PREGUNTAS DEL PARCIAL

- **Explica la diferencia entre git merge y git rebase**

  Git merge y git rebase son dos formas diferentes de combinar cambios en un repositorio de Git:

  1. **Git Merge**:
     - Crea un nuevo commit que tiene dos padres: el commit actual y el commit que se está fusionando.
     - Mantiene la historia original del proyecto intacta, preservando la secuencia de commits.
     - Es útil para fusionar ramas independientes o cuando se desea mantener un registro detallado de la historia de desarrollo.

  2. **Git Rebase**:
     - Reorganiza la historia de commits al mover los commits de una rama sobre la punta de otra rama.
     - Crea una historia lineal y más limpia sin bifurcaciones.
     - Puede dar la impresión de que los cambios de una rama se aplicaron directamente sobre la otra.
     - Es útil para mantener una historia de desarrollo más simple y ordenada, pero se debe usar con precaución en repositorios compartidos.

  En resumen, Git merge fusiona ramas manteniendo la historia original, mientras que Git rebase reorganiza la historia en una línea más limpia, pero altera la secuencia de commits. La elección entre ellos depende de la preferencia y los objetivos del proyecto.

- **¿Qué comando(s) utilizarías para revertir un commit que ya ha sido publicado en una rama compartida sin eliminar el historial?**

  Para revertir un commit que ya ha sido publicado en una rama compartida sin eliminar el historial, puedes usar el comando “git revert”. Este comando crea un nuevo commit que deshace los cambios introducidos por el commit que deseas revertir. Aquí está la forma de hacerlo:

  git revert <hash-del-commit-a-revertir> => git revert abc123

  Esto creará un nuevo commit que deshace los cambios del commit “abc123”, pero conserva el historial de revisiones completo y público. Luego, puedes empujar este nuevo commit a la rama compartida para que otros colaboradores puedan ver la reversión y su motivo en el historial del repositorio compartido.

- **Menciona y describe tres buenas prácticas al escribir mensajes de commit.**

  Escribir mensajes de commit informativos y bien estructurados es una buena práctica fundamental en el desarrollo de software. Aquí te menciono tres buenas prácticas clave al escribir mensajes de commit:

  1. Mantén los mensajes concisos y descriptivos:

  Los mensajes de commit deben ser breves pero informativos. Evita mensajes vagos o genéricos.

  Describe de manera clara y concisa qué cambios introduces en el commit.

  Incluye detalles relevantes, como por qué se realiza el cambio y cómo afecta al código. Ejemplo de un buen mensaje de commit:

  Corrige un error de ortografía en el archivo README.md

  2. Usa un formato consistente:

  Adopta un formato consistente para tus mensajes de commit. Una convención popular es el formato *"Subject" (Título) + "Body" (Cuerpo)*, separados por una línea en blanco.

  El título debe ser breve (50-72 caracteres) y expresar de manera clara la esencia del cambio.

  El cuerpo es opcional, pero puede ser útil para proporcionar detalles adicionales cuando sea necesario.

  Ejemplo de un mensaje de commit con formato consistente:

  Agrega validación de entrada de usuario

  La nueva función de validación garantiza que los datos ingresados por el usuario cumplan con los requisitos de formato especificados en la documentación.

  3. Usa imperativos en el título:

  Escribe el título del mensaje de commit en tiempo presente y en modo imperativo. Esto indica lo que hace el commit, no lo que hizo.

  Usa verbos como "Agrega", "Corrige", "Actualiza" en lugar de "Agregó", "Corrigió", "Actualizó".

  Ejemplo de un mensaje de commit con imperativo en el título:

  Corrige la lógica de validación de formularios

- **¿Qué es un .gitignore y por qué es esencial para un proyecto?**

  Un archivo .gitignore es un archivo de configuración en un repositorio de Git que especifica patrones de archivos y directorios que Git debe ignorar al realizar operaciones como git status, git add, y git commit. Estos patrones indican a Git qué archivos y carpetas deben ser excluidos del seguimiento y control de versiones.

- **¿Qué es un "Pull Request" y qué beneficios ofrece en un proceso de desarrollo colaborativo?**

  Un "Pull Request" (PR), en el contexto de sistemas de control de versiones como Git, es una solicitud que un colaborador de un proyecto hace a los mantenedores del repositorio para que revisen y, potencialmente, fusionen los cambios que ha realizado en una rama de código a la rama principal del repositorio. Aunque el nombre "Pull Request" es común en plataformas como GitHub, GitLab y Bitbucket, en otros contextos también puede llamarse "Merge Request" u otros términos similares.

- **Explica la diferencia entre un "branch" y un "fork" en el contexto de GitHub.**

  En el contexto de GitHub, un "branch" y un "fork" son dos conceptos relacionados pero diferentes que se utilizan para administrar y colaborar en proyectos de código abierto.

  Branch (Rama):

  Un "branch" (rama) es una copia de la línea de desarrollo principal (generalmente la rama main o master) de un repositorio de GitHub. Cada vez que creas un nuevo "branch", estás creando una copia independiente del código en un punto específico del historial del repositorio.

  Los "branches" se utilizan para trabajar en nuevas características, correcciones de errores u otras modificaciones sin afectar la rama principal del repositorio.

  Puedes realizar cambios en un "branch", crear commits en él y, cuando estés satisfecho con los cambios, fusionarlo de nuevo en la rama principal a través de un Pull Request (solicitud de extracción) después de que tus cambios hayan sido revisados y aprobados.

  Fork (Bifurcación):

  Un "fork" (bifurcación) es una copia completa de un repositorio de GitHub, incluyendo todas las ramas, commits, historial y archivos.

  Normalmente, se crea un "fork" cuando quieres contribuir a un proyecto de código abierto que no es de tu propiedad o al que no tienes acceso de escritura directa.

Puedes realizar cambios en tu propia copia del repositorio ("fork") como si fuera tu propio repositorio, creando "branches" y realizando commits en ellos. Luego, puedes enviar Pull Requests desde tu "fork" al repositorio original para proponer cambios que los mantenedores del proyecto original pueden revisar y fusionar.

- **Menciona y describe dos ventajas de utilizar un flujo de trabajo como "Git Flow" en un proyecto de desarrollo.**

  "Git Flow" es un flujo de trabajo de Git popular que define una estructura de ramificación y una serie de reglas para administrar el ciclo de vida del desarrollo de software. Dos ejemplos de “Git Flow” son:

  Organización y estructura clara:

  Git Flow establece una estructura de ramificación clara y predecible que incluye ramas principales como master (para versiones estables) y develop (para desarrollo en curso).

  También define ramas temporales para características (feature), lanzamientos (release), correcciones de errores (hotfix), y soporte (support).

  Esta estructura proporciona un marco ordenado para el desarrollo, facilitando la organización de tareas y la colaboración en equipo.

  Cada tipo de rama tiene un propósito específico, lo que simplifica la toma de decisiones sobre dónde realizar cambios y cómo fusionarlos.

  Control de versiones y estabilidad:

  Git Flow promueve un enfoque de control de versiones robusto y escalable.

  La rama master se utiliza exclusivamente para las versiones estables del software, lo que garantiza que siempre haya una versión confiable y funcional disponible para los usuarios finales.

  Las ramas de características (feature) y lanzamientos (release) permiten que las nuevas características se desarrollen de manera aislada y que las versiones se preparen cuidadosamente antes de fusionarlas en master.

  Las ramas de corrección de errores (hotfix) permiten abordar rápidamente problemas críticos en versiones en producción sin perturbar el desarrollo en curso.