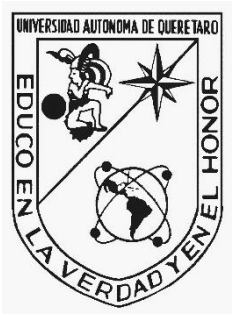

UNIVERSIDAD AUTÓNOMA DE QUERÉTARO

FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN

MANUAL PARA GIT.

Temas selectos de Estadística

Licenciatura en Actuaría

Xonthe Ashly

259027

Séptimo Semestre

Grupo 17

Profesor: Edgar Ruiz Tovar

INDICE

¿Qué es Git?

Es un sistema de control de versiones distribuido, lo que significa que es una herramienta que te permite rastrear y gestionar los cambios que se realizan en archivos a lo largo del tiempo, especialmente en proyectos de programación.

¿Qué es GitHub?

Es una plataforma en línea diseñada para que los desarrolladores de software puedan guardar, compartir y trabajar conjuntamente en proyectos de código. Piensa en ella como un almacén digital donde puedes conservar todos tus archivos de programación.

¿Para qué sirve GitHub?

Fue diseñado para almacenar, gestionar y colaborar en proyectos de código. Permite crear repositorios que actúan como un historial de cambios, facilitando la gestión de versiones. Además, promueve la colaboración en equipo, permitiendo que varios desarrolladores trabajen simultáneamente en un proyecto, con cada cambio registrado y revisable. Gracias a Git, se puede rastrear y revertir cambios, comparar versiones y entender la evolución del proyecto.

GitHub también permite compartir código, ofreciendo la opción de hacer los proyectos públicos o privados. Los proyectos públicos son accesibles para cualquier persona, fomentando la colaboración y el aprendizaje. La plataforma alberga una extensa comunidad de desarrolladores, facilitando la resolución de problemas, el aprendizaje y la contribución a proyectos de código abierto.

Antecedentes de GitHub:

Sus orígenes están relacionados con la evolución de los sistemas de control de versiones a principios del siglo XXI. Antes de Git, sistemas como SVN (Subversión) eran utilizados, aunque tenían limitaciones en escalabilidad y facilidad de uso para grandes proyectos.

Creado por Linus Torvalds, Git surgió como una solución a las limitaciones de los sistemas existentes. Fue diseñado para ser rápido, eficiente y distribuido, introduciendo conceptos como ramas locales y repositorios distribuidos, mejorando la flexibilidad y colaboración entre desarrolladores.

GitHub llevó Git a otro nivel al ofrecer una plataforma web que facilitaba su uso y fomentaba la colaboración. Añadió una interfaz gráfica intuitiva, herramientas de colaboración como pull requests y forking, y características sociales como seguir a otros usuarios y proyectos.

Desde su lanzamiento en 2008, GitHub ha crecido exponencialmente, convirtiéndose en la plataforma de desarrollo más popular del mundo. Ha sido un catalizador para el movimiento de software libre y open source, revolucionando la colaboración entre desarrolladores y fomentando la innovación al permitir compartir y construir sobre el trabajo de otros.

Clonar repositorio en Git

Imagina que tienes un proyecto de programación que está alojado en un servidor remoto, como GitHub. Este proyecto, con todo su código y su historial de cambios, se denomina **repositorio**. Al ejecutar el comando git clone seguido de la dirección web (URL) de este repositorio, estás esencialmente descargando una copia completa de ese proyecto a tu computadora.

Una vez teniendo un repositorio clonado básicamente lo que se debe hacer es que se crearía una nueva carpeta dentro de la cual tendríamos un directorio con un espacio nuevo que tu nombraste a tu parecer.

```
git clone https://github.com/usuario/mi-proyecto.git
```

Inicializar Git

Cuando se ejecuta el comando en una carpeta, Git crea un subdirectorio oculto llamado .git. Este directorio es como el "cerebro" del repositorio, donde se almacena toda la información sobre el historial de cambios, las ramas, etc. Inicializa el seguimiento de archivos: A partir de ese momento, Git empieza a "vigilar" todos los archivos dentro de esa carpeta. Esto significa que cualquier cambio que hagas en esos archivos será registrado y podrás hacer un seguimiento de su evolución.

Si estás comenzando un nuevo proyecto, git init es lo primero que debes hacer. De esta manera, asegurarás que todos los cambios que realices queden registrados y puedas volver atrás si es necesario. Si ya tienes un proyecto y quieres empezar a usar Git para controlarlo, también puedes ejecutar git init en la carpeta raíz del proyecto.

```
git init
```

Estado Git

Es una herramienta fundamental para los usuarios de Git. Este comando te ofrece una vista instantánea del estado de tu repositorio, mostrando los cambios que has hecho en los archivos pero que aún no has registrado formalmente (commits).

- Te indica en qué rama estás trabajando.
- Muestra los archivos modificados que aún no has añadido al área de preparación (staging area), y que no se incluirán en el próximo commit.
- Indica los archivos que has marcado para el próximo commit, que se añadirán al historial del proyecto al ejecutar git commit.
- Muestra los archivos creados que Git aún no está rastreando.
- Te permite ver qué archivos has modificado y cuáles estás a punto de confirmar.
- Ayuda a asegurarte de que estás confirmando los cambios correctos antes de ejecutar git commit.

```
git status
```

```
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes
 in working directory)
        modified:   README.md

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   index.html
```

Agregación de archivos Git.

El comando git add es una herramienta crucial en Git con una función específica. Piensa en tu repositorio de Git como una caja fuerte para tus archivos. git add funciona como una bolsa en la que colocas los archivos que deseas guardar en la próxima "deposición" en la caja fuerte.

- Toma los cambios en tus archivos y los deja listos para ser incluidos en el siguiente commit, añadiéndolos al área de preparación (staging area).

```
git add index.html
```

Para agregar todos los archivos que aun no han sido agregados:

```
git add .
```

- Permite seleccionar qué cambios específicos incluir en el próximo commit, en lugar de hacerlo de una sola vez.
- Ayuda a mantener un historial de cambios organizado y limpio, agrupando cambios relacionados en un solo commit.

Para archivos que han sido modificados:

```
git add -u
```

Agregar todos los cambios y nuevos archivos:

```
git add -A
```

Confirmación de proyecto Git

Una vez que has hecho cambios en tus archivos y los has puesto en el área de preparación usando git add, puedes utilizar git commit para crear una confirmación (commit). Esta acción captura una instantánea de tu proyecto en ese momento específico, incluyendo todos los cambios preparados.

- Al confirmar los cambios, se guardan de manera segura en tu repositorio local.
- Cada confirmación añade un nuevo punto en el historial del proyecto, permitiendo revertir a versiones anteriores si es necesario.
- Al confirmar, puedes añadir un mensaje que describa los cambios realizados, lo cual es útil para recordar y entender las modificaciones hechas en esa versión.

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

Visualización Git

Es una herramienta crucial en Git que permite ver el historial de modificaciones realizadas en un repositorio. Es como retroceder en el tiempo y observar la evolución del proyecto.

- Permite ver los cambios realizados, quién los hizo y por qué.
- Facilita encontrar commits con ciertas palabras clave o realizados por un usuario en particular.
- Analiza el historial que se comprende cómo ha evolucionado el proyecto y las decisiones tomadas.
- Si surge un error, permite volver a un commit anterior para deshacer los cambios problemáticos.
- Comparando diferentes commits, ayuda a aislar el cambio que introdujo un bug o una regresión.

--oneline: Muestra una línea por commit, con el hash y el mensaje.

--graph: Muestra una representación gráfica del historial de ramas.

--author="tu_nombre": Filtra los commits realizados por un autor específico.

--since="2023-01-01": Muestra los commits realizados después de una fecha determinada.

```
git log
```

Control de ramas en Git

El comando git checkout es una herramienta crucial en Git con múltiples usos. Básicamente, permite cambiar entre diferentes versiones o ramas de un proyecto.

- Si tienes varias ramas en tu proyecto, como una rama principal y otras para nuevas funcionalidades, git checkout te permite moverte entre ellas. Esto es útil para trabajar en diferentes partes del proyecto de manera aislada.

```
git checkout feature1
```

- Puedes usar git checkout para crear una nueva rama desde la rama actual, lo cual es útil para comenzar una nueva característica o corregir un error sin afectar el código principal.

```
git checkout -b bugfix
```

- Especificando un commit, git checkout restaurará tu proyecto a ese estado, lo cual es útil para deshacer cambios o regresar a un punto estable del proyecto.

```
git checkout abcdef
```

Ignorar ficheros Git

Cuando queremos ocultar un archivo y activamos este comando, se crea un archivo conocido como .gitignore que es vital en Git. Este archivo le indica a Git qué archivos o directorios no deben ser tenidos en cuenta al rastrear cambios. Esto es especialmente importante para archivos temporales, configuraciones locales y archivos compilados que no necesitan estar en el repositorio. No ayuda para:

- Programas que generan archivos temporales o de caché innecesarios para el proyecto.
- Configuraciones específicas de cada desarrollador no deben incluirse en el repositorio.
- Archivos generados a partir del código fuente no necesitan almacenarse ya que pueden regenerarse.

```
git ignore
```

```
# Ignorar todos los archivos que terminen en .o o .a
*.o
*.a

# Ignorar el directorio build
build/

# Ignorar archivos de configuración específicos
config.local.json
```

Visualización de diferencias en Git

El comando git diff es esencial en Git, ya que permite ver las diferencias entre dos estados de tu código, ya sea entre versiones de un archivo, el directorio de trabajo y el área de preparación, o entre ramas. Este comando es valioso para:

- Revisar cambios antes de realizar un commit, viendo qué modificaciones se incluirán.
- Comparar diferentes versiones de un archivo para ver su evolución.
- Resolver conflictos al fusionar ramas, identificando áreas problemáticas.

- Depurar errores al comparar una versión que funciona con una que tiene errores, para encontrar la causa.

```
git diff <opción>
```

git diff --cached: Compara el área de preparación con la última confirmación. Muestra los cambios que estás a punto de confirmar.

git diff <commit1> <commit2>: Compara dos commits específicos.

git diff <branch1> <branch2>: Compara las últimas confirmaciones de dos ramas.

Restaurar cambios Git

El comando git reset es una herramienta muy poderosa en Git para revertir cambios en tu repositorio, aunque debe usarse con precaución porque puede modificar el historial del proyecto que puede ser irregenerable.

- Te permite mover el puntero de la rama (HEAD) a un commit anterior, descartando los commits posteriores. Útil si necesitas revertir un commit erróneo.
- Permite revertir cambios añadidos al área de preparación pero aún no confirmados.
- Puedes ajustar el último commit, por ejemplo, para cambiar el mensaje o añadir archivos olvidados.

--soft: Mueve HEAD al commit especificado, manteniendo los cambios en el área de preparación.

--mixed: Mueve HEAD al commit especificado y elimina los cambios del área de preparación.

--hard: Mueve HEAD al commit especificado, eliminando los cambios del área de preparación y descartando los cambios locales no confirmados.

```
git reset --hard HEAD~2
```

Registro de operaciones Git

El comando git reflog es una herramienta valiosa en Git que permite visualizar un registro de todas las operaciones realizadas en un repositorio. Funciona como un historial de cambios, incluso de aquellos que ya no están directamente referenciados por ninguna rama.

- Si borras accidentalmente un commit importante o mueves la rama principal incorrectamente, git reflog te muestra una lista de todas las acciones realizadas en tu repositorio, incluyendo los hashes de los commits.
- Hash del commit: Un identificador único para cada commit.
- Indica cuándo se realizó la acción.
- Detalla acciones como checkout, reset, merge, etc.
- Muestra la rama donde se realizó la acción.

```
git reflog
```

Marcado de puntos específicos Git

El comando git tag en Git permite marcar puntos específicos en la historia de un proyecto. Sirve para identificar versiones, marcar hitos importantes y crear referencias. Hay dos tipos de etiquetas: etiquetas ligeras, que son simples marcadores de commits, y etiquetas anotadas, que incluyen información adicional como el autor, la fecha y un mensaje descriptivo.

- Marcar hitos: Etiquetar eventos importantes en el desarrollo, como nuevas características o la corrección de errores críticos.
- Crear referencias: Utilizar las etiquetas para volver rápidamente a un punto específico en el historial del proyecto.

Tipos de etiquetas:

- Etiquetas ligeras: Simples marcadores que apuntan a un commit específico.

```
git tag <nombre_de_la_etiqueta> <commit>
```

- Etiquetas anotadas: Además de apuntar a un commit, incluyen información adicional como el autor, la fecha y un mensaje descriptivo, proporcionando más contexto.

```
git tag -a v1.0 -m "Versión 1.0 lanzada"
```

Gestión de ramas Git

El comando git branch es una herramienta fundamental en Git para administrar las ramas de tu proyecto. Las ramas actúan como líneas de tiempo paralelas, permitiendo que desarrolles diferentes características o versiones de tu proyecto de manera aislada.

- Trabaja en varias funcionalidades o versiones al mismo tiempo sin interferencias.
- Prueba nuevas ideas en ramas separadas sin afectar el código principal.
- Asigna diferentes ramas a distintos miembros del equipo para tareas específicas.

Inicia el trabajo en una nueva característica o corrige errores sin afectar el código principal.

```
git branch feature1
```

Muestra todas las ramas en tu proyecto y te indica en cuál estás trabajando actualmente.

```
git branch
```

Permite moverte entre ramas para trabajar en diferentes partes del proyecto.

```
git checkout feature1
```

Borra ramas que ya no necesites después de completar su desarrollo.

```
git branch -d feature1
```

Cambio de ramas Git

El comando git switch es esencial en Git, permitiendo cambiar de rama de manera fácil y eficiente. Las ramas son como diferentes caminos en un árbol, cada una representando una línea de desarrollo independiente de tu proyecto. Esto permite trabajar en distintas funcionalidades o versiones sin afectar otras áreas del proyecto. Con git switch, puedes cambiar rápidamente entre ramas, crear nuevas ramas desde la actual y eliminar ramas que ya no se necesiten.

- Hace que cambiar de rama sea más sencillo e intuitivo.
- Al trabajar en ramas separadas, el historial de cambios es más claro y fácil de seguir.
- Varios desarrolladores pueden trabajar simultáneamente en diferentes partes del proyecto.

Históricamente, git checkout se utilizaba tanto para cambiar de rama como para restaurar archivos. Para simplificar y evitar confusiones, se introdujo git switch exclusivamente para cambiar de rama, mientras que git checkout se reserva para otras operaciones como la restauración de archivos.

- Te permite moverte rápidamente de una rama a otra, para trabajar en diferentes partes de tu proyecto.

```
git switch feature-nueva-funcionalidad
```

- Puedes crear una nueva rama a partir de la rama actual.

```
git switch -c nueva-rama
```

- También puedes eliminar ramas que ya no necesites.

```
git branch -d nombre_de_la_rama
```

Combinación de proyectos Git

El comando git merge es esencial en Git para combinar el trabajo de diferentes ramas y unificar el historial de desarrollo del proyecto. Este comando permite unir ramas, resolver conflictos y avanzar el historial mediante la creación de un nuevo commit que representa la fusión de ambas líneas de desarrollo. Existen dos tipos de fusiones: rápidas y con conflictos, cada una gestionada de manera específica.

- Facilita el trabajo simultáneo de múltiples desarrolladores.
- Permite desarrollar nuevas funcionalidades sin afectar el código principal.
- Mantiene un historial de desarrollo organizado y comprensible.

Tipos de fusión:

- Fusión rápida: Si no hay conflictos, Git realiza una fusión rápida, creando un nuevo commit que apunta a la última confirmación de ambas ramas.
- Fusión con conflictos: Si hay conflictos, Git detendrá la fusión y te mostrará los archivos con conflictos. Tendrás que resolver estos conflictos manualmente y luego utilizar git add para marcar los conflictos como resueltos y git commit para completar la fusión.

```
git merge feature
```

Guardado temporal en Git

El comando git stash es muy útil en Git para guardar temporalmente los cambios no confirmados en tu directorio de trabajo. Permite "pausar" esos cambios para trabajar en otra tarea y recuperarlos después. Funciona guardando una instantánea de los cambios en el área de preparación y el directorio de trabajo, limpiando el directorio a su estado anterior al último commit, y permitiendo recuperar los cambios con git stash pop o git stash apply.

- Guarda los cambios: Cuando ejecutas git stash, Git toma una instantánea de los cambios que tienes en el área de preparación (staging area) y en el directorio de trabajo, y los almacena en una pila llamada "stash".

```
git stash
```

Cambios mas recientes del stash y elimina.

```
git stash pop
```

- Limpia el directorio de trabajo: Después de guardar los cambios, Git vuelve a dejar tu directorio de trabajo en el estado en el que estaba en tu último commit.

```
git stash list
```

Elimina un stash en especifico.

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

Elimina todos los stashes.

```
git stash clear
```

- Recuperar los cambios: Puedes recuperar los cambios guardados en el stash usando el comando git stash pop o git stash apply.

```
git stash apply
```

Admistración de conexiones Git

El comando git remote es una herramienta fundamental en Git que te permite administrar las conexiones con otros repositorios. Estos repositorios externos, llamados "remotos", son como espejos de tu repositorio local. Te permiten:

- Subir tus cambios a un repositorio remoto para que otros puedan acceder a ellos.
- Trabajar en conjunto con otros desarrolladores en el mismo proyecto.
- Tener una copia de seguridad de tu código en un servidor remoto.

Te ayuda a:

- Simplifica la gestión de repositorios: Te permite trabajar con múltiples repositorios de forma sencilla.
- Permite hacer copias de seguridad: Protege tu trabajo en caso de problemas en tu equipo local.

Agregar un remoto nuevo.

```
git remote add origin https://github.com/tu_usuario/tu_repositorio.git
```

Ver remotos existentes.

```
git remote -v
```

Eliminar un remoto.

```
git remote remove origin
```

Renombre un remoto.

```
git remote rename <nombre_actual> <nuevo_nombre>
```

Mostrar información de un remoto.

```
git remote show <nombre>
```

Sincronización de repositorio local con repositorio remoto

El comando git fetch es fundamental en Git para actualizar tu repositorio local con los cambios realizados en un repositorio remoto. Cuando trabajas en un proyecto en equipo, cada miembro tiene su copia local, y estas pueden desactualizarse a medida que se realizan cambios en el repositorio remoto. Git fetch descarga la nueva información (commits, ramas, etiquetas) desde el remoto y la guarda en tu repositorio local, sin integrar automáticamente estos cambios en tu rama actual.

- Permite conocer los cambios en el repositorio remoto sin modificar tu trabajo actual.
- Facilita la revisión de cambios antes de decidir si integrarlos en tu rama.
- Descargar los cambios por separado ayuda a resolver conflictos antes de integrarlos.

Diferencia entre git fetch y git pull:

- git fetch: Descarga los cambios del remoto pero no los integra en tu rama actual.
- git pull: Combina git fetch y git merge, descargando e integrando automáticamente los cambios en tu rama actual.

```
git fetch origin
```

Sincronización en Git

El comando git pull es una herramienta esencial en Git que te permite sincronizar fácilmente tu repositorio local con un repositorio remoto. Si trabajas en equipo y cada miembro tiene su propia copia del proyecto, las actualizaciones remotas pueden hacer que tu copia local quede desactualizada. Sirve para:

- Similar a git fetch, descarga todos los cambios nuevos (commits, ramas, etiquetas) desde el repositorio remoto.

- A diferencia de git fetch, git pull intenta integrar automáticamente los cambios descargados en tu rama actual, fusionando las modificaciones más recientes del remoto con tu trabajo local.

Función:

- Asegura que tu copia local esté siempre al día con los últimos cambios de los demás miembros del equipo.
- Permite que todos los miembros del equipo trabajen con la versión más reciente del código.
- Intenta resolver automáticamente los conflictos que puedan surgir al integrar los cambios.

```
git pull origin main
```

Subida de cambios en Git

El comando git push es una herramienta clave en Git para compartir tus cambios con otros desarrolladores o para hacer una copia de seguridad en un repositorio remoto. Git push actúa como un botón de "enviar", permitiéndote subir los cambios realizados en tu repositorio local al remoto. Sus funciones son:

- Transfiere los commits de tu rama local al repositorio remoto.
- Mantener el repositorio remoto actualizado con los cambios más recientes.
- Permitir que otros desarrolladores vean y utilicen tus cambios.

Es importante porque:

- Facilita el trabajo en equipo al permitir que varios desarrolladores trabajen.
- Crea una copia de seguridad de tu trabajo en un servidor remoto.
- Permite rastrear los cambios realizados en el proyecto a lo largo del tiempo.

Se usa:

- Cuando hayas terminado de trabajar en una característica o corregido un error.
- Cuando quieras que otros vean y usen tus cambios.

```
git push origin main
```

Revertir los cambios de Git

El comando git revert es una herramienta eficaz en Git que te permite revertir los cambios introducidos por un commit específico, generando un nuevo commit que deshace dichos cambios.

- Git revert mantiene intacto el historial del proyecto, lo cual es vital para rastrear cambios y comprender la evolución del código.
- En lugar de borrar el commit original, git revert crea uno nuevo que revierte los cambios, facilitando la comprensión del historial.
- Al generar un nuevo commit, se reduce el riesgo de perder información crucial o causar conflictos con otras ramas.
- Utiliza el hash del commit o el nombre de una rama para especificar el commit a revertir.
- Git revert genera un commit que deshace los cambios realizados por el commit original.
- Este nuevo commit se incorpora al historial del proyecto

git revert vs git reset:

Aunque ambos comandos sirven para deshacer cambios, tienen diferentes efectos. git revert crea un nuevo commit, mientras que git reset puede modificar el historial de commits de forma más drástica.

```
git revert <hash_del_commit>
```

Selección de commit en Git

El comando git cherry-pick es una herramienta poderosa en Git que permite seleccionar un commit específico de una rama y aplicarlo en otra. Es como tomar una "cereza" (commit) de un árbol (rama) y ponerla en otro.

Se utiliza para:

- Tomar un cambio realizado en una rama y aplicarlo a otra sin necesidad de hacer una fusión completa.
- Si un error se solucionó en una rama de desarrollo, cherry-pick permite aplicar esa corrección de manera selectiva en la rama principal.
- Útil para revertir un commit específico en otra rama.

```
git cherry-pick <hash_del_commit>
```

Identificación de introducción de un cambio no deseado en Git}

El comando git bisect es una herramienta muy útil en Git que te permite encontrar el commit específico que introdujo un error o cambio no deseado en tu código, utilizando una técnica similar a la "búsqueda binaria" para localizar el commit problemático. Forma de funcionar:

- Identificas un commit "bueno" donde el código funcionaba correctamente y un commit "malo" donde ya está presente el error.
- Git selecciona un commit intermedio y te solicita verificar si el error estaba presente en ese punto.
- Basado en tu respuesta, Git ajusta su búsqueda y selecciona otro commit intermedio.
- Este proceso se repite hasta encontrar el commit exacto que introdujo el error.

Sirve para:

- Encontrar la causa raíz de errores en tu código.
- Identificar qué cambio específico causó un problema.
- Poder revertir o aislar el cambio que causó el error.

```
git bisect start
git bisect good <commit_bueno>
git bisect bad <commit_malo>
```

Ubicación de quien realizo el ultimo cambio en Git

El comando git blame es una herramienta muy útil en Git que te permite descubrir quién hizo el último cambio en cada línea de un archivo. Es como un detective para tu código, ayudándote a encontrar al responsable de una determinada sección de código. Sirve para:

- Mencionar quién hizo el último cambio en esa línea y en qué commit.
- Ayudarte a determinar quién es el propietario de una sección específica de código.
- Puede ayudarte a rastrear el cambio que introdujo el problema.

Funcionamiento:

- Cuando ejecutas git blame en un archivo, Git analiza cada línea y te muestra:
- El hash del commit en el que se realizó el último cambio en esa línea.
- El nombre de la persona que hizo el commit.
- La fecha en que se realizó el commit.

```
git blame index.html
```

Trabajo en múltiples ramas en Git

El comando git worktree es una herramienta poderosa en Git que permite trabajar en varias ramas de un mismo repositorio simultáneamente, cada una en su propio directorio de trabajo.

- Puedes hacer cambios en una rama sin interferir en otras.
- Facilita ver las diferencias y la evolución del proyecto.
- Permite probar nuevas ideas sin afectar la rama principal.

Funcionamiento:

- Se crea un nuevo directorio en tu sistema con el contenido de la rama especificada.
- Los cambios en un directorio no afectan a los otros.
- Puedes cambiar entre diferentes árboles de trabajo usando los comandos git worktree list para ver los existentes y git checkout -w <nombre_del_árbol> para cambiar.

```
git worktree add ../my-feature-branch feature
```

Incorporación de repositorio Git en un subdirectorio

El comando git submodule en Git permite incluir un repositorio completo como un subdirectorio dentro de otro repositorio. Esto posibilita tener un proyecto principal que referencia a otros proyectos más pequeños, gestionando su versión de forma independiente.

- Permite organizar el proyecto en módulos más pequeños y reutilizables.
- Cada submódulo conserva su propio historial de versiones.
- Si un componente se usa en varios proyectos, puedes mantener una única copia en un submódulo.
- Facilita una gestión más eficiente de las dependencias del proyecto.

Al añadir un submódulo, Git guarda una referencia al commit específico del submódulo en ese momento. Si el submódulo se actualiza, debes actualizar manualmente la referencia en el repositorio principal.

Comandos principales relacionados:

git submodule add: Añade un nuevo submódulo al repositorio.

git submodule update: Actualiza los submódulos a la última versión.

git submodule status: Muestra el estado de los submódulos.

git submodule init: Inicializa los submódulos en un repositorio clonado.

Para añadir:

```
git submodule add https://github.com/usuario/mi-biblioteca.git mi-biblioteca
```

Para actualizar:

```
git submodule update --init --recursive
```

Reorganización del historial en Git

El comando git rebase es una herramienta poderosa en Git que te permite reorganizar y reescribir el historial de commits de una rama. Imagina que tu historial de commits es como una línea del tiempo de los cambios que has realizado en tu proyecto. Con git rebase, puedes modificar esta línea del tiempo de diversas formas.

- Puedes combinar múltiples commits en uno solo, eliminar commits innecesarios o reordenar los commits para crear un historial más limpio y fácil de entender.
- Si tienes dos ramas que se han desviado mucho, puedes usar git rebase para integrar los cambios de una rama en otra de una manera más limpia que con git merge.

- Puedes editar mensajes de commit, cambiar el orden de los commits o incluso descartar commits.

En esencia, git rebase toma una serie de commits y los vuelve a aplicar sobre otro punto de partida. Esto significa que estás reescribiendo el historial de commits.

- Cuando quieres preservar el historial completo: merge crea un nuevo commit que muestra la fusión de las dos ramas, preservando el historial completo.
- Cuando estás trabajando en una rama pública: Es más seguro usar merge en ramas públicas para evitar conflictos con otros desarrolladores.

Rebase de una rama sobre otra.

```
git checkout rama_destino
git rebase rama_origen
```

Rebase interactivo.

```
git rebase -i HEAD~3
```

```
C4 - feat: Nueva funcionalidad
C3 - fix: Corrección de bug
C2 - feat: Otra funcionalidad
C1 - feat: Funcionalidad inicial
```

```
pick C4 feat: Nueva funcionalidad
pick C3 fix: Corrección de bug
pick C2 feat: Otra funcionalidad
```

Fork

Un fork en Git es como hacer una copia de un repositorio para poder trabajar en él de forma independiente. Es como tener tu propia versión del proyecto original, donde puedes hacer cambios, experimentar y realizar pruebas sin afectar el repositorio original.

- Si quieres contribuir a un proyecto open source pero no tienes permisos de escritura en el repositorio original, puedes hacer un fork. Esto te permite hacer tus cambios en tu propia copia y luego enviar una solicitud de fusión (pull request) al proyecto original para que tus cambios sean considerados.
- Si quieres modificar un proyecto existente para adaptarlo a tus necesidades específicas, un fork te permite hacer los cambios sin afectar el proyecto original.
- Puedes utilizar un fork para probar nuevas ideas, funcionalidades o tecnologías sin arriesgar el proyecto original.

Funcionamiento:

- Cuando haces un fork de un repositorio, se crea una copia exacta en tu cuenta.
- Puedes hacer todos los cambios que quieras en tu fork.
- Si quieres que tus cambios sean incluidos en el proyecto original, puedes enviar una solicitud de fusión (pull request). El propietario del proyecto original revisará tus cambios y decidirá si los incorpora o no.