de Linux. Lo usó para desarrollar el Kernel de Linux.

¿Cómo funciona Git a un nivel más profundo?

Git almacena los cambios como diferencias entre versiones, lo que lo hace muy eficiente en términos de almacenamiento. Cada vez que haces un cambio en tu proyecto y le dices a Git que lo guarde (esto se llama hacer un "commit"), Git crea un nuevo objeto que contiene las diferencias entre las diferentes versiones de archivo. En lugar de guardar una copia completa de todos tus archivos cada vez que haces un cambio, solo guarda las diferencias sobre el último archivo.
Esto hace que Git tenga las siguientes **características**:

- **Velocidad**: Las operaciones locales son extremadamente rápidas debido a su diseño distribuido.
- **Flexibilidad**: Permite crear ramas y fusionarlas fácilmente, lo que facilita la experimentación y el desarrollo en paralelo.
- **Integridad:** La estructura de datos de Git garantiza la integridad de los datos y hace que sea muy difícil corromper el historial.
- **Descentralización:** No depende de un servidor central, lo que lo hace más robusto y confiable frente a posibles caídas del servidor central.

Conceptos básicos de Git

Hay una serie de conceptos que debes conocer antes de comenzar a trabajar con Git. Son los siguientes, y en paralelo, quiero recomendarte que te pases por learngitbranching.js.org donde podrás ver, de forma muy clara, la estructura y el uso de los diferentes comandos de Git:

Repositorio:
Es una carpeta especial que contiene todos los archivos de un proyecto y un registro detallado de todos los cambios realizados en esos archivos a lo largo del tiempo.
Este registro se almacena en una base de datos interna de Git y se denomina "historial".

Commit:
Lo podríamos traducir como “Compromiso”.
Es una instantánea de tu proyecto en un momento específico.
Cada vez que haces un commit, estás guardando un estado concreto de todos los archivos que están siendo rastreados por Git.
A cada commit se le asigna un identificador único (un hash) que lo distingue de los demás.
El principal comando para hacer un commit es `git commit -m “mensaje explicativo”`.

Área de Staging o índice:
Es un área temporal donde se añaden los cambios que deseas incluir en el próximo commit.
Antes de hacer un commit, debes agregar los archivos modificados al área de staging. Para hacerlo, utilizarás `git add`.

Branch:
Una rama o branch es una línea de desarrollo independiente.
Imagina que a partir de cierto punto en tu código, quieres realizar algún cambio para mejorarlo, solucionar un problema,... Entonces, puedes crear una o más ramas para trabajar en el código sin afectar la rama principal.
Cada rama tiene su propio conjunto de commits y apunta a un commit específico. Al último commit de la branch en la que trabajas suele apuntar el HEAD.
El principal comando para crear ramas es `git branch nombre_rama` .

HEAD:
HEAD es un puntero que siempre señala al commit en el que te encuentras actualmente. HEAD está estrechamente relacionado con las ramas. Cuando creas una nueva rama o cambias de rama, HEAD se mueve para apuntar al último commit de esa rama. Ya no apunta a una rama específica, sino a un commit concreto.
En este caso, estarás en un estado llamado "detached HEAD". Esto suele ocurrir cuando realizas un `git checkout` a un commit específico en lugar de a una rama.

Implicaciones del "detached HEAD":

1. **Estado flotante:** Estás en un estado "flotante" en el historial de commits. Esto significa que cualquier cambio que realices no se asociará directamente a una rama existente.
2. **Pérdida de contexto:** Al no estar en una rama, pierdes el contexto de dónde proviene ese commit y qué cambios se han realizado desde entonces.
3. **Dificultad para compartir cambios:** Si intentas hacer push desde este estado, es probable que te encuentres con errores, ya que no hay una rama de destino clara.
4. **Riesgo de perder cambios:** Si no creas una nueva rama desde este estado y realizas más commits, podrías perder esos cambios si vuelves a una rama existente.

¿Cómo solucionar este problema?

Crear una nueva rama:
`git checkout -b nueva_rama`

Esto creará una nueva rama a partir del commit en el que te encuentras y el HEAD apuntará a esta nueva rama.

Volver a una rama existente:
`git checkout main`

Si deseas volver a la rama principal, simplemente realiza un `checkout` a esa rama.

Merge:
La fusión o merge es el proceso de combinar los cambios de una rama en otra.
Cuando fusionas una rama, estás tomando los cambios de esa rama y los estás integrando en otra rama.
El comando es `git merge *nombre_rama_a_agregar*`

Origen (Origin):
Es el nombre por defecto que se le da al repositorio remoto cuando se clona un proyecto.
Un repositorio remoto es una copia del repositorio local que se almacena en un servidor.

Pull:
Es la acción de obtener los cambios más recientes de un repositorio remoto y actualizar tu repositorio local.
Utilizamos git pull *origin rama_a_descargar*.

Push:
Es la acción de enviar los cambios de tu repositorio local a un repositorio remoto.
El comando es git push *origin nombre_rama_a_subir*.

¿Cómo es el flujo de trabajo en Git?

El concepto que os voy a explicar ahora es muy importante, así que espero explicároslo correctamente. Entre nuestra computadora y el repositorio, existe una etapa intermedia que llamamos Staging area. A esta zona intermedia, podemos añadir (git add) todos los archivos que consideremos, pero podría ser que no quisiéramos subirlos todos al repositorio externo (GitHub, Radicle, GitLab, Bitbucket,...), así que una vez añadidos, para subirlos al repositorio tenemos que "comprometerlos", es decir, hacer un git commit. Más adelante, te explico cómo hacerlo.

INSTALACIÓN Y PRIMEROS PASOS CON GIT

1. **Instalación.**
   Instala Git desde https://git-scm.com/downloads
   Al instalarlo, te pedirá que escojas un editor de código y un terminal (te aconsejo Git Bash).

2. Ejecuta `git --version` si la instalación fue exitosa, verás la versión de Git instalada.

3. **Configuración.**
   Ejecuta:
   `git config --global user.name "Tu Nombre"`
   `git config --global user.email "tu.correo@ejemplo.com"`
   `git config --list` : si quieres verificar tu configuración.

4. **Crea un nuevo directorio para tu proyecto:**
   - Abre tu terminal y navega hasta la ubicación donde deseas crear tu proyecto. ¿Cómo abrimos el terminal? Abre el buscador de tu ordenador y teclea git bash.
   - Ejecuta el siguiente comando para crear un nuevo directorio:
     `mkdir mi_proyecto`
   - Entra al directorio recién creado:
     `cd mi_proyecto`
5. **Inicializa un repositorio de Git dentro del directorio creado:**
   `git init`

6. Verifica el estado de tu repositorio después de cada ejecución de comandos:
   `git status`

7. Empieza a añadir archivos a la Área de Staging:
   `git add`

8. Sube tu primer commit. Recuerda especificar bien el mensaje. Si prefieres hacerlo dentro de tu editor de texto, escribe sólo `git commit`. Entonces, se abritá tu editor de texto y podrás configurar el mensaje cómo tú quieras.
   `git commit -m “cuál es el commit”`

9. **Conectando con un Repositorio Remoto (Opcional)**
    - Crea un repositorio en una plataforma de alojamiento como GitHub, Radicle, GitLab o Bitbucket. GitHub es el más utilizado y centraliza un montón de repositorios que puedes descargar. Te permite crear tu propio espacio donde compartir tu código. Fue comprado por Microsoft en 2018. Por otro lado, Radicle es una opción libre y descentralizada.
    - Agrega el repositorio remoto: Copia la URL del repositorio remoto que has creado. Ejecuta el siguiente comando, reemplazando `https://tu_url_remoto.git` con la URL real:
      git remote add origin https://tu_url_remoto.git
    - Envía tus cambios al repositorio remoto:

      git push -u origin main

PARA SABER MÁS

Objetos de Git

Git utiliza una serie de objetos internos para gestionar eficientemente el historial de versiones de tus proyectos. Estos objetos son fundamentales para entender cómo funciona Git a un nivel más profundo.

Los **principales objetos** de Git son:

Blob:

- Representa el contenido de un archivo.
- Es como un contenedor de datos binarios sin ninguna estructura adicional.
- Cada archivo único en tu proyecto se almacena como un blob.

Piensa en un blob como un capítulo dentro de un libro. Un blob almacena el contenido de un archivo individual: puede ser un archivo de texto, una imagen, un archivo de código, etc. Es la unidad más básica de almacenamiento en Git.

Tree:

- Representa la estructura de un directorio.
- Contiene referencias a otros árboles o blobs, indicando la relación entre los archivos y directorios.
- Un árbol define la estructura de un directorio en un momento específico.

Commit:

- Representa una instantánea completa de tu proyecto en un momento dado.
- Contiene:
    - Un mensaje que describe los cambios realizados.
    - Una referencia al árbol raíz que define la estructura del proyecto en ese momento.
    - Referencias a los commits padres (excepto para el commit inicial).
- Cada commit es como un marcador en el historial de tu proyecto. Un commit siempre apunta a un árbol, que representa la estructura del proyecto en ese momento.

Imagina el commit como el libro completo en nuestra biblioteca. Un commit captura un estado completo de tu proyecto en un momento específico.

Tag:

- Es una etiqueta que se puede asignar a un commit específico.
- Sirve para marcar puntos importantes en el historial, como releases o versiones estables. Un **release** o lanzamiento es una colección de commits que representa una versión estable y completa del proyecto. Sirve para **compartir el código** (cuando estás listo para que otros usen tu software, puedes crear un release. Esto hace que sea fácil para los usuarios descargar una versión específica y confiable), **documentar cambios** (cada release suele ir acompañado de un conjunto de notas que describen las nuevas características, correcciones de errores y otros cambios importantes), **gestionar versiones** y **automatizar procesos.**

  ¿Cómo se crean los **releases** en Git?

  1. **Crea una etiqueta:** Un release se basa en una etiqueta (tag) de Git. Una etiqueta es una referencia a un commit específico.
     La sintaxis básica de una etiqueta es la siguiente:
     `git tag [opciones] *nombre_etiqueta* *hash_del_commit*`

     Donde las opciones pueden ser:
     `-a`: Crea una etiqueta anotada.
     `-m mensaje`: Añade un mensaje a la etiqueta anotada.
     `-d nombre_etiqueta`: Elimina una etiqueta.
     `-l patrón`: Lista las etiquetas que coinciden con un patrón.
     `-v nombre_etiqueta`: Muestra información detallada sobre una etiqueta.

  2. **Agrega metadatos:** Al crear un release, puedes agregar información adicional como:
     - Un nombre descriptivo para el release (por ejemplo, "v1.0.0")
     - Notas de lanzamiento
     - Archivos asociados (binarios, documentación, etc.)

     Puedes agregar esos metadatos siguiendo el siguiente ejemplo:
     `git tag -a *nombre_etiqueta* -m "Mensaje descriptivo" *hash_del_commit*`

3. **Publica el release:** Una vez creado, el release se puede publicar en plataformas como GitHub, Radicle, GitLab o Bitbucket, donde estará disponible para que otros lo descarguen. Puedes hacerlo de dos maneras:
    - git push origin *nombre_etiqueta*
    - git push origin --tags (esta opción envía todas las etiquetas).

La relación entre estos objetos forma un gráfico acíclico dirigido (DAG), donde los commits son los nodos y las referencias entre commits son las aristas. Las aristas, en un DAG de Git, son como flechas que conectan los commits y muestran la relación de parentesco entre ellos. Son esenciales para entender la historia de un proyecto y cómo se han realizado los cambios a lo largo del tiempo.

A continuación, un ejemplo de DAG.

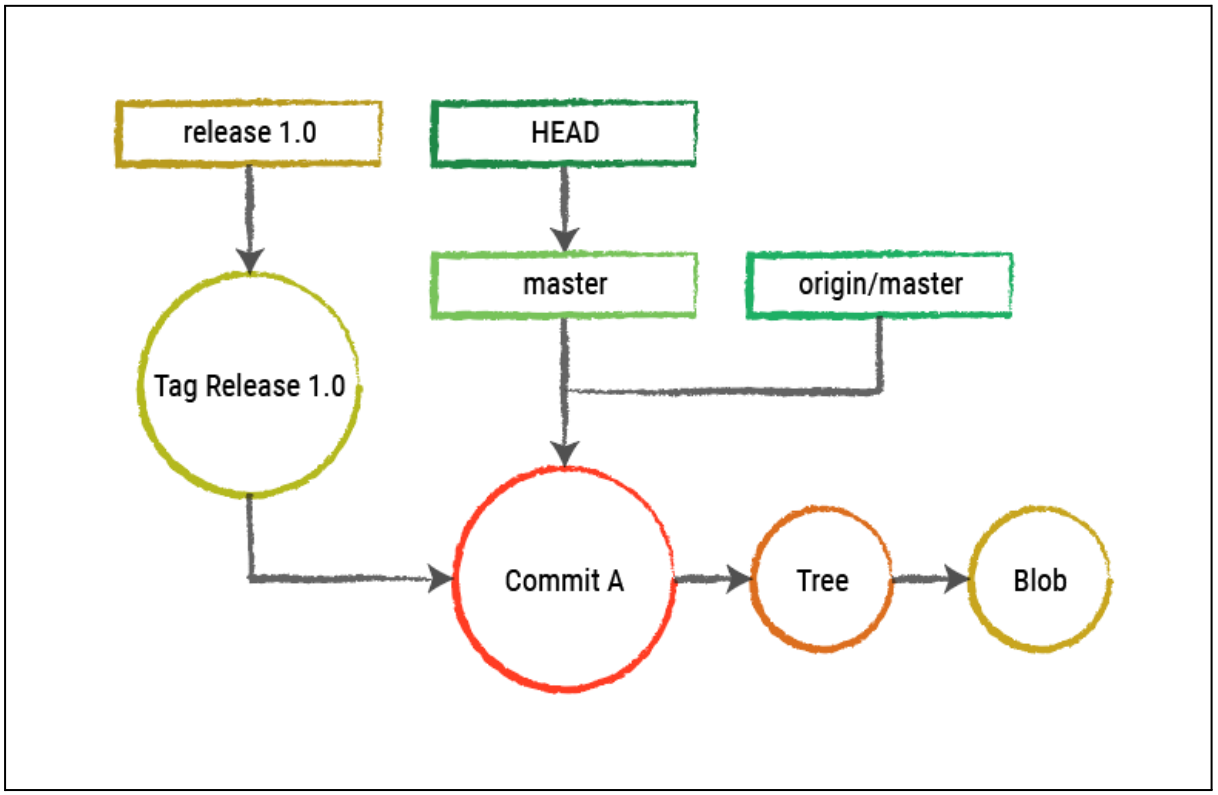

Imagen extraída de https://blog.10pines.com/2018/05/21/the-model-behind-git/

Cada círculo representa un objeto almacenado en el repositorio y los rectángulos representan las referencias que sirven como punto de entrada a los objetos.

PRINCIPALES COMANDOS DE GIT

COMANDOS DE CONFIGURACIÓN	
git config --global user.name *nombre_autor*	Configuramos el nombre del autor/a.
git config --global user.email autor@gmail.com	Configuramos el email del autor/a. Si preferís que vuestro email no sea público y vais a usar GitHub, podéis usar el que os proporciona la plataforma. https://github.com/settings/emails
OBTENER Y CREAR	
git init	Creamos un repositorio en una carpeta existente de archivos. Tenemos que estar en la carpeta que queremos enlazar con el remoto.
git clone *url_dirección_proyecto*	Obtenemos una copia de un proyecto desde un servidor
INSTANTÁNEAS BÁSICAS	
git status	Enumera los archivos que pueden ser añadidos a la *staging area*. Si usamos git status -s, nos mostrará la forma abreviada del informe. La usaremos para saber el estado actual de los archivos, para revisar si se han añadido, commiteado,...
git add *nombre_archivo*	Sube el archivo nombrado a la *staging area*.
git add.	Sube todos los archivos a la *staging area*.
git commit -m *“mensaje explicativo”*	Subimos el archivo al repositorio. Debe acompañarse de un mensaje explicativo para una mejor comprensión del estado del repositorio.

git commit	Subimos el archivo al repositorio. Te abrirá el editor de texto que tengas configurado para que puedas redactar el texto explicativo del commit. Te recomiendo que le pongas título y luego expliques con claridad y extensamente en qué consiste el commit, sobre todo si se trata del primer commit.
git restore *nombre_archivo*	Nos permite restaurar archivos o directorios a un estado anterior o recuperarlos después de borrarlos.
git reset HEAD~número git reset *commit_hash*	Nos permite mover el puntero de nuestra rama (HEAD) a un commit anterior, esencialmente reescribiendo tu historial de commits. Si quieres mantener un registro de los cambios que estás revirtiendo, `git revert` es la opción preferida. Si simplemente quieres eliminar un commit del historial, `git reset` puede ser más adecuado.
git rm *nombre_archivo*	Lo utilizamos para eliminar archivos del seguimiento de Git. Cuando ejecutas `git rm`, Git realiza dos acciones principales: 1. **Elimina el archivo de la staging area** 2. **Elimina el archivo del directorio de trabajo**

git mv *nombre_antiguoarchivo nombre_nuevoarchivo_o_ubicación*	Lo utilizamos para renombrar o mover archivos dentro de tu repositorio. Cuando utilizas `git mv`, Git realiza varias acciones: 1. **Renombra o mueve el archivo físicamente:** Al igual que el comando `mv`, cambia el nombre o la ubicación del archivo en tu sistema de archivos. 2. **Actualiza el índice:** Informa a Git sobre el nuevo nombre o ubicación del archivo, actualizando así el seguimiento. 3. **Crea un nuevo registro en el historial:** Registra este cambio como una modificación en el historial de tu repositorio.
git diff **--cached:** Compara el índice (staging area) con el último commit. **<commit1> <commit2>`:** Compara dos commits específicos.  **<branch1> <branch2>`:** Compara las puntas de dos ramas. **--stat:** Muestra un resumen estadístico de los cambios. **--name-only:** Muestra solo los nombres de los archivos que han cambiado.	Nos permite comparar dos estados de tu proyecto y ver las diferencias entre ellos. **¿Para qué sirve git diff?** ● **Revisar cambios antes de hacer un commit:** Te permite asegurarte de que los cambios que estás a punto de confirmar son los correctos. ● **Comparar diferentes versiones de un archivo:** Puedes ver cómo ha evolucionado un archivo a lo largo del tiempo. ● **Resolver conflictos de merge:** Cuando fusionas dos ramas, `git diff` te ayuda a identificar y resolver los conflictos. ● **Identificar la causa de un error:** Al comparar un commit que funciona con uno que no, puedes encontrar el cambio que introdujo el error.

RAMAS Y FUSIONES	
git branch	Nos dice las ramas que tenemos y en qué rama nos encontramos.
git branch *nombre_rama*	Creamos una rama con el nombre que decidamos.
git branch -f *rama_a_mover hash_commit_donde_se_mueve*	**Forzamos a una rama** a apuntar a un commit específico. En otras palabras, estás reescribiendo el historial de la rama para que su punta (HEAD) se mueva a la posición que tú indiques.
git branch -d *rama_a_borrar*	Borramos la rama especificada
git checkout *nombre_rama*	Nos lleva a la rama nombrada
git checkout -b *nombre_rama*	Creamos una rama con un nombre y nos movemos inmediatamente a esta.
git merge *nombre_rama_a_agregar*	Unimos main con la rama que especificamos. El commit resultante contiene la información de las dos ramas. Si existen conflictos entre ambas informaciones, Git nos pedirá resolverlos manualmente. Si no estamos en main deberemos hacer `git checkout main` y después `git merge`
git log **`--oneline`:** Muestra una línea por commit, con el hash del commit y el mensaje. **`-p`:** Muestra los cambios detallados de cada commit (diff). **`--graph`:** Muestra una representación gráfica del historial de ramas. **`--author=<autor>`:** Filtra los commits por autor.	Nos permite **visualizar el historial de cambios** que se han realizado en tu repositorio.

`--since=<fecha>`: Filtra los commits a partir de una fecha determinada. **`--until=<fecha>`:** Filtra los commits hasta una fecha determinada. **`--grep=<patrón>`:** Filtra los commits por palabras clave en el mensaje.	
git tag Ejemplo de etiqueta ligera: git tag v1.0.0 Ejemplo de etiqueta anotada: git tag **`-a`** v1.0.0 -m *"Versión inicial del producto"*	Git tag es una herramienta muy útil para organizar y documentar el historial de tu proyecto. Nos permite crear marcadores personalizados que facilitan la navegación y la gestión de tu código. **Tipos de etiquetas:** • **Etiquetas ligeras:** Son simplemente punteros a un commit específico. Son más rápidas de crear, pero tienen menos información asociada. • **Etiquetas anotadas:** Contienen más metadatos, como el autor, la fecha, un mensaje detallado y una firma GPG. Son más seguras y recomendadas para releases oficiales.
COMPARTIR Y ACTUALIZAR	
git fetch *origin_o_main*	Descargamos los cambios más recientes de un repositorio remoto (origin o main) y los almacenará en tu repositorio local. Esto incluye nuevos commits, ramas y etiquetas. Sin embargo, **no fusiona automáticamente estos cambios con tu rama actual**, es decir, no hace **merge.**

	`git fetch` es una herramienta esencial para mantener tu repositorio local sincronizado con el repositorio remoto. Diferencias entre `git fetch` y `git pull`: • `git fetch`: Descarga los cambios, pero no los fusiona. • `git pull`: Descarga los cambios y los fusiona automáticamente con tu rama actual. ¿Cuándo usar `git fetch` en lugar de `git pull`? • Antes de hacer un **merge**: Te permite revisar los cambios antes de fusionarlos. • Cuando quieres mantener tu rama local limpia: Puedes hacer varios `git fetch` y luego decidir qué cambios quieres fusionar. • Cuando trabajas con múltiples ramas: Puedes descargar los cambios de varias ramas remotas sin afectar tu rama actual.
git push *origin nombre_rama_a_subir* Este comando enviará los cambios de la rama main de tu repositorio local a la rama main del repositorio remoto llamado origin. **-u** o **--set-upstream:** Establece una relación de seguimiento entre la rama local y la remota. **-f** o **--force:** Fuerza la subida de los cambios, incluso si hay conflictos. **¡Usar esta opción con cuidado!** Puede sobrescribir cambios de otros colaboradores.	Subimos los cambios que hemos realizado en nuestro repositorio local al repositorio remoto. Es como publicar nuestros cambios para que otros colaboradores puedan verlos y trabajar con ellos.

git pull *origin rama_a_descargar*	Nos permite mantener nuestro repositorio local actualizado con los cambios de otros colaboradores y trabajar de forma más eficiente. **¿Cuándo usar git pull?** ● **Antes de empezar a trabajar:** Asegúrate de tener la última versión del código. ● **Regularmente:** Para mantener tu repositorio local sincronizado con el remoto. ● **Después de resolver conflictos:** Una vez que hayas resuelto los conflictos, puedes usar `git pull` para completar la fusión.
git remote git remote -v Esto te mostrará una lista de los repositorios remotos configurados y sus URLs. git remote add *nombre_del_repositorio_remoto URL_del_repositorio_remoto* Agregamos un nuevo repositorio remoto	Nos permite gestionar las conexiones entre nuestro repositorio local y los repositorios remotos. Es esencial para colaborar en proyectos con otros desarrolladores y para mantener una copia de seguridad de tu código.
APLICANDO CAMBIOS	
git revert *hash_del_commit*	Este comando nos permite **deshacer los cambios de un commit específico** creando un nuevo commit que invierte esos cambios. Al crear un nuevo commit que invierte los cambios, preserva el historial completo de tu proyecto y facilita la colaboración con otros desarrolladores.

git rebase *nombre_rama_destino*	Usamos git rebase para actualizar en la rama de destino, que suele ser main, las ramas en las que hemos trabajado sin afectar al mismo. En lugar de crear un nuevo commit de fusión, git rebase reescribe la historia de una rama, aplicando los cambios de una rama sobre otra. Primero, nos posicionamos en la rama creada con git checkout y luego hacemos git rebase.
git rebase -i HEAD~número	Nos permite revisar y modificar cada commit individualmente antes de rebasarlos. El número especifica cuántos commits hacia atrás quieres incluir en el rebase interactivo.
git cherrypick *hash_del_commit*	Seleccionamos commits (cambios) específicos de una rama y los aplicamos a otra. Es una herramienta muy útil cuando necesitas aplicar un cambio de una rama a otra sin tener que hacer un **merge** completo.