Parte 2.

1. **git merge**:
    - **Fusión de ramas**: **git merge** se utiliza para fusionar los cambios de una rama en otra rama. Por lo general, se fusiona una rama secundaria en una rama principal (por ejemplo, fusionar una rama de características en la rama principal).
    - **Historia de confirmaciones**: Cuando se utiliza **git merge**, se conserva la historia de confirmaciones de ambas ramas. Esto significa que se creará una nueva confirmación de fusión que indica dónde se combinaron las ramas.
    - **Fácil de entender**: **git merge** es más sencillo de entender visualmente en el historial de confirmaciones, ya que muestra claramente cuándo se hicieron las fusiones.
    - **Preserva la historia original**: Al usar **git merge**, la historia original de cada rama se conserva, lo que puede ser útil para rastrear el historial de cambios en cada rama por separado.

    Ejemplos:

    - Git checkout main
    - Git merge feature-branch

2. **git rebase**:
    - **Reorganización de la historia**: **git rebase** se utiliza para reorganizar la historia de confirmaciones al mover todas las confirmaciones de una rama sobre otra base. Esto significa que parece que los cambios de la rama secundaria se hicieron directamente sobre la rama principal, como si hubieran sido creados allí desde el principio.
    - **Historia lineal**: Con **git rebase**, obtienes una historia de confirmaciones más lineal, lo que puede hacer que el historial sea más limpio y más fácil de seguir.
    - **Puede reescribir la historia**: Un aspecto importante a tener en cuenta es que **git rebase** puede reescribir la historia de una rama. Esto significa que, si alguien más está trabajando en la misma rama y ha compartido cambios, podrías causar problemas al reescribir la historia.
    - **Uso común en flujos de trabajo de características**: **git rebase** a menudo se utiliza en flujos de trabajo de ramas de características para mantener un historial más limpio y lineal.

Ejemplos:

- Git checkout feature-branch
- Git rebase main

En resumen, **git merge** es útil para fusionar cambios y conservar la historia original de las ramas, mientras que **git rebase** es útil para crear una historia más lineal y reorganizada, pero puede reescribir la historia de confirmaciones. La elección entre Uno u otro depende de tu flujo de trabajo y de cómo desees que se vea y se organice el historial de confirmaciones de un Proyecto.

Comandos para revertir un commit sin eliminar el historial:

Para revertir un commit que ya ha sido publicado en una rama compartida sin eliminar el historial, puedes utilizar el siguiente comando:

Git revert <hash_del_commit>

Este comando creará un nuevo commit que deshace los cambios introducidos por el commit especificado, manteniendo intacto el historial de cambios. Es una forma segura de deshacer cambios sin alterar la historia existente.

Tres buenas prácticas al escribir mensajes de commit:

1. **Sé descriptivo:** El mensaje debe ser lo suficientemente claro como para que otros desarrolladores puedan entender qué cambios introduces con el commit sin necesidad de revisar el código. Utiliza un lenguaje claro y preciso.
2. **Limita la longitud:** Los mensajes de commit deben ser concisos y no exceder las 72-80 columnas de ancho. Esto facilita la lectura en la línea de comandos y en herramientas de seguimiento de cambios.
3. **Usa un formato consistente:** Puedes seguir un formato estándar como "Título en una línea" y "Descripción opcional en líneas adicionales". También puedes usar convenciones como las definidas en las pautas de commit de Git (por ejemplo, "fix:", "feat:", "docs:", etc.) para indicar el propósito del commit.

¿Qué es un .gitignore y por qué es esencial en un proyecto?

Un archivo. gitignore es un archivo de configuración utilizado por Git para especificar patrones de archivos y directorios que deben ser ignorados por el control de versiones. Esto significa que los archivos y directorios enumerados en el archivo .gitignore no se incluirán en los commits ni se rastrearán por Git.

La importancia de un archivo **.gitignore** radica en que ayuda a mantener limpio y organizado el repositorio de Git al evitar que archivos y directorios innecesarios o sensibles se incluyan en el control de versiones. Esto es esencial para:

1. Evitar archivos generados automáticamente: Muchos lenguajes de programación generan archivos temporales o de compilación que no deben incluirse en el repositorio (por ejemplo, archivos **.pyc**, **.class**, **.log**, etc.).
2. Proteger información sensible: Puedes evitar la inclusión accidental de contraseñas, claves de API u otros datos sensibles al agregarlos al **.gitignore**.
3. Reducir el tamaño del repositorio: Excluir archivos binarios grandes o directorios con muchos archivos de compilación puede ayudar a mantener un repositorio más pequeño y ágil.

Parte 3.

Un "Pull Request" (PR) es una función proporcionada por sistemas de control de versiones colaborativos como GitHub, GitLab y Bitbucket. Un PR es una solicitud que un desarrollador hace para fusionar sus cambios (commits) desde una rama (branch) de su repositorio a otra rama en el repositorio principal del proyecto. Los beneficios de un PR en un proceso de desarrollo colaborativo incluyen:

1. **Revisión y Colaboración:** Los PR permiten a otros desarrolladores revisar y discutir los cambios propuestos antes de que se fusionen en la rama principal. Esto fomenta la colaboración y mejora la calidad del código al permitir la detección temprana de errores o problemas de diseño.
2. **Seguimiento y Documentación:** Los PRs proporcionan un registro claro y documentado de los cambios que se han realizado en el proyecto. Cada PR incluye una descripción detallada de los cambios y los comentarios de revisores, lo que facilita el seguimiento del desarrollo y la toma de decisiones informadas.

Diferencia entre un "branch" y un "fork" en GitHub:

- **Branch (Rama):** Un branch es una bifurcación temporal de una rama principal (por lo general, la rama **main** o **master**) en un repositorio. Los branches se utilizan para desarrollar nuevas funcionalidades o solucionar problemas sin afectar la rama principal. Los cambios realizados en un branch se pueden fusionar de vuelta en la rama principal a través de un PR una vez que estén listos. Los branches son locales en el repositorio original y se pueden crear y gestionar directamente desde ese repositorio.
- **Fork (Bifurcación):** Un fork es una copia completa de un repositorio en una cuenta de usuario diferente. Los forks se utilizan cuando un colaborador quiere contribuir a un proyecto del cual no tiene permisos de escritura. El colaborador crea un fork, realiza cambios en su propia copia y luego envía un PR al repositorio original para proponer los cambios. Los forks son repositorios independientes y separados del original, lo que permite una mayor separación y control.

Ventajas de utilizar un flujo de trabajo como "Git Flow" en un proyecto de desarrollo:

"Git Flow" es un modelo de flujo de trabajo para proyectos de desarrollo que define una estructura de ramas y reglas claras para la gestión de características, lanzamientos y correcciones. Dos ventajas clave de utilizar Git Flow son:

1. **Organización y Estructura Clara:** Git Flow establece una estructura de ramas clara y organizada que facilita la gestión de características, lanzamientos y correcciones. Esto ayuda a mantener el repositorio ordenado y permite a los equipos colaborar de manera eficiente.
2. **Control sobre Versiones y Estabilidad:** Git Flow promueve la idea de tener ramas específicas para lanzamientos (como **release** y **hotfix**) que permiten un control preciso sobre las versiones del software. Esto garantiza que las versiones se puedan preparar y lanzar de manera más predecible y que las correcciones de errores críticos se puedan aplicar de manera rápida y segura.